VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Ergonomics and efficiency of workflows on HPC clusters

Ergonomie a efektivita workflow na HPC klastrech

Ing. Jakub Beránek

PhD Thesis

Supervisor: Ing. Jan Martinovič, Ph.D.

Ostrava, 2024

Abstrakt a přínos práce

Tato práce se zabývá spouštěním grafů úloh na vysoce výkonných systémech (superpočítačích), se zaměřením na efektivní využití výpočetních zdrojů a poskytnutí ergonomických rozhraní pro návrh a spouštění grafů úloh. Programování na základě úloh je oblíbeným způsobem pro definici vědeckých soustav výpočtů, které jsou určeny pro spouštění na distribuovaných systémech. Nicméně spouštění těchto úloh na superpočítačích přináší unikátní výzvy, například problémy s výkonem způsobené značným rozsahem úloh nebo problematickou interakci úloh se systémy pro správu alokací na superpočítačích, jako jsou například PBS (Portable Batch System) nebo Slurm. Tato práce zkoumá, jaké jsou hlavní problémy ovlivňující spouštění úloh v této oblasti a navrhuje různé přístupy, které by měly pomoci tyto problémy částečně či zcela vyřešit, a to jak v oblasti výkonu, tak i ergonomie vývoje.

Tato práce poskytuje tři hlavní přínosy. Prvním z nich je prostředí pro simulaci spouštění grafů úloh, které umožňuje jednoduché experimentování a měření různých plánovacích algoritmů. Toto prostředí bylo použito pro provedení rozsáhlé studie kvality různých plánovačů úloh. Dále práce analyzuje výkonnostní charakteristiku moderního nástroje pro spouštění úloh Dask, a poskytuje alternativní implementaci Dask serveru, která výrazně zvyšuje jeho efektivitu v případech, které vyžadují vysoký výkon. Hlavním přínosem práce je metoda pro plánování úloh a správu zdrojů, která umožňuje jednoduché spouštění grafů úloh na heterogenních superpočítačích, které zároveň maximalizuje využití dostupných výpočetních zdrojů. Práce také poskytuje referenční implementaci využívající této metody v rámci nástroje HyperQueue, který je dostupný jako software s otevřeným zdrojovým kódem pod licencí MIT (Massachusetts Institute of Technology) na adrese `https://github.com/it4innovations/hyperqueue`.

Klíčová slova

distribuované výpočty, výpočetní grafy, heterogenní zdroje, vysoce výkonné počítání

Abstract and Contributions

This thesis deals with the execution of task graphs on High-performance Computing (HPC) clusters (supercomputers), with a focus on efficient usage of hardware resources and ergonomic interfaces for task graph submission. Task-based programming is a popular approach for defining scientific workflows that can be computed on distributed clusters. However, executing task graphs on supercomputers introduces unique challenges, such as performance issues caused by the large scale of HPC workflows or cumbersome interactions with HPC allocation managers like PBS (Portable Batch System) or Slurm. This work examines what are the main challenges in this area and how do they affect task graph execution, and it proposes various approaches for alleviating these challenges, both in terms of efficiency and developer ergonomics.

This thesis provides three main contributions. Firstly, it provides a task graph simulation environment that enables prototyping and benchmarking of various task scheduling algorithms, and performs a comprehensive study of the performance of various task schedulers using this environment. Secondly, it analyzes the bottlenecks and overall performance of a state-of-the-art task runtime Dask and provides an implementation of an alternative Dask server which significantly improves its performance in HPC use-cases. And primarily, it introduces a unified meta-scheduling and resource management design for effortless execution of task graphs on heterogeneous HPC clusters that facilitates efficient usage of hardware resources. It also provides a reference implementation of this design within an HPC-tailored task runtime called HyperQueue, which is available as open-source software under the MIT (Massachusetts Institute of Technology) license at `https://github.com/it4innovations/hyperqueue`.

Keywords

distributed computing, task graphs, heterogeneous resources, high-performance computing

Acknowledgment

I would like to thank my supervisor, Jan Martinovič, for his advice. I would also like to thank Ada Böhm, Vojtěch Cima and Martin Šurkovský, who have co-authored several publications with me and supported me during my PhD studies. I am especially grateful to Ada Böhm for her mentorship and constant readiness to provide both research and technical guidance. I would like to express my gratitude to all the wonderful people that I met during my internship at the SPCL lab at ETH Zurich. Furthermore, my thanks also go to Vanessa DeRhen for proofreading this thesis.

The development of Estee was supported by several projects. It has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955648. This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID: 90140) and ACROSS (ID: MC2104) projects.

The development of HyperQueue was supported by several projects. It has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 956137. This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID: 90254) and LIGATE (ID: MC2102) projects.

Last but not least, I thank my wife Jana for her support and endless patience.

Contents

List of Abbreviations

ABFE Absolute Binding Free Energy 132

AI Artificial Intelligence 14

API Application Programming Interface 12, 20, 24, 26, 51, 55, 78–80, 82, 83, 91, 96, 113, 117–119, 122, 130–132, 158–161, 164, 182

ARC-CE Advanced Resource Connector Compute Element 134, 135

AWH Accelerated Weight Histogram 132

CADD Computer-aided Drug Design 131, 132

CI Continuous Integration 48

CLI Command-line Interface 12, 55, 117, 118, 120, 122–124, 126, 127, 129, 133, 135, 158, 159, 161, 164

CPU Central Processing Unit 10, 14, 15, 21, 26, 30, 32–34, 43, 44, 48, 49, 59, 103, 104, 107, 109–112, 115, 116, 122, 123, 125, 131–136, 138, 143, 144, 146–149, 152–154, 160, 163, 165

CSV Comma-separated Values 26, 131, 149

DAG Directed Acyclic Graph 22–24, 28, 29, 60, 119

DSL Domain-specific Language 16, 24, 25

FPGA Field-programmable Gate Array 14, 27, 49, 116, 129

GCC GNU Compiler Collection 20

GIL Global Interpreter Lock 9, 81, 83, 87, 88, 100, 142

GPFS General Parallel File System 155

GPU Graphics Processing Unit 10, 14, 15, 20, 21, 26, 27, 30, 32, 34, 43, 48, 49, 107, 109–112, 116, 122–124, 131–133, 145–147, 160, 165

HLFET Highest Level First with Estimated Times 64

HPC High-performance Computing 14–19, 21–24, 26–29, 33, 38–56, 58–60, 77–79, 88, 95, 100–103, 106, 111, 114, 117–119, 123, 126, 131–134, 157, 159–165

HTTP HyperText Transfer Protocol 130

I/O Input/Output 48, 80, 81, 83, 88, 122, 134, 155–157, 163, 164

JSON JavaScript Object Notation 91, 133, 135

MD Molecular Dynamics 131–133, 165

MPI Message Passing Interface 12, 15, 19–21, 49–51, 53, 54, 103, 104, 126, 164

MSD Minimal Scheduling Delay 9, 59, 62, 63, 67, 71–73, 76

NIC Network Interface Controller 50

NUMA Non-uniform Memory Access 14, 15, 49, 57, 110, 122–125, 143–145

OLAP Online Analytical Processing 26

OOM Out Of Memory 127

OpenMP Open Multi-processing 12, 15, 19–21, 23, 27

PBS Portable Batch System 41, 43–45, 48, 114, 115, 128, 130, 134

PEP Python Enhancement Proposal 81

PGAS Partitioned Global Address Space 19

PRNG Pseudorandom Number Generation 65, 84

RAM Random-access Memory 15, 32, 49, 51, 83, 110, 122, 123, 135

RBFE Relative Binding Free Energy 132

RDMA Remote Direct Memory Access 19, 50

RSS Resident Set Size 153

SPMD Single Program, Multiple Data 19

SQL Structured Query Language 26

TCP/IP Transmission Control Protocol/Internet Protocol 80, 81, 118

TOML Tom's Obvious Minimal Language 119, 158

TPU Tensor Processing Unit 14

YAML Yet Another Markup Language 55, 158

List of Figures

List of Tables

List of Source Code Listings

List of Definitions

Chapter 1

Introduction

HPC (High-performance Computing) infrastructures are crucial for the advancement of scientific research, as they offer unparalleled computational power that can be leveraged to perform the most complex scientific experiments. This massive performance is required (among other use-cases) in various scientific domains, such as weather forecasting [5], computational fluid dynamics [6], bioinformatics [7] or deep learning [8].

Over the last several decades, the performance of HPC clusters (*supercomputers*) has been steadily increasing, effectively doubling every few years, in line with Moore's Law and Dennard scaling [9]. However, it also became more difficult for HPC users to tap into that performance increase. Thirty years ago, it was possible to get essentially double the performance for free, just by using a new (super)computer every two years, without having to modify existing programs. This phenomenon had started to diminish by the end of the last century, as chip designers became limited by the memory wall [10] and especially the power wall [11].

To keep up with the expectations of exponential performance increases, CPUs (Central Processing Units) had to become more complex. Processor manufacturers started implementing various buffers and caches, multiple cores, simultaneous multithreading, out-of-order execution and a plethora of other techniques that would allow the CPU to run faster, without requiring massive increases of power draw or memory bandwidth. The existence of multiple cores and sockets and the need for ever-increasing memory sizes has also made the memory system more complex, with NUMA (Non-uniform Memory Access) memories becoming commonplace in HPC. To achieve even more performance, HPC clusters started massively adopting various accelerators, like the Intel Xeon Phi [12] manycore coprocessor or general-purpose NVIDIA or AMD GPUs (Graphics Processing Units), which eventually became the backbone of the majority of current supercomputers [13]. Some clusters have also adapted more unconventional accelerators, like reconfigurable hardware, such as FPGAs (Field-programmable Gate Arrays), or AI (Artificial Intelligence) accelerators, such as TPUs (Tensor Processing Units). This trend gave rise to heterogeneous clusters that offer various types of hardware, each designed for specific workloads.

These hardware improvements have managed to keep up with Moore's Law, but no longer without requiring changes to the software. The increasing complexity and heterogeneity of HPC hardware has caused the "HPC software stack" and the corresponding programming models to

become more complex, making it far from trivial to leverage the available performance offered by supercomputers. Individual computers of HPC clusters (called *computational nodes*) can consist of hundreds of CPU cores each, yet it is challenging to write programs that can scale to such high core counts. The RAM (Random-access Memory) of each node contains multiple levels of complex cache hierarchies, and it has such a large capacity that it has to be split into multiple physical locations with varying access latencies (NUMA), which requires usage of specialized programming techniques to achieve optimal performance. And the ever-present accelerators, for example GPUs, might require their users to adopt completely different programming models and frameworks.

Historically, optimized HPC software was primarily 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 (Open Multi-processing) [14], CUDA [15] or MPI (Message Passing Interface) [16]. 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. It is unreasonable to expect that most domain scientists that develop software for HPC clusters (who are often not primarily software developers) will be able to use all these technologies efficiently without making the development process slow and cumbersome. This task should be left to specialized performance engineers, enabling the scientists to focus on the problem domain [17].

With the advent of more powerful hardware, HPC systems are able to solve new problems, which are more and more demanding, both in terms of the required computational power, but also in terms of data management, network communication patterns and general software design and architecture. Areas such as weather prediction, machine-learning model training or big data analysis require executing thousands or even millions of simulations and experiments. These experiments can be very complex, consisting of multiple dependent steps, such as data ingestion, preprocessing, computation, postprocessing, visualization, etc. It is imperative for scientists to have a quick way of prototyping these applications, because their requirements change rapidly, and it would be infeasible to develop them using only very low-level technologies.

The growing complexity of HPC hardware, software and use-cases has given rise to the popularity of task-based programming models and paradigms. Task-oriented programming models allow users to focus on their problem domain and quickly prototype, while still being able to describe complicated computations with a large number of individual steps and to efficiently utilize the available computational resources. With a task-based approach, a complex computation is described using a set of atomic computational blocks (*tasks*) that are composed together in a *task graph* which captures dependencies between the individual tasks. Task graphs abstract away most of the complexity of network communication and parallelization, and they are general enough to describe a large set of programs in a practical and simple way. At the same time, they remain amenable to compiler-driven optimization and automatic parallelization, which helps to bring the performance of programs described by a task graph close to manually parallelized and distributed programs, at a fraction of the development cost for the application developer. They are also relatively portable by default, as the task graph programming model typically does not make many assumptions about the target platform; therefore, the same task graph can be executed on various

systems and clusters, if the tasks and a task graph execution tool can run on that cluster.

Combined with the fact that task-based tools often allow users to implement their workflows in very high-level languages, such as Python or various DSLs (Domain-specific Languages), it makes them an ideal tool for rapid scientific prototyping. However, this does not mean that low-level high-performance kernels are not being used anymore; a common approach is to describe the high-level communication structure of the workflow using a task graph where the individual tasks execute the low-level kernels, rather than implementing a monolithic application that performs all the network communication explicitly.

Task graphs are already commonly being used and deployed on various distributed systems [18, 19, 20], yet there are certain challenges that limit their usage ergonomics and performance efficiency when deployed specifically on HPC systems. These challenges stem from various factors, such as the interaction of task graphs with HPC allocation managers, the heterogeneity and complexity of HPC cluster hardware, or simply from the potentially enormous computational scale. When task graph authors encounter these problems, they might have to step out of the comfort zone of this easy-to-use programming model, and implement parts of their applications using other, more complicated approaches, to either meet their performance goals or to even make it possible to execute their application on HPC clusters at all. Removing or alleviating some of those challenges could lower the barrier of entry, make task graph execution better suited for various HPC use-cases and turn it into an actual first-class citizen in the world of supercomputing.

To achieve the goal of making it easier and more efficient to execute task graphs on heterogeneous supercomputers, this thesis sets out the following objectives:

1. Identify and analyze existing challenges and bottlenecks of task graph execution on HPC clusters, particularly in the areas of efficient hardware utilization and usage ergonomics, and examine how are existing tools able to deal with them.

2. Introduce a set of guidelines and approaches for overcoming these challenges that would facilitate effortless execution of task graphs on modern heterogeneous clusters. These guidelines should serve as a template for implementing HPC-optimized task graph execution tools.

3. Implement a task graph execution tool using these guidelines and evaluate it on HPC use-cases.

The thesis is structured as follows. Chapter 2 describes various approaches for designing parallelized programs on distributed clusters, to provide context on how does task-based programming relate to them. It also concretizes a specific subset of task-based programming relevant for this thesis. Chapter 3 then defines key terms related to tasks and task graphs in detail, to provide a shared vocabulary that will be used throughout this thesis. It is followed by Chapter 4, which discusses various ergonomic challenges and performance bottlenecks faced by state-of-the-art distributed task runtimes when executing task graphs on HPC systems.

The following three chapters then discuss designs for overcoming these challenges. Chapters 5 and 6 focus solely on the efficiency aspects. Chapter 5 evaluates the quality of various task scheduling algorithms, which are important for achieving good performance when executing task workflows, and introduces Estee, a task graph execution simulator that can be used to prototype new

scheduling algorithms. Chapter 6 analyzes the runtime performance of Dask, a state-of-the-art task runtime, and introduces an alternative implementation of its server called RSDS, which is able to outperform Dask in various HPC use-cases. Chapter 7 then focuses on improving both the ergonomic and performance aspects of task execution using a meta-scheduling and resource management approach designed to facilitate task graph execution on heterogeneous clusters. This approach has been implemented in the HyperQueue task runtime, which is also described and evaluated in this chapter in detail. Finally, Chapter 8 summarizes the thesis and outlines future work.

Chapter 2

Parallel and distributed computing

There are many ways to design and implement applications for a distributed cluster, a set of computers, typically labeled as (computational) nodes, that have their own independent processors and memory and are connected together with a computer network so that they can cooperate on solving difficult problems. In the context of HPC, such distributed clusters are called *supercomputers*, and one of their distinguishing features is that all the computers of the cluster reside within one physical location, and they are connected with a very high-speed and low-latency network. Even though there are other kinds of distributed clusters, such as data centers or cloud-based systems, this thesis focuses exclusively on HPC systems and supercomputers; therefore, the term (distributed) cluster will denote an HPC cluster in the rest of the text.

Distributed applications are implemented using various *programming models* that allow expressing communication patterns which enable individual cluster nodes to cooperate and exchange data, and thus efficiently utilize available computational resources. Communication between nodes is crucial, as that is what allows distributed clusters to offer unparalleled performance by distributing the computational load among multiple computers and thus achieving speedup through parallelization.

This thesis is focused on a particular subset of task-based programming models and their interaction with HPC clusters. The goal of this chapter is to specify this subset. First, it provides a broad overview of the most popular approaches for implementing distributed and parallel applications (with particular focus on HPC use-cases), and then it gradually concretizes which niches of this diverse area are most relevant to the topic of this thesis.

2.1 Parallel programming models

This section describes the most important representatives of programming models that are used in the world of supercomputing. It divides the programming models into two broad categories: models that express parallelization and network communication explicitly, and models that do so implicitly.

Explicit parallelization

One way to design distributed applications is to leverage programming models that express the parallelization of computation and the exchange of data between nodes explicitly. This has been the predominant way of creating HPC software for many years, and it is still very popular today [21, 22, 23]. Below are a few examples of these explicit approaches.

Message passing has historically been the most popular method for implementing HPC software. In this model, a distributed computation is performed by a set of processes with separate memory address spaces that are typically executed across multiple nodes. The processes cooperate together to solve complex problems by exchanging network messages (hence the term *message passing*). Message passing applications are commonly implemented using the SPMD (Single Program, Multiple Data) [24] programming model, where the implementation logic of all the processes participating in the computation is embedded within a single program.

The most popular representative of this programming model is the MPI [25] framework, which is used by a large number of existing HPC applications [22]. It defines a set of communication primitives, operators and data types, which can be used to perform computations, exchange data and synchronize progress between either two (*point-to-point communication*) or multiple (*collective communication*) processes running on remote nodes. Listing 2.1 shows a simple MPI program designed to be executed in two (potentially distributed) processes. The first process sends a number to the second process, which waits until that number is received, and then prints it. Notice how network communication and process synchronization is expressed explicitly, by calling the `MPI_Send` and `MPI_Recv` functions. We can also see the SPMD paradigm in practice, because the code for both processes is interleaved within the same program.

PGAS (Partitioned Global Address Space) [26] is a relatively similar programming model, which also often employs the SPMD paradigm. Where it differs from message passing is in the way it expresses communication between processes. Message passing processes share their memory by communicating with other processes, while PGAS provides an abstraction of a shared memory address space and allows processes to communicate through it[1]. PGAS provides an illusion of a global memory address space that is available to processes that participate in the communication, which makes it slightly less explicit in terms of expressing the communication patterns within the program, because it translates certain memory operations into network messages on behalf of the programmer.

PGAS programs also often employ *one-sided communication* techniques, such as RDMA (Remote Direct Memory Access), which allows a process to directly read or write a region of memory from the address space of a different process (potentially located at a remote node).

Shared-memory multiprocessing is an approach that focuses on the parallelization within a single computational node, by leveraging multithreading to achieve speedup. In the area of HPC, it is common to use the OpenMP [14] framework to implement multi-threaded applications. Apart

[1] To paraphrase the famous "Do not communicate by sharing memory; instead, share memory by communicating" quote that originates from the `Go` programming language community.

```c
#include <mpi.h>
#include <stdio.h>

int main() {
    MPI_Init(NULL, NULL);

    // Find out the ID of this process
    int process_id;
    MPI_Comm_rank(MPI_COMM_WORLD, &process_id);

    if (process_id == 0) {
        // Send one integer to process 1
        int value = 42;
        MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (process_id == 1) {
        // Receive one integer from process 0
        int value = 0;
        MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", value);
    }

    MPI_Finalize();

    return 0;
}
```

Listing 2.1: MPI program implemented in C

```c
void compute_parallel(int* items, int count) {
    // This loop is executed in parallel
    #pragma omp parallel for
    for (int i = 0; i < count; i++) {
        items[i] = compute(i);
    }
}
```

Listing 2.2: C program using a simple OpenMP annotation

from providing interfaces for parallelizing code, synchronizing threads through barriers or various locks and performing atomic operations, it is also able to offload computation to various accelerators (like a GPU) attached to a node. OpenMP can be used together with the two previously mentioned programming models (it is often combined especially with MPI [27]), in order to achieve parallelization both intra-node (via multithreading) and inter-node (via network communication).

OpenMP does not only offer an API (Application Programming Interface), but it can also be integrated directly within a compiler, e.g. in GCC (GNU Compiler Collection) for programs written in the C or C++ programming languages. This enables it to provide source code annotations (called *pragmas*), which allow the programmer to parallelize a region of code with very little effort. An example of this can be seen in Listing 2.2, where a loop is parallelized simply by adding a single annotation to the source code.

These explicit programming models share a lot of desirable properties. They give their users a

lot of control over the exchange of data between individual cores and remote nodes, which allows creating very performant programs. Having the option to explicitly describe how will the individual cores and nodes cooperate also enables expressing arbitrarily complex parallelization patterns and data distribution strategies. However, in order to fully exploit the performance potential of explicit parallelization, the programmer must have advanced knowledge of the CPU or GPU hardware micro-architecture [28] and the memory model of the used programming language [29].

Even though explicitly parallelized programs can be very efficient, implementing correct applications using them is notoriously difficult. Multi-threaded and distributed programs are highly concurrent, which makes it easy to introduce various programming errors, such as deadlocks, race conditions or data races. Especially for distributed programs, debugging such issues can be incredibly challenging. Furthermore, programs that leverage explicitly parallel programming models are typically implemented in languages such as `C` or `C++`, which are infamous for making it difficult to write correct, memory-safe programs without memory errors and undefined behavior [30]. Memory safety issues are even more problematic in heavily concurrent programs, which further increases the difficulty and decreases the speed of developing correct distributed programs.

Apart from correctness, using explicit communication interfaces can also lead to over-dependence on a specific structure of available computational resources. For example, the MPI paradigm typically assumes that a fixed number of processes participate in the computation and might struggle with situations where some of these processes crash, which infamously makes it challenging to implement fully fault-tolerant MPI programs [31].

Implicit parallelization

Since it can take a lot of effort to implement a correct and efficient distributed program using explicitly parallel models, it would be unreasonable to expect that all users who want to leverage HPC resources to execute their experiments will "roll up their sleeves" and spend months implementing an explicitly parallel `C++` program that uses MPI and OpenMP. In fact, with scientific experiments becoming more and more complex each year, in most cases it would not even be feasible to develop custom (explicitly parallel) code for them from scratch. Instead, high-performance parallelized primitives implemented by specialized performance engineers [17] are moving into libraries and frameworks, such as GROMACS [32, 33] or TensorFlow [34, 35], that still leverage technologies like MPI or OpenMP internally, but do not necessarily expose them to their end users.

This allows users of HPC systems and scientists to focus on their problem domain, as their responsibility shifts from implementing communication and parallelization techniques by hand to describing high-level computational workflows using *implicitly parallel* programming models that are able to automatically derive the communication and parallelization structure from the program description. With an implicitly parallel model, the emphasis moves from *how* to perform a distributed or parallelized computation (which is the essence of the explicit models) to *what* should be computed and how the individual computational steps are related to each other, which is usually the main aspect that users actually want to focus on.

The primary benefit of implicitly parallel approaches is that they make it easy to define a computation that can be automatically parallelized, without forcing the user to think about how

exactly the parallelization and network communication will be performed. Execution frameworks are then able to ingest programs implemented using these models and automatically execute them on a parallel machine or even a distributed cluster in a parallel fashion, thus making it much easier for the user to leverage available hardware resources.

Implicit models are typically easier to use than the explicit models, and they facilitate rapid prototyping of parallelized programs. On the other hand, the main disadvantage of these models is the lack of control of how exactly is parallelization performed. Therefore, programs implemented using them might unable to achieve the same performance as explicitly parallelized programs.

There are various models that support implicit parallelization, for example stencil computations [36] or automatically parallelized functional languages [37]. But by far the most popular are the many tools and models based on tasks. Since task-based programming is a core topic of this thesis, the rest of the thesis will focus exclusively on programming models that leverage tasks. In particular, the following section will categorize these models based on several properties and describe representative tools that implement this paradigm.

2.2 Task-based programming models

In recent years, it has become very popular to define scientific computations running on distributed and HPC clusters using *task-based programming models* [18, 38, 19]. These models allow their users to describe the high-level structure of their computations using *computational workflows* (also called *pipelines* or *task graphs*[2]). A computational workflow is a DAG (Directed Acyclic Graph) of *tasks*, atomic and independent computational blocks with separate inputs and outputs that can depend on one another, which can be executed in a self-contained way. Such workflows can naturally express diverse scientific experiments, which typically need to execute and combine many independent steps with dependencies between themselves (for example preprocessing data, executing simulations, performing data postprocessing and analysis, etc.). They are also very flexible and easy to use, which makes them especially popular.

Since task-based programming models are implicitly parallel, their users do not have to imperatively specify how their computation should be parallelized, or when and how data should be exchanged between remote nodes. They merely describe the individual parts of their program that can theoretically be executed in parallel (the tasks) and then pass the created task graph to a dedicated execution tool (that we will label as a *task runtime*) that executes the tasks, typically on a distributed cluster. Because the program is represented with a graph, the task runtime can effectively analyze its properties (or even optimize the structure of the graph) in an automated way, and extract the available parallelism from it without requiring the user to explicitly define how the program should be parallelized.

It is important to note that from the perspective of a task runtime, each task is opaque. The tool knows how to execute it, but it typically does not have any further knowledge of the inner structure of the task. Therefore, the only parallelization opportunities that can be extracted by the tool have to be expressed by the structure of the task graph. A task graph containing a single task is thus

[2]These three terms will be used interchangeably in this thesis.

not very useful on its own. The individual tasks can of course also be internally parallel; however, this parallelization is not provided automatically by the task runtime. Tasks often achieve internal parallelism using shared memory multiprocessing, for example using the OpenMP framework.

Since task-based programming models are quite popular, there are hundreds of different tools that leverage them. It can be challenging to understand how these tools relate to one another, because umbrella terms like *task*, *task graph*, *DAG*, *workflow* or *pipeline* can represent vastly different concepts in different contexts. For example, the term task is used for many unrelated concepts in computer science, from an execution context in the Linux kernel, through a block of code that can be executed on a separate thread by OpenMP, to a program that is a part of a complex distributed computational workflow. It is thus possible to encounter two programming models or tools that both claim to use "task-based programming", even though they might have very little in common. The rest of this section will thus categorize existing state-of-the-art tools that use task-based programming models based on several properties, to put them into a broader context; it will also gradually specify which niches of this diverse area are most relevant for the topic of this thesis.

2.2.1 Batch vs stream processing

One of the most distinguishing properties that divides distributed task processing tools into two broad categories is the approach used to trigger the execution of the workflow. *Stream processing* tools are designed to execute continuously, and react to external events that can arrive asynchronously and at irregular intervals, while typically focusing on low latency. A streaming computational workflow is executed every time a specific event arrives. The workflow can then analyze the event, and generate some output, which is then e.g. persisted in a database or sent to another stream processing system. As an example, a web application can stream its logs, which are being generated dynamically as users visit the website, to a stream processing tool that analyzes the logs and extracts information out of them in real time. Popular representatives of stream processing are for example Apache Flink [39] or Apache Kafka [40]. Streaming-based tools can also be implemented using the *dataflow* programming model [41, 42].

In contrast, *batch processing* tools are designed to perform a specific computation over a set of input data (a batch) that is fully available before the computation starts, while focusing primarily on maximal throughput. Such workflows are usually triggered manually by a user once all the data is prepared, and the workflow stops executing once it has processed all of its inputs. In certain cases, a part of the computation can be defined dynamically while the workflow is already executing. For example, iterative workflows perform a certain computation repeatedly in a loop until some condition in met. Popular representatives of batch processing task runtimes are for example Dask [43], SnakeMake [44] or SciLuigi [45].

Streaming processing is common in the area of cloud computing, and is useful especially for analyzing data that is being generated in real time. However, it is not very common in the world of supercomputing, because HPC hardware resources are typically transient, and are not designed to be available for a single user at all times, which limits their usefulness for handling real-time events that occur at unpredictable times. On the other hand, batch processing workflows are a better fit for HPC clusters, since their execution time is bounded by the size of their input, which is often

known in advance, and thus they can be more easily mapped to transient, time-limited allocations of HPC hardware resources. **This thesis focuses exclusively on batch processing**.

2.2.2 Programming model generality

Even though most workflow processing tools are designed to execute a DAG of tasks, not all of them support arbitrary task graphs. Some tools use programming models designed for specialized use-cases, which allows them to offer very high-level and easy-to-use DSLs and APIs that are designed to perform a specific set of things well, but that do not allow expressing fully general task graphs.

An example of such a constrained approach is the Bulk synchronous parallel [46] model, which models a distributed computation with a series of steps that are executed in order. Within each step, a specific computation is performed in parallel, potentially on multiple remote nodes. At the end of each step, the nodes can exchange data among themselves, and they are synchronized with a global barrier, to ensure that there are no cyclic communication patterns in the computation. Even though this model does not use fully arbitrary computational graphs, it is still possible to express many algorithms with it [47].

A popular instance of this model that has gained a lot of popularity in the area of distributed computing is MapReduce [48]. Its goal is to allow parallel processing of large amounts of data on a distributed cluster in a fault-tolerant way, while providing a simple interface for the user. It does that by structuring the computation into three high-level operations, which correspond to individual bulk synchronous parallel steps:

1. A *map* operation (provided by the user) is performed on the input data. This operation performs data transformation and filtering, associates some form of a *key* to each data item and produces a key-value tuple.
2. A *shuffle* operation (implemented by a MapReduce framework) redistributes the tuples among a set of remote nodes, based on the key of each tuple, so that tuples with the same key will end up at the same node.
3. A *reduce* operation (provided by the user) is performed on each node. The reduction typically performs some aggregation operation (such as sum) on batches of tuples, where each batch contains tuples with the same key.

This approach is an instance of the *split-apply-combine* [49] paradigm, which describes the following intuitive strategy for parallel computations on a dataset: *split* the data into multiple chunks, *apply* some transformation to each chunk in parallel (this corresponds to the *map* operation) and then *combine* the results (this corresponds to the *reduce* operation).

Listing 2.3 shows a simple Python program that computes the frequency of individual words in a body of text[3] using a popular implementation of MapReduce called Apache Spark [50]. Even though the programmer only needs to provide a few simple *map* and *reduce* operations, this short program can be executed on a distributed cluster and potentially handle large amounts of data. Note that multiple *map* and *reduce* operations can be combined in a single program, which can also be observed in this example. While this program is implemented in the Python programming

[3]This computation is commonly known as *word count*.

```python
def word_count(context):
    file = context.textFile("shakespeare.txt")
    counts = file.flatMap(lambda line: line.split(" ")) \
        .map(lambda word: (word, 1)) \
        .reduceByKey(lambda x, y: x + y)
    output = counts.collect()
    print(output)
```

Listing 2.3: MapReduce word count implemented in Python

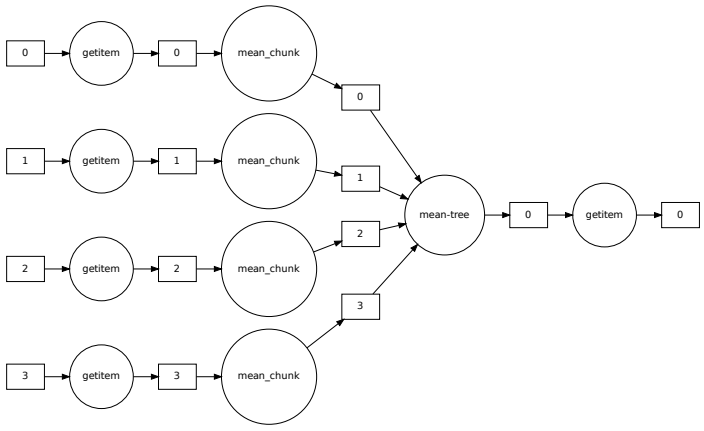
language, the most important parts of the computation are expressed using a few very specific implicitly parallel operations (*map* and *reduce*); therefore, it is possible to consider this to be a sort of DSL for expressing distributed computations.

The primary benefit of this approach is that it is easy to define a computation that can be automatically parallelized and distributed. Furthermore, the "layered" shape of task graphs produced by MapReduce programs facilitates implementation of fault tolerance, because there are clearly delimited checkpoints where the intermediate results of the computation can be persisted to disk (at the end of each *map* or *reduce* operation), so that they can be later restored in case of failure. This is only possible because the tools that execute MapReduce programs have additional knowledge and invariants about the shape of the task graph, due to its constrained nature.

However, this is also the main limitation of this model. If a computation cannot be naturally expressed using the *map* and *reduce* operations, it can be challenging, or even impossible, to implement it with the MapReduce paradigm. In particular, MapReduce assumes forward data-flow, where data is being sent through the computational pipeline in one direction. This is problematic especially for iterative computations that have to be executed repeatedly until some condition is met, which typically use the output of a previous iteration as an input for the next iteration, thus creating a loop in the flow of data.

```python
import dask.dataframe as pd

df = pd.read_csv("dataset.csv")
# This query is translated to the
# task graph shown on the right
df.groupby("Country").GDP.mean()
```



Listing 2.4: Dask DataFrame query and its corresponding task graph

Specialized types of task graphs are also commonly used to parallelize *dataframe* processing,

which is a popular approach for performing exploratory data analytics and OLAP (Online Analytical Processing) queries over two-dimensional tabular datasets (dataframes). Tools such as Dask DataFrame [43], Modin [51], Vaex [52] or CuDF [53] offer Python or SQL (Structured Query Language) interfaces for expressing queries over dataframes, and then translate these queries into task graphs that are executed on parallel machines, distributed clusters or GPU accelerators. Users of these tools do not even necessarily have to know that they are interacting with task graphs, since their usage is mostly an internal implementation detail. Listing 2.4 shows a short Python program that leverages the Dask DataFrame API for performing a query on a dataframe loaded from a CSV (Comma-separated Values) file. The query is internally translated by Dask to the task graph shown on the right side of the figure, which can then be executed e.g. on a distributed cluster.

DataFrame processing is often used in scientific workflows. However, it typically represents only a certain part of the workflow (e.g. a single task or a set of tasks), and it is typically not used to define the structure of the whole workflow.

While MapReduce, DataFrame processing or other similar models are useful for computations that have a predictable structure, they are not always general enough to express complex HPC scientific workflows. **We will thus further only consider tools and programming models that allow executing arbitrary task graphs**.

2.2.3 Task granularity

An important aspect of tasks is their *granularity*, which determines how much work the task performs and into how many subtasks it could be potentially divided to achieve more parallelization opportunities. In an extreme case, a *coarse-grained* (low granularity) task could represent the whole computation that we want to perform, while a *fine-grained* (high granularity) task could represent merely the execution of a single CPU instruction. With an increasing granularity, each task performs less work, and the whole workflow will thus have to be divided into a larger number of tasks. This makes it easier to parallelize the workflow; however, it also increases the overall overhead introduced by the task runtime, which has to manage and schedule more tasks. The same properties also apply inversely for a decreasing granularity of tasks.

Since tasks represent arbitrary computations, it is not always straightforward to determine how granular they are. Usually it is much simpler to use a proxy metric for task granularity based on the duration it takes to execute the task. Therefore, if a task takes a short time to execute (e.g. milliseconds or less), we will consider that it is highly granular, while if a task takes a long time to execute (e.g. hours or more), we will consider that it has low granularity. However, it should be noted that what constitutes high or low granularity depends on a specific use-case and the task runtime used to execute it.

Task granularity is important primarily because it determines how efficiently a task graph can be parallelized and if it even makes sense to distribute it to multiple nodes at all. For example, if certain tasks would take mere nanoseconds to execute, there would be no point in dynamically load balancing them across multiple remote nodes, because the overhead of doing so would dwarf the execution time of the tasks themselves, due to the latency induced by the network communication. In other words, it would be faster to just execute all such tasks on the local node rather than sending

```
cilk int fibonacci(int n) {
    if (n < 2) {
        return n;
    } else {
        // Spawn a new task for each recursive call
        int x = spawn fibonacci(n - 1);
        int y = spawn fibonacci(n - 2);
        // Use a barrier to wait until the tasks finish
        sync;
        return x + y;
    }
}
```

Listing 2.5: Task-parallel Fibonacci calculation using Cilk
Source code snippet adapted from [55].

them across the network. It could still be worth it to distribute a large number of such extremely granular tasks to multiple nodes, but the distribution would need to happen in an amortized way, for example only once at the beginning of the computation, to avoid excessive communication overhead.

Some task-based programming models focus primarily on high granularity tasks, for example StarPU [54], Cilk [55], HPX [56], PaRSEC [57], Legion [58], TBB [59] or OpenMP [14][4]. In these models, which are sometimes called *task-parallel* [60], tasks usually represent either functions or blocks of code that contain only a couple of instructions, and are typically relatively quick to execute (they can run e.g. for milliseconds or seconds). While some of these tools also support distributed computing, their main use-case is to provide intra-node parallelization on multicore systems with shared memory, and often also to offload computation to attached accelerators, such as GPUs or FPGAs.

They can be used e.g. to implement high-performance numerically intensive computational kernels, such as matrix multiplication or QR factorization [61], or to parallelize recursive algorithms or algorithms with irregular data access patterns. Listing 2.5 shows a program implemented in the Cilk programming language, which calculates the n-th Fibonacci number using a task-parallel approach.

Task-parallel models definitely have their place in HPC; however, they are not primarily designed to execute high-level scientific workflows, as they deal with a different set of challenges and constraints. Therefore, they are not the primary focus of this thesis and will not be considered in the rest of the text.

There is also a set of task-based tools that reside at the other end of the granularity spectrum, for example Apache Airflow [62], Dagster [63], Prefect [64] or Argo [65]. These are commonly labeled as *Workflow Management Systems*. Although it cannot be said that there is a single distinguishing feature that would separate these tools from other task runtimes, what they have in common is that they were not primarily designed for HPC use-cases. They typically work with

[4]Note that even though OpenMP has been previously presented as an example of an explicitly parallel model, it also offers a task system that provides facilities for implicit parallelization.

very coarse-grained workflows with a relatively small number of tasks, they put strong emphasis on workflow reproducibility, data provenance, execution monitoring (typically through a web interface) and scheduled execution (i.e. "execute this workflow every hour"). They are also typically being applied in cloud environments, moreso than on supercomputers. This thesis will thus not consider these tools in great detail.

Summary

This chapter has introduced various approaches for creating distributed applications designed for HPC clusters. It categorized them into two broad categories, explicitly parallel and implicitly parallel models, and demonstrated several examples of both approaches.

It has also specified the subset of distributed computing that is most relevant for this thesis. In particular, this thesis focuses on batch processing task programming models that define computations using fully general DAGs and that are designed for distributed execution on an HPC cluster, with tasks whose granularity (duration) typically spans from (tens of) milliseconds to hours.

These programming models are being commonly used for defining scientific workflows using tools that will be described in Chapter 4, which will also discuss various challenges faced by these tools on HPC systems. But before that, the following chapter will first define a vocabulary of task related terms so that we can refer to them in the rest of the thesis.

Chapter 3

Task-based programming

In order to discuss task-based programming, it is useful to provide a vocabulary of terms related to it and a set of definitions that describe the basic properties of task graphs and also the rules of their execution. These definitions necessarily have to be relatively general, so that they can be applied to a wide range of systems; even though the previous chapter has specified a relatively precise subset of task-based programming models that will be examined and analyzed in this thesis, there is still a large number of tools and systems that belong to this area of interest, each with their own distinct concepts and semantic rules. Therefore, it would be infeasible to provide a single unifying and complete task theory that would encompass details of all these tools and programming models.

This chapter defines a set of concepts related to tasks, with a particular focus on properties that are important for task graph execution in HPC environments, which will be further described in Chapter 4. The definitions described in this chapter are specifically adapted to the needs and terminology of this thesis, as is usual for other research works. They form the lowest common denominator that can be applied to the specific task execution tools and programming models discussed throughout this thesis. The presented terms are similar to definitions that can be found in existing works [66, 67, 68], although they differ in several aspects. In particular, we do not assume that task execution times nor output data sizes are known in advance, and we define a very general concept of resource management, which can be used to model complex resources of heterogeneous clusters. Some of the definitions related to resource management will be further extended in Chapter 7.

3.1 Task graphs

A computational workflow in the task-based programming model is represented with a DAG that we will label as a *task graph*. From a high-level perspective, it describes which individual computational steps should be performed, what are the constraints for where and in which order they should be computed and how should data be transferred between the individual steps of the workflow.

There are many variations of task graphs that differ in the computational properties they are able to describe. In the most basic form, task graph vertices represent computations that should be performed and the graph arcs (edges) represent dependencies between the computations, which

enforce an execution order. However, numerous other concepts can also be encoded in a task graph. For example, in addition to dependencies, the arcs could represent abstract communication channels, through which the outputs of one computation are transferred to become the inputs of another computation that depends on it. As another example, there could be a special type of arc which specifies that the outputs of a computation will be streamed to a follow-up computation, instead of being transferred in bulk only after the previous computation has finished.

As was already noted, the exact semantics of vertices and arcs of task graphs depend heavily on the specifics of tools that implement them, and it is thus infeasible to provide a single definition that would fit all variants used "in the wild". To provide a baseline definition, we will formally define a task graph that can model dependencies between tasks, the notion of transferring data outputs between tasks, and also the resources needed for the execution of individual tasks.

[**Definition 1**] A *task graph* is a tuple $G = (T, O, A, RK_t, Res_t)$, where:

- T is a set of *tasks*.
- O is a set of *data objects*.
- $A \subseteq ((T \times O) \cup (O \times T))$ is a set of arcs.
- $(T \cup O, A)$ has to form a finite directed acyclic graph.

  The absence of cycles is important; otherwise, the task graph could not be executed.
- For every data object, there has to be exactly one task that produces it:
  $\forall o \in O : |A \cap (T \times \{o\})| = 1$
- RK_t is a set of *resource kinds*. Each resource kind describes a type of resource (e.g. a CPU core or a GPU device) that might be required to execute a specific task.
- $Res_t \colon T \times RK_t \to \mathbb{N}_{\geq 0}$ is a function that defines the *resource requirement* of a task for a given resource kind.

  A resource requirement specifies what amount of the given resource kind is required to be available at a computational provider so that it can execute the given task. $Res_t(t, r) = 0$ specifies that task t does not require resource kind r for its execution.

Note that the definition above allows the existence of tasks that have no resource requirements. While there might be use-cases where this is desirable, in typical situations we might want to ensure that each task requires at least some resource (e.g. at least a single CPU core), to avoid situations where all such tasks could be scheduled to a single worker, which could lead to oversubscription. We could express that constraint using the following property: $\forall t \in T, \exists r \in RK_t : Res_t(t, r) \neq 0$.

Below, we will define several terms that will be useful for describing task graphs and their properties. For the following definitions, assume that we work with a task graph $G = (T, O, A, RK_t, Res_t)$:

- If there is an arc from a task to a data object ($(t, o) \in (T \times O)$), then we call t the *producer* of o and o the *output* of t.
- If there is an arc from a data object to a task ($(o, t) \in (O \times T)$), then we call t the *consumer* of o and o the *input* of t.

- Let us introduce a binary relation DR_G:
  $DR_G = \{(t_1, t_2) \in (T \times T) \mid t_1 \neq t_2 \wedge \exists o \in O\colon (t_1, o) \in A \wedge (o, t_2) \in A\}$
  When $(t_1, t_2) \in DR_G$, we say that t_2 *directly depends* on t_1. We can also state that t_2 consumes the output produced by t_1.
- Let us introduce a binary relation D_G, which forms the transitive closure of DR_G. More explicitly, tasks (t, t') belong to D_G if there exists a sequence $(t_1, t_2, \ldots, t_n)$ such that $t = t_1 \wedge t' = t_n$ and $\forall i \in \{1, 2, \ldots, n-1\} : (t_i, t_{i+1}) \in DR_G$.

  When $(t_1, t_2) \in D_G$, we say that t_2 *depends* on t_1 and that t_1 is a *dependency* of t_2.
- We call tasks without any inputs *source tasks*:
  $S_G = \{t \in T \mid \forall t_d \in T\colon (t_d, t) \notin D_G\}$

  It is a simple observation that unless the task graph is empty ($T = \emptyset$), there is always at least one source task in the graph, because the graph is acyclic and finite.
- We call tasks that are not depended upon by any other task *leaf tasks*:
  $L_G = \{t \in T \mid \forall t_d \in T : (t, t_d) \notin D_G\}$.

An example of a simple task graph is shown in Figure 3.1. Tasks are represented as circles, data objects as (rounded) rectangles and arcs as arrows. Task t_1 generates two data objects, which are then used as inputs for four additional tasks. The outputs of these four tasks are then aggregated by a final task t_6. This could correspond e.g. to a workflow where t_1 generates some initial data, tasks t_{2-5} perform some computation on that data and t_6 then performs a final postprocessing step and stores the results to disk.

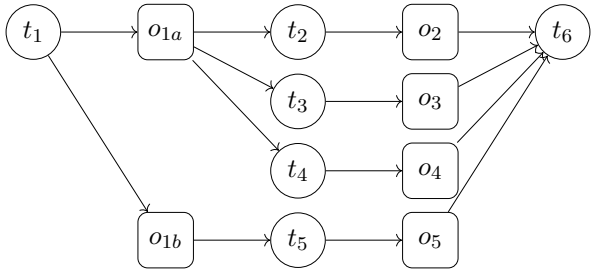


Figure 3.1: Simple task graph with six tasks and six data objects

Note that the presented definition of a task graph does not describe its semantics, i.e. how will the graph be created and executed or what will be the interactions between tasks and data objects. This depends on the specific tool that will execute the task graph. A baseline formal definition of task graph execution properties will be provided in Section 3.2.

A *task* is a serializable description of a computation that can be executed repeatedly. The serializability property is crucial, as it allow us to treat computation as data. That is a powerful concept, because it allows tasks to be sent between different nodes in a cluster, stored to disk and also to be transparently recomputed an arbitrary number of times. Enabling the recomputation of tasks is useful for achieving fault tolerance, as tasks might need to be recomputed later if some failure occurs during their execution.

In practice, a single task will typically represent either the invocation a function (an executable block of code) or the execution of a complete program. Multiple tasks in a task graph can refer to the same function or program; each such task can have different inputs. In fact, this is a common use-case, as task graphs are often used to parametrize a small set of functions or programs with many different input parameters.

Even though we have defined the inputs and outputs of tasks as sets, in practice they are usually stored using either ordered sequences or a mapping that associates a name with each input or output, because it is important to maintain a specific ordering of both inputs and outputs. For functions, the inputs are passed as arguments and the output is derived from its return value (which can potentially form a sequence of values). Therefore, we have to be able to associate each task input to a specific argument index. The same holds for tasks that execute programs. In this case, inputs can be mapped to command-line arguments and the content of the `standard input stream`, and the output can be e.g. the content of the `standard output stream` generated by the executed program or a set of files that it writes to the filesystem.

Each task can define its *resource requirements*, a set of constraints that need to be satisfied by a computational provider so that it can execute that task. As an example, a task that performs training of a machine-learning model might require a GPU to be present on the computational node where it will be executed. Other resources might include e.g. a specific number of CPU cores or a minimum amount of RAM necessary to execute a given task.

A *data object* represents a dynamically computed result of a task; its value is not known at the time of the task graph creation. Typically, it is a serialized blob of data that is eventually transferred from the node where its producer was computed to the node where its consumer should be executed. If a task programming model does not encode direct data transfers between tasks, then data objects simply serve as "empty" markers of dependencies and they do not hold any actual data. In that case, we could even remove them from the task graph completely and represent task dependencies directly with arcs between tasks.

It is important to note that not all data used by tasks has to be encoded as a data object in the task graph. As an example, tasks that represent function invocations are usually created by the execution of some program (e.g. a Python script). A task graph defined in this way is usually constructed with a specific set of input data for its *source tasks*. This data can be embedded directly within the definition of the function itself; in that case it is not represented as an explicit data object. In other words, a task might represent a serializable description of a computation along with its input data. That is why in the presented formal definition, *source tasks* do not have any explicit *inputs*; it is expected that input data is embedded directly within them.

Additionally, tasks can also read and modify the state of the environment in which they are being executed, in a way that is observable by other tasks. For example, a function can read or modify the value of a global variable, while a program can read an environment variable or create a file on a disk, without it being specified as a task output. Such actions, which are usually called side effects, are also typically not encoded within the task graph. Tasks should ideally contain as few side effects as possible, because they can make task execution non-deterministic and cause them to produce different outputs when executed multiple times, which is typically undesirable.

3.2 Task execution

Task graphs merely describe some computation; therefore, they have to be executed in order to actually produce some outputs and results. This is the responsibility of a *task runtime*, a tool that analyzes task graphs and executes them in some *computational environment*, e.g. a personal computer or a distributed cluster. Such an environment contains a set of computational providers that are able to execute tasks. We will label these providers as *workers*. A worker can execute a task by invoking its computation description (typically by calling a function or executing a program), while passing it the previously computed inputs of the task.

There are many existing task runtimes with varying architectures, features and trade-offs, which affect factors like performance, fault tolerance or expressivity of the supported variant of the task-based programming model. Several task runtimes will be discussed throughout this thesis. In the rest of this section, we will consider a case typical for HPC environments; a distributed task runtime with a central manager that communicates with a set of workers running on remote nodes that communicate together via a network. We will also define a set of baseline properties and rules that should be applicable to most task runtimes using this architecture, without necessarily going into details of execution semantics that could differ across runtimes.

In general, a task runtime oversees all aspects of task graph execution. Its two main responsibilities can be divided into managing communication with workers and handling the scheduling and execution of tasks.

Worker management involves handling the lifetime of workers, facilitating data transfers between them or providing resiliency in case of worker failures. A single worker is typically a program running on a computational node, which is connected to the runtime through a network connection. It receives commands from the task runtime, executes tasks and sends information about task execution results back to the runtime. Each worker typically manages some hardware resources that are available for tasks during their execution. Hardware resources can be assigned to workers in various ways. There can be a single worker per the whole computational node or there could be multiple workers per node, each managing a subset of the available resources (e.g. a single worker per CPU core).

The second main aspect that has to be handled by the runtime is the management of tasks. It has to keep track of which tasks have already been computed, which tasks are currently being executed on some worker(s) or which tasks are ready to be executed next. Two important responsibilities in this area are fault tolerance and scheduling.

We will define *fault tolerance* as the ability to gracefully handle various kinds of failures that can happen during task graph execution, such as task or worker failures. When the execution of a task fails with some error condition (e.g. because a worker executing the task crashes), a fault-tolerant task runtime will be able to transparently restart it by launching a new execution of that task. We will use the term *task instance* (or simply *instance*) for a specific execution of a task. Task runtimes might impose limits on retrying failed tasks, e.g. by attempting to execute up to a fixed number of task instances for each task before giving up, in order to avoid endless failure loops.

The fact that it is even possible to execute a task multiple times is one of the main advantages of the task-based programming model, where tasks declaratively describe a self-contained computation that can be re-executed arbitrarily many times. This crucial property of tasks makes fault-tolerant execution of task graphs easier to achieve than in other programming models, where individual computations are not self-contained and serializable.

Below, we provide several definitions related to task graph execution that should be applicable to most existing task runtimes. First, we will formally define a computational environment (e.g. a cluster), an environment in which a task graph can be executed.

[**Definition 2**] A *computational environment* is a tuple $C = (W, RK_w, Res_w)$, where:

- W is a set of workers (computational providers).
- RK_w is a set of *resource kinds*. Each resource kind describes some type of resource (e.g. a CPU core or a GPU device) that can be provided a worker.

  Note that in all following definitions, we will assume that the set of resource kinds of a computational environment is equal to the set of resource kinds of a task graph computed in that environment.
- $Res_w \colon W \times RK_w \to \mathbb{N}_{\geq 0}$ is a function which defines how many resources are provided by a worker for a specific resource kind.

Now we can describe the execution of a task graph. However, formally defining the behavior of such a dynamic process is much more challenging than defining the previous (relatively static) concepts, such as the task graph and the computational environment. Each task runtime has its own set of execution semantics that affect the details of how are tasks assigned to workers, how they are executed, how is data being transferred over the network, etc. Providing a formal definition of this process that would be general and could be applied to multiple task runtimes would thus be infeasible. On the other hand, it would be useful to have an option to examine if a concrete task graph execution satisfied several constraints related to the dependencies and resource requirements of tasks that most users would intuitively expect to hold.

Therefore, instead of defining the behavior of an execution itself, we assume that we have available a set of information about an already performed execution, and we will then examine if that execution has satisfied various properties. Note that in definitions related to task graph execution, the set of non-negative real numbers ($\mathbb{R}_{\geq 0}$) will be used to represent points in time. For conciseness, the term *execution* will also be used to denote a *task graph execution* in the rest of the text.

[**Definition 3**] A *task graph execution* is a tuple $E = (G, C, X)$, where:

- $G = (T, O, A, RK, Res_t)$ forms a *task graph*.
- $C = (W, RK, Res_w)$ forms a *computational environment*.
- $X \colon T \times \mathbb{R}_{\geq 0} \to W \cup \{\bot\}$ is a function that returns the worker that was currently executing a specific task at a given point in time in E, or $\bot$ if that task was not being executed at the given time.

- Each task had to be executed at least once: $\forall t \in T, \exists tp \in \mathbb{R}_{\geq 0} \colon X(t, tp) \neq \bot$
- Each task had to eventually finish its computation:
  $\forall t \in T, \exists tp \in \mathbb{R}_{\geq 0}, \forall tp' \in \mathbb{R}_{\geq 0} \colon tp' > tp \Rightarrow X(t, tp') = \bot$

Note that the definition above assumes that each task in E was being executed on at most a single worker at the same time. It could be generalized for tasks that can be executed on multiple workers at once; however, that would have to be performed in the context of a specific task runtime, as the semantics of *multi-node* execution can vary significantly between runtimes.

It is also important to note that based on the definition of the X function provided above, each task in E could have been started multiple times (in multiple *instances*), even on different workers. This captures a basic form of fault tolerance. However, we assume that each task must eventually successfully finish its execution. Additional semantics of fault-tolerant task execution are not defined, because task runtimes handle task re-execution in different ways; it would be infeasible to provide a general definition of task retry semantics.

Next, we will define three helper functions in the context of *task graph execution* $E = ((T, O, A, RK, Res_t), (W, RK, Res_w), X)$, which will be used in later definitions.

- Let $WX_E \colon W \times \mathbb{R}_{\geq 0} \to \mathcal{P}(T)$ be a function that returns the set of tasks that were currently being executed on a given worker at a given point in time in E:
  $WX_E(w, tp) = \{t \in T \mid X(t, tp) = w\}$
- Let $S_E \colon T \to \mathbb{R}_{\geq 0}$ be a function that returns the earliest point in time at which (any *instance* of) a given task started its computation in E:
  $S_E(t) = \min\limits_{tp \in \mathbb{R}_{\geq 0}} X(t, tp) \neq \bot$
- Let $F_E \colon T \to \mathbb{R}_{\geq 0}$ be a function that returns the latest point in time at which (any *instance* of) a given task finished its computation in E:
  $F_E(t) = \max\limits_{tp \in \mathbb{R}_{\geq 0}} X(t, tp) \neq \bot$

Unless a task runtime has some special semantics, then each execution should uphold the following three basic constraints, which ensure a reasonable behavior w.r.t. dependencies, task resource requirements and worker resources:

[**Definition 4**] A *dependency constraint* in the context of *task graph execution* $E = (G, C, X)$, where $G = (T, O, A, RK, Res_t)$ is a *task graph*, is defined as follows:
$\forall t_1 \in T, \forall t_2 \in T \colon (t_1, t_2) \in D_G \Rightarrow F_E(t_1) \leq S_E(t_2)$

Informally, this property states that if a task t_2 depends on a task t_1, then it cannot begin executing until t_1 has finished executing. This is a common interpretation of the dependence relation between tasks that is enforced in most task runtimes. We assume that executions performed by all task runtimes that will be further discussed in this thesis will always uphold this constraint.

[**Definition 5**] A *worker resource constraint* and a *task resource constraint* in the context of *task graph execution* $E = ((T, O, A, RK, Res_t), (W, RK, Res_w), X)$ are defined as follows:
$\forall tp \in \mathbb{R}_{\geq 0}, \forall w \in W, \forall r \in RK \colon \left(\sum_{t \in WX_E(w, tp)} Res_t(t, r)\right) \leq Res_w(w, r)$

This property both ensures that all resource requirements of all tasks are satisfied at any point in time when these tasks are being executed and also that resources of workers are not being oversubscribed.

3.3 Task scheduling

One of the most important responsibilities of a task runtime is *task scheduling*. It is the act of deciding in which order and on which specific worker(s) should each task execute, in a way that optimizes some key metric. We will use the term *scheduler* for a component of the task runtime that is responsible for assigning tasks to workers by creating some form of a *schedule*.

In general terms, a schedule is a mapping that assigns tasks to specific workers that should execute them and also assigns an order in which the tasks should be executed. However, as with task graph execution, the semantics of scheduling and the structure of schedules depend on the specific implementation of a task runtime. Schedules can be *static*, in which case they are produced just once before the task graph begins executing, or *dynamic*, where the scheduler generates the assignments on-the-fly, based on the current utilization of workers and the observed durations of tasks that have already been executed. Some schedulers also retroactively modify already produced schedules in reaction to dynamic situations that occur during task graph execution (e.g. if a new worker connects to the cluster or if some worker is underloaded), while others might not use any explicit schedules at all. Furthermore, the semantics of scheduling are tightly coupled to the semantics of task graph execution in each specific task runtime, such as fault tolerance, resource management, and other aspects. We will thus not provide a formal definition of schedules, as it would necessarily have to choose a specific schedule structure that might not be applicable to all task runtimes describes in this thesis.

What we can examine (and define) is some measure of the quality of a specific task graph execution performed by a task runtime, which is typically affected by the behavior of its scheduler. There are various metrics that a scheduler can optimize for, such as the latency to execute specific critical tasks, but the most commonly used metric is *makespan*:

[**Definition 6**] The *makespan* M_E of execution $E = ((T, O, A, RK, Res_t), C, X)$ is defined as follows: $M_E = \max_{t \in T}(F_E(t)) - \min_{t \in T}(S_E(t))$

Informally, makespan is the duration between the time when the earliest task starts to be executed to the completion of all tasks.

Task scheduling is so crucial because it has a profound effect on the efficiency of the whole workflow execution. We can observe that in Figure 3.2, which shows two executions of a simple task graph that demonstrate how a trivial change in the used schedule can severely affect the resulting makespan. The figure contains a task graph with four tasks and two data objects. The size of the circles is proportional to the execution duration of the tasks and the size of the rounded rectangles is proportional to the size of the data objects.

Let us assume that we want to execute this task graph in a computational environment with three workers (w_1, w_2, w_3). Two different executions using different schedules are shown in the

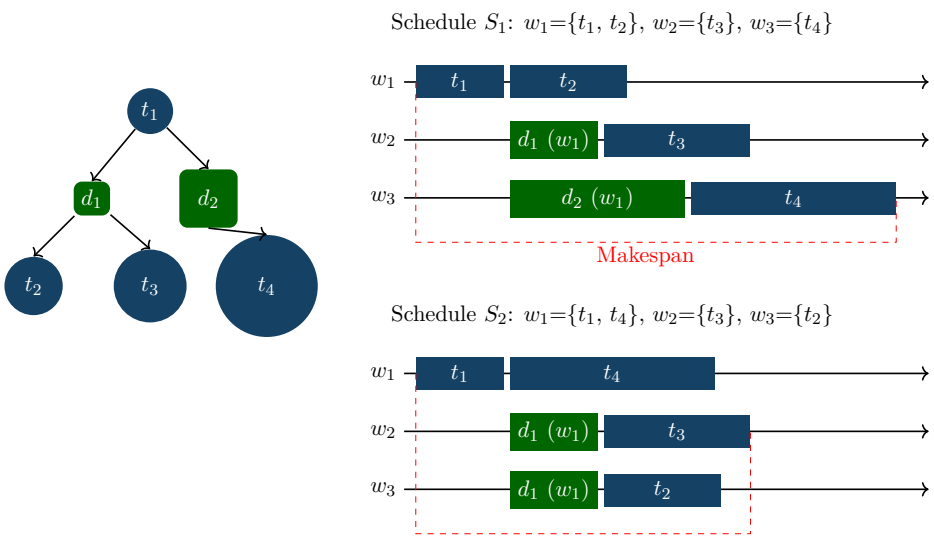


Figure 3.2: Task graph executed with two different schedules

figure. Schedule S_1 assigns tasks t_1 and t_2 to worker w_1, task t_3 to worker w_2 and task t_4 to worker w_3, while schedule S_2 assigns tasks t_1 and t_4 to worker w_1, task t_3 to worker w_2 and task t_2 to worker w_3. The two timelines show the execution of tasks (blue rectangles) and the network transfers of data objects between workers (green rectangles) for each individual worker. It is clear that with S_2, the task graph will be computed faster than with S_1, even though the only difference between the two schedules is that the tasks t_2 and t_4 were swapped between workers w_1 and w_3. Note that the timeline assumes that a worker can overlap the computation of a task with the transfer a data object to another worker over the network, which is commonly supported by existing task runtimes.

Optimal scheduling of tasks to workers is an NP-hard [69] problem even for the most basic scenarios, when the exact execution duration of each task is known, and even if we do not consider the duration of transferring data between workers over a network. Task runtimes thus resort to various heuristics tailored to their users' needs. Some classic task scheduling heuristics and their comparisons can be found in [1, 70, 71, 67, 68]. Chapter 5 provides a comprehensive survey of various task scheduling algorithms.

Scheduling heuristics have to take many factors into consideration when deciding on which worker should a task be executed:

Resource requirements The scheduler should respect all resource requirements specified by tasks. The runtime thus has to observe the dynamically changing available resources of each worker and schedule tasks accordingly, to uphold their requirements. This can be challenging especially in the presence of complex resource requirements.

Data transfer cost If the runtime operates within 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 more 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 generating the schedule itself also cannot be underestimated. As was already stated, computing an optimal solution quickly 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.

Summary

This chapter has provided a general definition of the most important terms related to task-based programming models that will be used throughout this chapter. It has introduced the notion of task graphs, tasks, data objects, resource requirements, workers, task runtimes and task scheduling.

The following chapter will focus on describing what challenges are faced by users and task runtimes when they execute task graphs on HPC clusters.

Chapter 4

State of the Art

In order to design approaches for seamless execution of task graphs on supercomputers, it is first necessary to examine the current limitations and issues faced by users that want to use task graphs in an HPC setting. This chapter describes several challenges in this area, which belong to the two broad categories that form the main focus of this thesis, namely efficiency (how to achieve high hardware utilization and make task graph execution scalable) and ergonomics (how to make it easy for users to define and execute task graphs). Most of these challenges stem from the sheer scale of HPC workloads and clusters and from the complexity and idiosyncrasies of supercomputers, which have historically been designed for different kinds of programming models.

Execution ergonomics (in the context of task-based programming models) is a broad area that encompasses several aspects, such as providing an easy way to define the structure of the task graph, allowing its execution in an effortless manner (both on a local computer and a distributed cluster), handling fault tolerance without the user's intervention, allowing users to express complex resource requirements and many others. In particular, we will focus on identifying ergonomic challenges that form a barrier for executing task graphs on HPC clusters and that limit efficient usage of heterogeneous hardware resources.

We will also discuss how are existing task runtimes able to deal with these challenges and what approaches do they use. This is important, because in order to achieve a seamless task graph execution experience on supercomputers, one should use a task runtime that takes these HPC peculiarities in mind; using off-the-shelf tools that were not designed for HPC use-cases might be challenging.

There is a large body of tools designed for executing batch-oriented task graphs on diverse computing platforms, ranging from consumer-grade laptops, through cloud deployments, to distributed and HPC clusters. They are known under various terms, such as task executors, job managers, distributed job schedulers, dataflow engines or orchestrators. We will use the term *task runtime* for all such task execution tools in this thesis, as has already been discussed in the previous chapters. These runtimes are quite popular, and they are being used for computing all kinds of scientific workflows on HPC clusters [72, 73, 74, 18].

Examples of such task runtimes include e.g. Dask [43], Parsl [75], Ray [76], PyCOMPSs [77], HyperLoom [78], GNU parallel [79], SnakeMake [44], Merlin [80], AutoSubmit [81] or

FireWorks [82]. Each task runtime defines its own instance of a task-based programming model, and has a different set of trade-offs in areas such as performance and scalability, fault tolerance, ease-of-use, ease-of-deployment and others.

Various representatives of task runtimes will be described in this chapter in the context of the individual task graph execution challenges. Chapter 6 will then provide a detailed analysis of Dask, a state-of-the-art task runtime, and Section 7.6 will compare several existing task runtimes with HyperQueue, an HPC-optimized task runtime that will be the main topic of Chapter 7.

Below you can find a list of the most important identified challenges; they will be described in detail in the rest of this chapter.

Allocation manager Any computation that is to be executed on an HPC cluster shared by many users typically has to go through a queue managed by an *allocation manager*, which provides access to the cluster through transient hardware allocations. This complicates the execution of task graphs due to the need to assign tasks to individual allocations in order to maximize hardware utilization and overcome various limits. Ideally, task runtimes should offer a way to automate this process.

Cluster heterogeneity Modern HPC clusters are very heterogeneous and might contain various kinds of accelerator devices. It is important for a task runtime to offer complex support for arbitrary resource requirements in order to take advantage of complex HPC hardware and achieve high utilization of hardware.

Performance and scalability The large scale of HPC computations and the vast amount of available hardware resources introduce unique performance challenges. Task runtimes should have minimal overhead in order to efficiently execute even very large task graphs.

Fault tolerance The scale of HPC task graphs and clusters and the transient nature of allocations makes failures an ordinary occurrence, rather than a rare edge case. This necessitates special considerations for the design of task runtimes.

Multi-node tasks Many HPC applications are designed to run on multiple nodes in parallel. This is an uncommon requirement for most task-based programming models; it requires special support to make this use-case a first-class concept.

Deployment Supercomputers typically provide a severely locked-down user environment, where it can be quite challenging to build and deploy software that requires non-trivial build or runtime dependencies. Task runtimes should thus be trivial to deploy in order to facilitate their usage on HPC clusters.

Programming model HPC task graphs can be very diverse, and range from structurally simple task graphs that contain a large number of tasks to very heterogeneous workflows with complex dependencies. Task runtimes should ideally provide a programming model that is able to support diverse use-cases in an ergonomic way.

4.1 Allocation manager

Users of HPC clusters are not typically allowed to directly perform arbitrary computations on the computational nodes (machines designed to perform expensive computations) of the cluster. Instead, they connect to machines that are usually called *login nodes*, from which they have to enqueue their desired computation into a queue handled by a submission system that manages the hardware resources of the cluster. We will use the term *allocation manager* for these submission systems and the term *allocation*[1] for a computational request submitted by a user into these managers.

Allocation managers are required for providing fair access to the resources of the cluster, because HPC clusters are typically used by many people at the same time. Without a centralized management, hardware resources could be inadvertently shared among multiple users at once, which could have undesirable performance and security implications, and could lead to oversubscription. Furthermore, usage of these clusters is usually metered. Users can typically only use a certain amount of resources assigned to their *computational project*, and when their resources run out, they have to ask (or pay) for more resources. Allocation managers thus also implement user and project accounting, so that there is a clear historical record of how many resources were consumed by individual users of the cluster.

The majority of HPC clusters [83] use one of the two most popular allocation manager implementations, either Slurm [84] or PBS (Portable Batch System) [85][2]. For simplicity, Slurm will be used as a default representative of allocation managers in the rest of this thesis (unless otherwise noted), since it shares essentially all the important described constraints with PBS.

The following process describes how computations are typically executed on HPC clusters that use an allocation manager:

1. The user enqueues a computational request (allocation) into the manager from a login node. The request typically has to specify at least how many nodes should be allocated and the maximum duration of the computation (usually labeled as *wall-time*), after which the computation will be forcibly stopped by the manager. It can also contain additional configuration, such as what kinds of nodes should be allocated or what is the priority of the request.
2. The allocation manager puts the request into a queue and schedules it to be executed at some time in the future. Since users submit their allocations into the manager continuously, each allocation has different properties and priorities, and it is not possible to precisely predict for how long an allocation will run, the schedule can be very dynamic and unpredictable, so users might have to wait seconds, minutes, hours or even days before their allocation starts to execute.
3. Once the allocation gets to the front of the queue and there are enough resources available, the manager provisions the requested amount of hardware resources (typically a number of whole computational nodes) and either executes a script (a *batch* allocation) or provides the user with an interactive terminal session on one of the allocated nodes (an *interactive* allocation).

[1] The term *job* is also commonly used for the concept of HPC computational requests.
[2] Or some of its many derivatives, such as TORQUE [86] or OpenPBS [87]

Allocations are often configured in a way that provides their authors with exclusive access to the allocated hardware resources, in which case no other user will be able to use these resources (nodes) until the allocation finishes.

4. Once the executed script finishes (or the wall-time duration is reached), the allocation ends, and its hardware resources are released so that they can be used by another allocation.

Although it might not be obvious from the above description at first, the presence of an allocation manager presents perhaps the largest obstacle for ergonomic execution of task graphs on HPC clusters. Instead of executing their task graphs directly on the cluster, users first have to think about how to split their task graph into separate allocations and manage their submission and execution. While this is true of any computation executed on HPC clusters in general, in the case of task graphs it is especially difficult due to their complex structure, and also because the concept of allocations was historically created with different programming models in mind. Mapping task workflows to HPC allocations can thus be non-trivial [88, 89].

To execute a task graph using an allocation manager, it is desirable to find a way to map tasks to allocations in a way that efficiently utilizes HPC resources and is as transparent as possible for the users. There are several approaches that can be used for performing this task-to-allocation assignment. In order to understand the various trade-offs and constraints being involved (which are mostly caused by various performance bottlenecks and limitations of existing allocation managers), we will first describe three straightforward methods for mapping tasks to allocations, which can be used even without any special support from a task runtime. After that, we will examine more automated approaches taken by existing task runtimes and other research works.

Single allocation per task graph

At first, it might seem that executing a task graph within a single allocation is a simple solution, since all the user has to do is submit an allocation that will eventually execute the complete task graph (using some task runtime), which is a similar approach that would be used for executing the task graph e.g. on a personal computer.

And it is indeed simple – when it is possible at all. The problem is that allocation managers tend to place fairly strict limits on the maximum possible execution duration of an allocation (the wall-time) and also on the number of resources (nodes) that can be requested at once, to ensure a fairer assignment of resources to users of a cluster. Therefore, when a task graph has too many tasks, takes too long to execute, or the user wants to leverage more computational resources than can fit in a single allocation, this approach will not work.

In fact, if the task graph computation is short enough that it can fit within a single allocation, it might not always even make sense to use an HPC cluster to compute it. A more realistic scenario is that even if an individual task graph can be executed quickly, users might want to execute many such task graphs (for example to execute many experiments with different parametrizations). This situation can be seen as a special case of a large task graph that consists of many disjoint components (smaller task subgraphs). In this case, it will again typically not be possible to execute all such task graphs inside a single allocation.

Even when a task graph can be reasonably executed within a single allocation, this approach might lead to hardware underutilization and resource waste [90]. Consider a typical data analysis or a machine learning workflow that works in several stages. First, it loads a dataset from disk and preprocesses it, then it trains a machine-learning model, and finally it performs some postprocessing step, which e.g. analyzes the resulting model. The preprocessing and postprocessing steps are usually not very resource intensive and run on CPUs only, while the training step typically consumes a lot of resources and is usually executed on a GPU accelerator. To execute such a workflow in a single allocation, we will need to ask for a set of resources that contain such an accelerator. The issue is that all resources of an allocation are reserved for its whole duration; therefore, we will have to pay the price for the expensive accelerated nodes (and prevent others from using them) even during the execution of workflow steps in which these nodes will be underutilized or completely idle. This is caused by the fact that all allocated resources are tied to the lifetime of the whole allocation, and using this approach we have to ask for a set of resources that form a union of the resource requirements of all tasks of the workflow.

One additional disadvantage of this approach is queuing latency. Large allocations that require a long wall-time duration or request many computational nodes typically spend a much longer time in the allocation manager queue, because it is more difficult for the manager to make sure that all the requested resources are available at the same time. As a concrete example, ten allocations that require a one hour wall-time can potentially be allocated much quicker than a single allocation that requests a ten hour wall-time, because these allocations do not need to run at the same time, and are thus easier for the manager to schedule. Similar behavior can be observed w.r.t. the number of requested nodes.

This approach can be used with essentially all existing task runtimes, since it does not require any special support from the runtime. The user simply submits a single allocation, and when it starts, the whole task graph is computed with a task runtime all in one go. This method is thus relatively simple for the user, although as has been mentioned, it might not be feasible for many use-cases, and it can lead to resource waste and a long latency before receiving the first results of finished tasks.

Separate allocation for each task

The previous approach mapped all tasks to a single allocation. An opposite extreme would be to map each task to a separate allocation. This approach might seem intuitive, because from a certain point of view, HPC allocation managers can also be viewed as task runtimes, if we consider allocations to be tasks; therefore, it might be tempting to treat them as such. Both PBS and Slurm even support a crude notion of dependencies between allocations, which could allow users to express computational graphs. Indeed, in an ideal world, there would be no difference between an allocation manager and a task runtime, and users could simply construct an arbitrarily granular task graph and execute it simply by submitting it directly into the allocation manager.

However, in practice, this approach is mostly infeasible, at least with the currently popular allocation managers (PBS and Slurm), because they introduce a non-trivial amount of overhead per each allocation [91]. In a best case scenario, Slurm is able to launch a few hundred allocations

per second [92]. However, more realistically, users on a crowded cluster might experience at least a few hundred milliseconds overhead per each allocation, if not more, which is order(s) of magnitude more than the overhead of an efficient task runtime [2].

Even though there is still probably room for reducing the overhead of contemporary HPC allocation managers, it is important to note that some of the performance limitations are inherent. Allocation managers have to (among other things) provide accurate accounting, handle robust and secure provisioning and cleanup of hardware resources provided to allocations, manage user and process isolation on computational nodes and ensure user fairness. These responsibilities are out of scope for most task runtimes, and thus it is not surprising that they can usually achieve much higher performance. While in theory, it could be possible to design an allocation manager that can also act as a (performant) task runtime at the same time (some efforts have been made on this front e.g. by Flux [93], which will be described later below), it is probably infeasible to modify the two most prominent allocation managers that are used almost ubiquitously in the current HPC ecosystem to support this use-case.

Due to this overhead, allocation managers usually limit the number of allocations that can be enqueued by a single user at any given time (e.g. to a few hundred or a thousand), to ensure that the manager is not overloaded and that it can continue to serve requests from other users of the cluster. Therefore, unless the task graph is relatively small, it will most likely be infeasible to create a separate allocation for each task.

It should be noted that both Slurm and PBS allow partially overcoming this limitation through *allocation arrays* (also labeled as job arrays). This concept allows users to submit a large amount of computations with the same shape and properties in a single allocation. However, it still has several disadvantages. One is a lack of flexibility; it does not easily allow submitting heterogeneous tasks with different resource requirements, and it also has only a very crude support for dependencies between the submitted tasks. Another disadvantage is fault tolerance; if some of the tasks of the array fail, users have to manually identify and resubmit them in another allocation, which is far from effortless. Furthermore, clusters impose limits for allocation arrays as well, and thus even the array approach might not be enough to encompass all tasks of massive task graphs.

Apart from the maximum allocation count, there are other limitations that can be imposed by allocation managers. Some tasks of scientific workflows can be granular, and require only few resources, e.g. a single CPU core. To avoid wasting resources, an allocation mapped to such a task should thus ask only for the exact set of resources needed by it. However, allocation managers are sometimes configured in a way that only offers node-level granularity of hardware resources, and thus does not allow users to request less than a whole computational node [94]. In these cases, it would be wasteful if we had to request a whole computational node e.g. for a simple task that runs for just a couple of seconds and requires only a couple of cores.

There are two primary reasons why allocation managers limit the granularity of allocations. First, reducing the granularity of allocations lessens the overall overhead. If users are able to ask e.g. for individual cores of each node, the manager would have to manage many more allocations than when users ask for complete nodes, which could quickly become unmanageable. The second reason is security. If the manager provides the ability to reserve only a fraction of a node, allocations

from multiple users can execute on the same node at the same time. This reduces isolation between users, which might potentially have security implications. It can also affect performance, since concurrently running allocations might compete for some shared resource on the same node (e.g. the operating system process scheduler). Since each node runs a single instance of an operating system, it forms a natural isolation domain, and it is thus also frequently used as a unit of granularity for allocations.

Another reason why using an allocation manager directly as a task runtime might be impractical is that it makes debugging and prototyping task graphs more difficult. It is useful to have the ability to examine the execution of a workflow locally, e.g. on the user's personal computer, before running it on a cluster. However, if the task graph were implemented directly in an allocation manager, users would have to deploy tools like PBS or Slurm locally in order to debug their workflows, which could be challenging.

To summarize, using an allocation manager directly as a task runtime is currently mostly impractical, primarily because of the overhead associated with each allocation and the resulting difference in task and allocation granularity, which can lead to resource waste. Therefore, users who want to execute a task graph on an HPC system usually use a separate task runtime rather than defining task graphs using the allocation manager directly.

Task graph partitioning

Both of the mentioned extreme approaches have severe disadvantages. A compromise between them is to partition the task graph into smaller subgraphs and submit each subgraph as a separate allocation. This approach allows mapping a large number of tasks into a smaller number of allocations, and thus amortize the allocation overhead. Most users will probably sooner or later converge to a similar approach, once their task graph becomes sufficiently large and complex, and they will have to reconcile the coarse-grained nature of allocations with the fine-grained nature of tasks.

This process is far from straightforward if it is performed manually, i.e. if the user has to manually split the task graph before submitting the allocations, and then start an independent instance of a task runtime inside each allocation. Not just because an optimal graph partitioning is itself a notoriously difficult NP-hard [95] problem in general, but also because it requires implementation efforts outside the boundaries of the used task-based programming model. In other words, users might need to reimplement some parts of a task runtime due to the fact that a part of the task graph execution happens inside allocations and a part happens outside allocations.

For example, the intermediate outputs of computed tasks of a partitioned subgraph might have to be persisted to a filesystem before the corresponding allocation ends, and the results from multiple allocations then have to be merged together to produce all the necessary data outputs of the complete task graph. Another issue is that if some tasks fail and their corresponding allocation ends, the task runtime does not get a chance to recompute them. Users will thus need to identify such failed tasks and then selectively resubmit them into new allocations.

Meta-scheduling

While the task graph partitioning approach is a general solution to the problem of running task graphs using allocation managers, it is fairly cumbersome when it has to be performed by users manually. There are various task runtimes that can help to (either partially or fully) automate this process. We will use the term *meta-schedulers* for these task runtimes in this thesis. These tools typically operate on a level above the allocation manager, instead of only running within the context of a single allocation. They are thus able to manage task graphs that span multiple allocations.

Existing task runtimes use various approaches for providing some form of meta-scheduling. Tools such as GNU PARALLEL [79] can automatically map each task to a separate allocation. While this can be useful for running simple task graphs on supercomputers, it does not help overcoming allocation manager submission limits. HYPERSHELL [96] improves upon this by enabling users to specify a *task bundle size* parameter, which batches the configured number of tasks together in the same allocation. While this method can help amortize the cost of allocations for large task graphs, it is mostly only useful for task graphs without dependencies, for which it is possible to express the partitioning using a single number. It can be used by HYPERSHELL since it does not support dependencies between tasks.

SNAKEMAKE [44] allows its users to explicitly assign each task to a *group*. Tasks with the same group are then computed in the same allocation. While this approach is more flexible than using a single batching parameter, it requires users to perform the partitioning manually, which is quite challenging to do well in order to achieve good node utilization. Without using groups, SNAKEMAKE executes each task in a separate allocation, which suffers from the already mentioned disadvantages, and is even actively discouraged on some HPC clusters [97].

An improved partitioning process has been described in [98]. It partitions task graphs based on their structure, by splitting them into levels that contain subgraphs of tasks that are somehow related. A similar approach is used by task runtimes such as PEGASUS [18] or AUTOSUBMIT [81]. For example, a "horizontal level" partitioning groups tasks that do not have dependencies between each other and that can run in parallel, while a "vertical level" partitioning groups tasks that form a dependency chain. While this approach makes partitioning easier for users, they still need to decide which partitioning mode should be used for specific parts of the graph. Furthermore, a significant disadvantage is that the partitioning is performed statically, before the task graph is executed. This can lead to suboptimal hardware utilization, because it is not possible to load balance tasks across different allocations. It can also lead to resource waste, because allocations created to execute a group of tasks will typically be configured so that they ask for a sum of all the resource requirements of tasks within that group.

E-HPC [90] employs a similar approach, where it splits individual stages of workflows into separate allocations that contain the ideal amount of resources for that stage. To tackle the issue of allocations that reserve too many hardware resources, it allows these stages to dynamically modify their resource requirements through the use of checkpointing. When a stage dynamically asks for a different set of resources, it is checkpointed, moved to a different allocation and then restarted. While this approach does alleviate some problems with resource waste, it is relatively

inflexible; workflows have to be manually separated into individual stages with coarse-grained resource requirements, and these stages are still submitted as individual allocations rather than being load balanced across different allocations.

To achieve a better utilization of computational nodes, some task runtimes allow users to deploy a generic computational provider (a worker) in an allocation, which is then able to execute any task (based on its resource requirements), instead of directly submitting tasks into allocations. This enables the task runtime to dynamically assign tasks to workers even across different allocations, which helps improve the achieved hardware utilization. It also makes the partitioning process much easier, because users simply have to spawn a set of workers and submit tasks, and the task runtime then takes care of the rest.

Task runtimes such as Merlin [80] or Parsl [75] support this dynamic meta-scheduling approach, although they require users to preconfigure each worker for a specific subset of resource requirements. For example, a worker might be configured for executing tasks that require a specific number of cores, or tasks that require to be executed on multiple nodes etc. This can be limiting for very heterogeneous task graphs or clusters; the task scheduler cannot make full use of all available resources because it has to adhere to worker configurations that were predetermined by the user.

Other task runtimes, such as Balsam [99] or FireWorks [82], support a more flexible approach, where each worker can execute any task whose resource requirement can be fulfilled by the resources assigned to that worker. This enables fully dynamic load balancing across allocations and all available hardware resources.

An additional important feature that simplifies submission of workers is some form of an automatic allocation system. As an example, Balsam or Dask [43] with the Dask JobQueue [100] plugin are able to automatically submit allocations on behalf of the user in response to computational load. These allocations then start a generic worker which can immediately start executing tasks. This removes an additional manual step that would otherwise have to be performed by users.

Alternatives

Apart from meta-scheduling, there are also alternative approaches to resolving limits of allocation managers. Some works examine what it would take to make existing allocation managers more workflow friendly. In [89], a modification to the Slurm allocation manager is proposed, which makes it workflow-aware and adds support for fine-grained resource management to it. This approach splits tasks of a workflow submitted within a single allocation into a larger number of allocations, and assigns them corresponding priorities and resources based on their requirements. While this improves certain scheduling aspects of Slurm, it still creates a separate allocation for each task, which does not remove the overhead associated with allocations. This method also assumes prior knowledge of the workflow structure; therefore, it is not possible to easily dynamically add tasks or computational resources while the workflow is being executed.

Another approach is to create a new kind of an allocation manager designed with tasks and workflows in mind. An example could be the Flux [93] HPC scheduler, which serves as a re-imagination of a modern allocation manager. It treats the whole cluster as a unified pool of resources and allows submitting computations that make use of these resources with high granularity, down to the

level of individual cores. Flux also provides dynamic management of non-computational resources such as storage and power, and even takes them into account during scheduling, for example by avoiding I/O (Input/Output) intensive computations when not enough I/O bandwidth is currently available. Furthermore, it enables defining generic resources, unlike traditional allocation managers that can only manage a small set of known resources, such as CPU cores or memory. Providing this kind of flexibility and granularity puts a lot of pressure on the system. Flux manages it by using a distributed and hierarchical scheduling design, where each allocation can act as a separate Flux instance that can then recursively subdivide its resources into additional nested allocations, and even dynamically ask for reducing or increasing its set of resources.

While modifying allocation managers to remove some of their workflow handling issues or creating new allocation managers are viable approaches, it should be noted that Slurm and PBS are currently dominating the most powerful supercomputers [83], and that replacing (or even heavily modifying) the allocation manager of an HPC cluster is a very difficult process, which requires a lot of implementation work, configuration tuning, documentation rewriting and also retraining the cluster administrators, and more importantly its users. Furthermore, using the allocation manager directly as a task runtime has a disadvantage that was already mentioned; it might not be so easy to run the workflows on personal computers or CI (Continuous Integration) servers, which limits the ability to prototype and test them.

A different approach that is designed to make it easier to use HPC clusters is taken by tools and frameworks such as CUMULUS [101], OPEN ONDEMAND [102], HEAPPE [103] or LEXIS [104]. These aim to simplify usage of clusters, and thus also remove the need for users to interact with allocations manually, by providing high-level web interfaces for managing computations. They focus on user authorization and authentication, data management, provisioning of resources across different clusters and also partially on (coarse-grained) workflow execution. This approach can make it easier for users that do not have experience with the Linux operating system or interacting with the terminal to leverage HPC resources. However, it does not resolve the limitations of allocation managers by itself, it simply moves the problem to a different place, and forces the high-level computational framework to deal with it. They can do that by leveraging task runtimes which are able to perform automatic meta-scheduling. For example, HEAPPE provides support for integration with the HYPERQUEUE [105] task runtime, which will be described in Chapter 7.

4.2 Cluster heterogeneity

Even though task graphs are designed to be portable and ideally should not depend on any specific computational environment, for certain types of tasks, we need to be able to describe at least some generic computational environment constraints. For example, when a task executes a program that leverages the CUDA programming framework [15], which is designed to be executed on an NVIDIA graphics accelerator, it has to be executed on a node that has such a GPU available, otherwise it will not work properly.

It should thus be possible for a task to define *resource requirements*, which specify resources that have to be provided by an environment that will execute such a task. For example, a requirement

could be the number of cores (some tasks can use only a single core, some can be multi-threaded), the amount of available main memory, a minimum duration required to execute the task or (either optional or required) presence of an accelerator like a GPU or an FPGA. In order to remain portable and independent of a specific computational environment, these requirements should be abstract and describe general, rather than specific, kinds of resources.

The challenge related to resource requirements of HPC tasks specifically is the diverse kinds of hardware present in modern HPC clusters, which have started to become increasingly heterogeneous in recent years. This trend can be clearly seen in the TOP500 list of the most powerful supercomputers [106]. Individual cluster nodes contain varying amounts and types of cores and sockets, main memory, NUMA nodes or accelerators like GPUs or FPGAs. Since HPC software is often designed to leverage these modern HPC hardware features, this complexity is also propagated to tasks and their resource requirements.

Existing task runtimes have various levels of support for resource requirements. Most task runtimes allow configuring at least a known set of resources per task, most often the number of CPU cores, amount of RAM memory or the number of GPU accelerators. This is enough for simple use-cases, but not for heterogeneous clusters with e.g. FPGA devices or multiple kinds of GPUs from different vendors. Some tools, such as Dask or SnakeMake, allow users to define their own resource kinds, which can be used to describe completely arbitrary kinds of resources. However, they do not support more complex use-cases, such as specifying that a task requires only a fraction of a resource, that some resources can have relations between them (for example cores residing in the same NUMA node) or that a task could support multiple sets of resource requirements.

A resource requirement that is fairly specific to HPC systems is the usage of multiple nodes per single task. This requirement is necessary for programs that are designed to be executed in a distributed fashion, such as programs using MPI, which are quite common in the HPC world. The use-case of tasks using multiple nodes is discussed in more detail later in Section 4.5.

4.3 Performance and scalability

The massive scale of HPC hardware (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.

Scheduling Task scheduling is one of the most important responsibilities of a task runtime, and with large task graphs, it can become a serious performance bottleneck. Existing task runtimes, such as Dask, can have problems with keeping up with the scale of HPC task graphs. The scheduling performance of various task scheduling algorithms will be examined in detail in Chapter 5.

Runtime overhead Using a task runtime to execute a task graph will necessarily introduce some amount of overhead, caused by scheduling, network communication and other actions performed by the task runtime. Task runtimes should be mindful of the overhead they add, because it can

severely affect the duration needed to execute a task graph. For example, even with an overhead of just 1 ms per task, executing a task graph with a million tasks would result in total accumulated overhead of more than fifteen minutes.

Many popular task runtimes, e.g. Dask, Parsl, Balsam, AutoSubmit or SnakeMake are implemented in the Python programming language. While Python provides a lot of benefits in terms of an ergonomic interface for defining task graphs, using it for the actual implementation of scheduling, network communication and other performance-critical parts of the task runtime can lead to severe bottlenecks, as Python programs are infamously difficult to optimize well.

This aspect will be examined in detail in Chapter 6, where we study the performance and overhead of Dask, a state-of-the-art task runtime implemented of Python.

Architecture A typical architecture of a task runtime consists of a centralized component that handles task scheduling, and a set of workers that connect to it over the network and receive task assignments. Even with a central server, the task runtime can achieve good performance, as we will show in Chapters 6 and 7. However, the performance of the server cannot be increased endlessly, and at some point, a centralized architecture will become a bottleneck. Even if the workers exchange data outputs directly between themselves, the central component might become overloaded simply by coordinating a vast number of 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 [76]. 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 location, the bottleneck will most likely remain even in an otherwise fully decentralized system.

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, could provide a non-trivial performance boost in this regard. Some existing runtimes, such as Dask, can make use of such technologies [107].

As we have demonstrated in [108, 109], in-network computing 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 in task runtimes, for example to reduce the latency of management messages between the scheduler and workers or to increase the bandwidth of large data exchanges among workers, by moving these operations directly onto the network card. This line of research is not pursued in this thesis, although it could serve as a future research direction.

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 of computational nodes, e.g. on login nodes of computing clusters or on client devices (e.g. laptops), whose

performance can be limited. It can take a lot of time to build, serialize and transfer such graphs over the network to the task runtime that runs on powerful computational nodes. This can create a bottleneck even before any task is executed; it has been identified as an issue in existing task runtimes [110].

In such case, it can be beneficial to provide an API for defining task graphs in a symbolic way that could represent a potentially large group of similar tasks with a compressed representation to reduce the amount of consumed memory. For example, if we want to express a thousand tasks that all share the same configuration, and differ e.g. only in an input file that they work with, we could represent this as a group of thousand similar tasks, rather than storing a thousand individual task instances in memory. This could be seen as an example of the classical Flyweight design pattern [111].

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. Dask is an example of a task runtime that supports such symbolic task graph representations.

Data transfers After a task is computed, it can produce various outputs, such as standard error or output streams, files created on a filesystem or data objects that are then passed as inputs to dependent tasks. Managing these data streams can be a bottleneck if the number of tasks is large.

Some task runtimes, such as SnakeMake, store all task outputs on the filesystem, since it is relatively simple to implement and it provides support for basic data resiliency out-of-the-box. A lot of existing software (that might be executed by a task) also makes liberal use of the filesystem, which can make it challenging to avoid filesystem access altogether. However, HPC nodes might not contain any local disks. Instead, they tend to use shared filesystems accessed over a network. While this can be seen as an advantage, since with a shared filesystem it is much easier to share task outputs among different workers, it can also be a severe bottleneck. Shared networked filesystems can suffer from 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 large, and thus storing it to a disk can be a bottleneck even without considering networked filesystems.

These bottlenecks can be alleviated by transferring data directly over the network (preferably without accessing the filesystem in the fast path). Direct network transfer of task outputs between workers is implemented e.g. by Dask. Some runtimes (such as HyperLoom [78]) also leverage RAM disks, which provide support for tasks that need to interact with a filesystem while avoiding the performance bottlenecks associated with disk accesses. Some task runtimes, such as HyperShell or Pegasus, allow streaming standard output and error streams over the network to a centralized location to reduce filesystem pressure. For all of these approaches, it is also possible to use HPC-specific technologies, such as InfiniBand [112] or MPI, to improve data transfer performance by fully exploiting the incredibly fast interconnects available in modern HPC clusters.

4.4 Fault tolerance

Fault tolerance is relevant in all distributed computational environments, but HPC systems have specific requirements in this regard. As was already mentioned, computational resources on HPC clusters are provided through allocation managers. Computational nodes allocated by these managers are provided only for a limited duration, which means that for long-running computations, some nodes will disconnect and new nodes might appear dynamically during the lifecycle of the executed workflow. Furthermore, since the allocations go through a queue, it can take some time before new computational resources are available. Therefore, the computation can remain in a paused state, where no tasks are being executed, for potentially long periods of time.

It is important for task runtimes to be prepared for these situations; they should handle node disconnections gracefully even if a task is being executed on a node that suddenly disconnects, and they should be able to restart previously interrupted tasks on newly arrived workers. In general, in HPC scenarios, worker instability and frequent disconnects should be considered the normal mode of operation rather than just a rare edge case.

Tasks are usually considered to be atomic from the perspective of the runtime, i.e. they either execute completely (and successfully), or they fail, in which case they might be restarted from scratch. Task granularity thus plays an important role here, since when a task is large, then a lot of work might have to be redone if it fails and is re-executed. Some runtimes try to alleviate this cost by leveraging task checkpointing [113], which is able to persist the computational state of a task during its execution, and then restore it in case of a failure, thus avoiding the need to start from the beginning.

Fault tolerance can be challenging in the presence data transfers between dependent tasks. When a task requires inputs from its dependency, the runtime might have to store them (either in memory or in a serialized format on disk) even after the task has started executing. Because there is always a possibility that it will have to restart the task in case of a failure, and thus it needs to hold on to its inputs. In some cases, it can actually be a better trade-off to avoid storing the inputs and instead re-execute the dependencies of the task (even if they have been executed successfully before) to regenerate the inputs. This can help reduce memory footprint, albeit at the cost of additional computation time, and it also might not work well if the execution of dependencies are not deterministic.

Because of its importance, fault-tolerant task execution is generally well-supported in most existing task runtimes. The differences are mostly in the level of automation. For example, FireWorks or Merlin require users to manually restart failed tasks, while runtimes such as Dask or Balsam can restart them automatically without user intervention.

An additional aspect of fault tolerance is persistence of submitted tasks. It can be useful to have the ability to resume task graph computation if the whole task runtime infrastructure (e.g. its primary server) crashes, to avoid needless recomputations. Some task runtimes, such as Dask, do not support such fault tolerance at all, others support it optionally, such as Ray, while some task runtimes are persistent by default, such as Balsam.

4.5 Multi-node tasks

Many existing HPC applications are designed to be executed on multiple (potentially hundreds or even thousands) nodes in parallel, using e.g. MPI or other communication frameworks. Multi-node execution could be seen as a special kind of resource requirement, which states that a task should be executed on multiple workers at once. Support for multi-node tasks is challenging, because it 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 were constantly trying to backfill available workers with single-node tasks, then multi-node tasks could be starved.

The scheduler might thus have to resort to keeping 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, where a task produces a set of complete data objects, which can then be used as inputs for dependent tasks. With multi-node tasks, the data distribution patterns become more complex, because when a task that is executed on multiple nodes produces a data object, the object itself might be distributed across multiple workers, which makes it more challenging to use in follow-up tasks. The task graph semantics might have to be extended by expressing various data distribution strategies, for example a reduction of data objects from multiple nodes to a single node, to support this use-case. This use-case is supported e.g. by PyCOMPSs [77].

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 of data between nodes.

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 is generally more complex. For example, when a task is being executed 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 number of nodes or by failing immediately).

Some task runtimes only consider single-node tasks in their programming model, which forces users to use various workarounds, such as emulating a multi-node task with a single-node task that uses additional resources that are not managed by the task runtime itself. In other tools, such as AutoSubmit, it is only possible to express node counts on the level of individually submitted

allocations. While this is useful for running coarse-grained MPI applications, it does not allow combining these multi-node applications with other, fine-grained single-node tasks in the same task graph. One notable exception is PyCOMPSs [77], which allows decorating Python functions that represent tasks with an annotation that specifies how many nodes does that task require, and combine these tasks with other single-node tasks.

4.6 Deployment

Even though it might sound trivial at first, an important aspect that can affect the ergonomics of executing task graphs (or any kind of computation, in general) on an HPC cluster is ease-of-deployment. Supercomputing clusters are notable for providing only a severely locked-down environment for their users, which does not grant elevated privileges and requires users to either compile and assemble the build and runtime dependencies of their tools from scratch or choose from a limited set of precompiled dependencies available on the cluster. However, these precompiled dependencies can be outdated or incompatible with one another.

Deploying any kind of software (including task runtimes) that has non-trivial runtime dependencies or non-trivial installation steps (if it is not available in a precompiled form for the target cluster) can serve as a barrier for using it on an HPC cluster. Many existing task runtimes are not trivial to deploy. For example, several task runtimes, such as Dask, SnakeMake or PyCOMPSs, are implemented in Python, a language that typically needs an interpreter to be executed, and also a set of packages in specific versions that need to be available on the filesystem of the target cluster. Installing the correct version of a Python interpreter and Python package dependencies can already be slightly challenging in some cases, but there can also be other issues, such as the dependencies of the task runtime conflicting with the dependencies of the software executed by the task graph itself. For example, since Dask is a Python package, it depends on a certain version of a Python interpreter and a certain set of Python packages. It supports tasks that can directly execute Python code in the process of a Dask worker. However, if the executed code requires a different version of a Python interpreter or Python packages than Dask itself, this can lead to runtime errors or subtle behavior changes[3].

A much bigger challenge is if a task runtime requires some external runtime dependencies. For example, Merlin requires a message broker backend, such as RabbitMQ or Redis, FireWorks needs a running instance of the MongoDB database, and Balsam uses the PostgreSQL database for storing task metadata. Compiling, configuring and deploying these complex runtime dependencies on HPC clusters can be quite challenging.

4.7 Programming model

One important aspect that affects the simplicity of defining and executing task graphs is the programming model and interfaces used to describe the execution logic, resource requirements,

[3]It is not straightforward to use multiple versions of the same Python package dependency in a single Python virtual environment, because conflicting versions simply override each other.

dependencies and other properties of tasks and task graphs. Below is a list of areas that affect the ergonomic aspects of task graph definition.

Task definition interfaces The method used to define tasks affects how easy it is to work with the task runtime, and how complex use-cases can be expressed with it. Existing interfaces usually belong to one of three broad categories. Tools such as GNU PARALLEL or HYPERSHELL offer a CLI (Command-line Interface), which enables defining tasks that should be computed directly in the terminal. This provides a quick way to submit structurally simple task graphs, although it is difficult to describe complex dependencies between tasks using this approach. A more general method of defining task graphs is to use a declarative workflow file, often using the YAML (Yet Another Markup Language) format. This approach is used e.g. by PEGASUS, AUTOSUBMIT or MERLIN. Since they are declarative, workflow files make it easier to analyze the task graph description, for example for providing data provenance. However, they can also be incredibly verbose, and are not suited for task graphs with a large number of tasks (unless the workflow file is automatically generated or it supports some form of compressed representation for groups of tasks). The most general (and very popular) method for describing task graphs is to leverage a general purpose programming language, very often Python. Many task runtimes provide a Python API for defining tasks, e.g. DASK, RAY, PARSL, BALSAM or PEGASUS. This approach allows users to create task graphs however they like, although it can be unnecessarily verbose for simple use-cases.

Data transfers Direct transfers of data between tasks have already been mentioned as a potential performance bottleneck, yet they also affect the ease-of-use of a given programming model. Some use-cases are naturally expressed by tasks passing their outputs directly to depending tasks. Support for this feature can make such use-cases much simpler, by not forcing users to pass intermediate task outputs through the filesystem.

Iterative computation Another important aspect of the programming model is its support for iterative computation. There are various HPC use-cases that are inherently iterative, which means that they perform some computation repeatedly, until some condition (which is often determined dynamically) is satisfied. For example, the training of a machine-learning model is typically performed in iterations (called epochs) that continue executing while the prediction error of the model keeps decreasing. Another example could be a molecular dynamics simulation that is repeated until a desired level of accuracy has been reached.

One approach to model such iterative computations would be to run the whole iterative process inside a single task. While this is simple to implement, it might not be very practical, since such iterative processes can take a long time to execute, and performing them in a single task would mean that we could not leverage desirable properties offered by the task graph abstraction, for example fault tolerance. Since the computation within a single task is typically opaque to the task runtime, if the task fails, it would then need to be restarted from scratch.

A better approach might be to model each iteration as a separate task. In such case, the task runtime is able to restart the computation from the last iteration if a failure occurs. However, this approach can be problematic if the number of iterations is not known in advance, since some

task runtimes expect that the structure of the task graph will be immutable once the graph has been submitted for execution. Dask and Ray are examples of task runtimes do not have this assumption; they allow adding tasks to task graphs dynamically during their execution.

Another option to handle iterative tasks is provided by L-DAG [114], which suggests an approach for transforming iterative workflows into workflows without loops.

Summary

This chapter has identified a set of challenges that can limit the usage ergonomics and efficiency of task graphs on supercomputers, and described how do existing task runtimes deal with them. Even though more HPC peculiarities can always be found, it should be already clear from all the mentioned challenges that HPC use-cases that leverage task graphs can be very complex and may require specialized approaches in order to reach maximum efficiency while remaining easy to use and deploy. The most important challenge that was described is the interaction of tasks with allocations, which has a fundamental effect on the achievable hardware utilization of the target cluster, and on the overall simplicity of defining and executing task graphs.

The rest of the thesis will discuss approaches for alleviating these challenges. Chapters 5 and 6 focus on the performance aspects of task graph execution, namely on task scheduling and the overhead introduced by task runtimes. Chapter 7 then also deals with the ergonomic challenges, and proposes a meta-scheduling and resource management design that is built from the ground up with the mentioned challenges in mind, to enable truly first-class task graph execution on heterogeneous HPC clusters.

Chapter 5

Task scheduling analysis

Task scheduling is one of the most important responsibilities of a task runtime, because the quality of scheduling has an effect on the makespan of the task graph execution and also on the achieved hardware utilization of worker nodes. It is crucial for the scheduler to be able to distribute the tasks among all available worker nodes to achieve as much parallelization as possible, without the induced network communication and task management overhead becoming a bottleneck. Unfortunately, optimal task scheduling is a very difficult problem, which is NP-hard even in the simplest cases [69], and there is thus no single scheduling algorithm that could quickly provide an optimal schedule for an arbitrary task graph.

There are many factors that affect the execution properties of task graphs and that pose some form of a challenge to task schedulers. The computational environment (e.g. a distributed cluster) can have varying amounts of nodes with heterogeneous hardware resources, and complex network topologies that can have a non-trivial effect on the latency and bandwidth of messages sent between the workers and the scheduler, and thus in turn also on the overall performance of the task graph execution. Task graphs can also have an arbitrarily complex structure, with large amounts of different kinds of tasks with diverse execution characteristics and resource requirements.

Furthermore, task graph execution might not be deterministic, and the scheduler has to work with incomplete information and react to events that dynamically occur during task execution and that cannot be fully predicted before the task graph execution starts. The communication network can be congested because of unrelated computations running concurrently on the cluster; tasks can also be slowed down by congested hardware resources that can be highly non-trivial to model, such as NUMA effects, and they can also sometimes fail unpredictably and have to be re-executed. Even the duration of each task, which is perhaps the most crucial property of a task coveted by the scheduler, is not usually known beforehand; the most the scheduler knows about is either an estimate from the task graph author or a running average based on historical executions of similar tasks, both of which can be inaccurate.

In theory, all these factors should be taken into account by task scheduling algorithms. In practice, it is infeasible to have a completely accurate model of the entire cluster, operating system, task implementations, networking topology, etc. Therefore, task schedulers omit some of these factors to provide a reasonable runtime performance. They rely on various heuristics with different trade-offs

that make them better suited for specific types of task graphs and computational environments. These heuristics can suffer from non-obvious edge cases that produce poor quality schedules or from low runtime efficiency, which can in turn erase any speedup gained from producing a higher quality schedule.

In HPC use-cases, the performance and quality of task scheduling is even more important, since the scale and heterogeneity of task graphs provides an additional challenge for the scheduler. HPC clusters also tend to contain advanced network topologies with low latency and high bandwidth [115, 116], which offer the scheduler more leeway to create sophisticated schedules leveraging large amounts of network communication, which would otherwise be infeasible on clusters with slower networks.

To better understand the behavior and performance of various scheduling algorithms, and to find out which scheduling approach is best suited for executing task graphs on distributed clusters, we have performed an extensive analysis of several task scheduling algorithms in *Analysis of workflow schedulers in simulated distributed environments* [1]. The two main contributions of this work are as follows:

1. We have created an extensible, open-source simulator of task graph execution, which allows users to easily implement their own scheduling algorithms and compare them, while taking into account various factors that affect task scheduling.
2. We have benchmarked several task schedulers from existing literature under various conditions, including factors affecting scheduling that have not been explored so far to our knowledge, like the minimum delay between invoking the scheduler or the amount of knowledge about task durations available to the scheduler, and evaluated the suitability of the individual algorithms for various types of task graphs. All parts of the benchmark suite, including the task graphs, source codes of the scheduling algorithms, the simulation environment and also benchmark scripts are provided in an open and reproducible form.

Various descriptions of schedulers, task graphs and other parts of the simulator and the benchmark configuration used in this chapter were adapted from our publication [1].

I have collaborated on this work with Ada Böhm and Vojtěch Cima, we have all contributed to it equally. Source code contribution statistics for Estee *can be found on GitHub*[1].

5.1 Task graph simulator

To analyze scheduling algorithms, some form of an environment for executing tasks has to be used. One possibility would be to use an actual distributed cluster, and implement multiple schedulers into an existing task runtime. However, this approach can be expensive, both computationally (executing a large number of task graphs with various configurations would consume a lot of cluster computational time) and implementation-wise (adapting existing runtimes to different scheduling algorithms is challenging). Therefore, task graph scheduling surveys tend to use some form of a simulated environment, which simulates selected properties of a distributed cluster, and allows

[1] `https://github.com/it4innovations/estee/graphs/contributors`

comparing the performance of multiple scheduling algorithms (or other factors of a task runtime) with a reduced accuracy, but at a fraction of the cost.

Many task scheduler surveys have been published over the years [70, 71, 67, 117, 68], yet it is difficult to reproduce and extend these results without having access to the exact source code used to implement the schedulers and the simulation environment used in these surveys. As we will show in the following chapter, the performance of scheduling algorithms can be affected by seemingly trivial implementation details, and having access only to a high-level textual description or pseudocode of a scheduling algorithm does not guarantee that it will be possible to reproduce it independently with the same performance characteristics. This makes it challenging to compare results between different simulation environments.

Apart from the environments used in existing surveys, there are also more general task simulation environments. DAGSim [118] offers a framework for comparing scheduling algorithms, and compares the performance of a few algorithms, but does not provide its implementation, which makes it difficult to reproduce or extend its results. SimDAG [119] is a task graph simulator focused on HPC use-cases built on top of the SimGrid [120] framework. It allows relatively simple implementation of new task scheduling algorithms; however, it does not support any task resource requirements (e.g. the number of used CPU cores), which are crucial for simulating heterogeneous task graphs.

In addition to simply comparing the performance of different schedulers, our goal was also to test two factors for which we have hypothesized that they might affect scheduling; we have not seen these explored in detail in existing works. Namely, we wanted to examine the effects of Minimal Scheduling Delay, the delay between two invocations of the scheduler and *information mode*, the amount of knowledge of task durations that is available to the scheduler. These factors will be described in detail in the following section. The existing simulation environments that we have evaluated did not have support for these factors, and it would be non-trivial to add support for them.

To summarize, our goal was to provide a simulation environment that would be open-source, facilitate reproducibility, support basic task resource requirements, and enable us to examine the two mentioned factors that affect scheduling. To fulfill these goals, we have implemented a task graph simulation framework called Estee. It is an MIT-licensed open-source tool [121] written in Python that provides an experimentation testbed for task runtime and scheduler developers and researchers. It is flexible; it can be used to define a cluster of workers, connect them using a configurable network model, implement a custom scheduling algorithm and test its performance on arbitrary task graphs, with support for specifying required CPU core counts for individual tasks. Additionally, it comes "battery-included"; it contains baseline implementations of several task schedulers from existing literature and also a task graph generator that can be used to generate randomized graphs with properties similar to real-world task graphs.

5.1.1 Architecture

Figure 5.1 shows the architecture of Estee. The core of the tool is the *Simulator* component, which uses discrete event simulation [122] to simulate the execution of a task graph. It manages

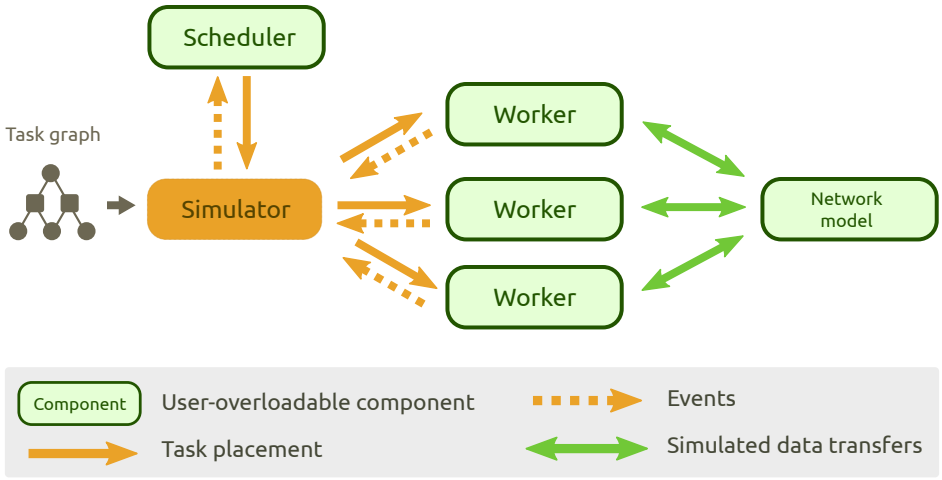


Figure 5.1: ESTEE architecture

tasks, queries the *Scheduler* component for task-to-worker assignments (schedules) and then assigns the tasks to their corresponding workers. It also ensures that all task graph executions satisfy the dependency constraint (Definition 4) and both task and worker constraints (Definition 5).

The *Worker* component simulates task execution and uses the provided network model to simulate exchanges of data (task outputs) between individual workers of the simulated cluster.

ESTEE provides abstract interfaces for the task scheduler, the worker and the network model (which simulates network communication and contention). Users can thus easily provide their own implementations of these interfaces, and in turn override both the behavior of the scheduler and of the cluster and its network topology.

One of our goals for ESTEE was to make it very easy to write new scheduling algorithms and make the scheduler approachable for other researchers that might want to experiment with task schedulers. That was also one of the motivations for deciding to create ESTEE in Python, which facilitates experimentation. Listing 5.1 shows an example of a task graph simulation that demonstrates the simplicity of defining a task graph simulation using ESTEE. The output of the simulation is both the makespan and also a detailed trace that can be used to visualize the individual task-to-worker assignments and task execution time spans.

ESTEE supports general task graphs represented by a DAG. Each task has an associated duration, and can contain multiple outputs (data objects), each with an associated size. It can also specify how many cores it requires, to model the common requirement of executing multi-threaded functions and programs on modern HPC machines. Tasks do not depend directly on other tasks; dependencies are formed between a task and a specific data output of another task. The used task graph model thus corresponds to the task graph definition introduced in Chapter 3.

5.1.2 Communication model

Some previous scheduler surveys assume that the time to transfer a data object from one worker to another depends merely on the size of the data object, and not on other factors, such as current

```python
from estee.common import TaskGraph
from estee.schedulers import BlevelGtScheduler
from estee.simulator import Simulator, Worker, MaxMinFlowNetModel

# Create a task graph containing 3 tasks
# Each task runs for 1s and requires 1 CPU core
#
#     t0
#     |
#     o (50MB output)
#    / \
#  t1   t2
tg = TaskGraph()
t0 = tg.new_task(duration=1, cpus=1, output_size=50)
t1 = tg.new_task(duration=1, cpus=1)
t1.add_input(t0)
t2 = tg.new_task(duration=1, cpus=1)
t2.add_input(t0)

# Create a task scheduler
scheduler = BlevelGtScheduler()

# Define cluster with 2 workers (1 CPU core each)
workers = [Worker(cpus=1) for _ in range(2)]

# Define MaxMinFlow network model (100MB/s bandwidth)
netmodel = MaxMinFlowNetModel(bandwidth=100)

# Run simulation, return the estimated makespan in seconds
simulator = Simulator(tg, workers, scheduler, netmodel, trace=True)
makespan = simulator.run()
print(f"Task graph execution makespan = {makespan}s")
```

Listing 5.1: Simple task graph simulation example using Estee

network utilization or interference [123, 124, 68, 125]. This is an unrealistic assumption, as the latency and bandwidth of actual computer networks is affected (among other things) by other communication happening concurrently on the same network. Moreover, a real worker implementation would download more than a single data object simultaneously, which further affects the transfer durations, because the worker's bandwidth will be shared by multiple network transfers. We will use the term *communication model* and *network model* interchangeably in this chapter.

We provide a more realistic network model that simulates full-duplex communication between workers, where the total (data object) upload and download bandwidth of each worker is limited. The sharing of bandwidth between worker connections is modeled by the *max-min fairness model* [126]. Max-min fairness provides a bandwidth allocation for each worker. If an allocation of any participant is increased, then we decrease the allocation of some other participant with an equal or smaller allocation. When a data object transfer starts or finishes, the data flow between workers is recomputed immediately; thus we neglect the fact that it may take some time for the bandwidth to fully saturate.

This model is not as accurate as e.g. packet-level simulation implemented in some other simulators [120], but it is a notable improvement over the naive model and it provides reasonable runtime performance. To provide a baseline that corresponds to the naive model described above, which has been used in several previous works, ESTEE also implements a *simple* network model. The used networking model can be configured with an arbitrary network bandwidth amount for a given simulation.

5.1.3 Scheduler parameters

ESTEE implements support for two parameters that can affect scheduler performance, and which we have not seen examined in detail in existing literature:

Minimal Scheduling Delay Non-trivial schedulers create task assignments continuously during task graph execution, based on the current worker load and task completion times. That means that they are not invoked only once, but rather the task runtime invokes them repeatedly to ask them to produce assignments for tasks that are (or soon will be) ready to be executed at any given point in time.

It then becomes important for a task runtime to decide when exactly it should invoke the scheduler. It could try to make a scheduling decision every time a task is finished; however, in practice there is often an upper bound on the number of scheduler invocations per second. It might be introduced artificially, to reduce the scheduling overhead, or it might be caused by a software or hardware limitation (e.g. messages containing task updates cannot be received more often). Furthermore, creating a new schedule after each task status change might not be optimal. The runtime can also accumulate changes for a short time period, and then provide the scheduler with a batch of status updates. While this increases the latency of task assignments, it can give the scheduler more context to work with, when it decides how it should assign tasks to workers.

To test our hypothesis that the scheduler invocation rate can affect its performance, we introduce a parameter called MSD (Minimal Scheduling Delay), which forces a minimal delay between two

scheduler invocations, i.e. the scheduler cannot be invoked again before at least MSD time units have elapsed since its previous invocation.

Information mode Many existing task scheduler descriptions assume that the duration of each task (and the size of each data object) is known in advance. However, this assumption is very seldom upheld when executing real-world task graphs. Tasks are usually specified using arbitrary function or binary invocations, and it is difficult to estimate their duration up front. Task runtimes thus have to work with completely missing information about task durations, depend on potentially imprecise user estimates, or calculate their own estimates based on historical task execution data. Task benchmarks usually use simulated task durations, which are provided to the scheduler. However, this might not realistically represent the scheduler's behavior for actual task graphs, for which we usually do not know task durations precisely before they are executed.

We use a parameter called *Information mode (imode)*, which controls the amount of knowledge the scheduler has of the duration of tasks. It can be set to one of the following values:

exact The scheduler has access to the exact duration of each task and the exact size of each data object in the whole task graph.

user The scheduler has access to user-defined estimations for each task in the task graph. These estimations are sampled from a random distribution that corresponds to a specific kind of task within the workflow. For example, in a task graph that performs three kinds of tasks (e.g. preprocessing, computation and postprocessing), each kind of task would have its own distribution. We have categorized the tasks of task graphs that we have used for scheduler benchmarks described in Section 5.2 manually, to simulate a user that has some knowledge of the task graph that they are trying to compute and is able to provide some estimate of task durations and data object sizes.

mean The scheduler only has access to the mean duration of all tasks and the mean size of all data objects in the executed task graph. This simulates a situation where a similar task graph is executed repeatedly, and thus there is at least some aggregated information about the task properties available from an earlier run.

Another possible mode to consider could be to not provide the scheduler with any task durations nor data object sizes in advance. This behavior would in fact correspond closely to a real-world execution of a task graph, where we usually do not know these task properties a priori. However, it is challenging to use this approach when benchmarking schedulers. Scheduler implementations are typically described with the assumption that task durations are known, and the scheduling algorithms are often fundamentally based on calculations that make use of them.

If we took away this information, some schedulers would not be able to function, as their behavior is strongly influenced by an estimate of the duration of each task. Therefore, we propose using the *mean* mode instead of not providing the scheduler with any information. We assume that even if the scheduler knows nothing in advance, it could always gradually record the durations and sizes of finished tasks, and these values would eventually converge to the global mean. In practice, this would take some time, while in our environment the schedulers know about the mean

in advance. Nevertheless, as was already mentioned, we can often get a reasonable estimate of the mean durations based on previous executions of similar workflows.

5.1.4 Schedulers

There are many task scheduling approaches, and an enormous number of various task scheduling algorithms. We have implemented a set of task schedulers that are representatives of several common scheduling approaches, inspired by a list of schedulers from a survey performed by Wang and Sinnen [68]. We have included several representatives of the simplest and perhaps most common scheduling approach, called list-scheduling, where the scheduler sorts tasks based on some priority criteria, and then repeatedly chooses the task with the highest priority and assigns it to a worker, which is selected by some heuristic. In addition to list-scheduling, we have also implemented more complex approaches, such as schedulers that use work-stealing or genetic algorithms.

Below is a list of schedulers that we have implemented and benchmarked[2]:

blevel HLFET (Highest Level First with Estimated Times) [70] is a foundational list-based scheduling algorithm that prioritizes tasks based on their *b-level*. B-level of a task is the length of the longest path from the task to any leaf task (in our case the length of the path is computed using durations of tasks, without taking data object sizes into account). The tasks are scheduled in a decreasing order based on their b-level.

tlevel Smallest Co-levels First with Estimated Times [127] is similar to HLFET, with the exception that the priority value computed for each task (which is called *t-level* here) is computed as the length of the longest path from any source task to the given task. This value corresponds to the earliest time that the task can start. The tasks are scheduled in an increasing order based on their t-level.

dls Dynamic Level Scheduling [128] calculates a dynamic level for each task-worker pair. It is equal to the static b-level lessened by the earliest time that the task can start on a given worker (considering necessary data transfers). In each scheduling step, the task-worker pair that maximizes this value is selected.

mcp The Modified Critical Path [129] scheduler calculates the ALAP (as-late-as-possible) time for each task. This corresponds to the latest time the task can start without increasing the total schedule makespan. The tasks are then ordered in ascending order based on this value, and scheduled to the worker that allows their earliest execution.

etf The ETF (Earliest Time First) scheduler [130] selects the task-worker pair that can start at the earliest time at each scheduling step. Ties are broken by a higher b-level precomputed at the start of task graph execution.

genetic This scheduler uses a genetic algorithm to schedule tasks to workers, using the mutation and crossover operators described in [131]. Only valid schedules are considered; if no valid schedule can be found within a reasonable number of iterations, a random schedule is generated instead.

ws This is an implementation of a simple work-stealing algorithm. The default policy is that each task that is ready to be executed (all its dependencies are already computed) is always assigned

[2]The labels of the individual schedulers correspond to labels used in charts that will be presented in Section 5.2.

to a worker where it can be started with a minimal transfer cost. The scheduler then continuously monitors the load of workers. When a worker starts to starve (it does not have enough tasks to compute), a portion of tasks assigned to other workers is rescheduled to the starving worker.

In addition to these schedulers, we have also implemented several naive schedulers, which serve as a baseline for scheduler comparisons.

single This scheduler simply assigns all tasks to a single worker (it selects the worker with the most cores). The resulting schedule never induces any data transfers between workers, and does not take advantage of any parallelism between workers.

random This scheduler simply assigns each task to a random worker using a PRNG (Pseudorandom Number Generation) engine.

We have strived to implement the mentioned list-based schedulers (*blevel*, *tlevel*, *dls*, *mcp*, *etf*) as closely as possible to their original description. These list-based algorithms mostly focus on selecting the next task to schedule, but an important question (that comes up during their implementation) is to what worker should the selected task be scheduled. The algorithm descriptions often mention assigning the task to a worker that allows the earliest start time of the task. While that is surely a reasonable heuristic, it is not clear how exactly such a worker should be found, because the exact earliest start time often cannot be determined precisely in advance, since its calculation might encompass network transfers whose duration is uncertain. This seemingly simple implementation detail is crucial for implementing the scheduler, and it should thus be included in the description of all scheduling algorithms that make use of such a heuristic.

Estee implementations of these schedulers use a simple estimation of the earliest start time, which is based on the currently executing and already scheduled tasks of a worker and an estimated network transfer cost based on uncongested network bandwidth (the *simple* network model is used for the scheduler's estimation of the network transfer cost).

In order to test our hypothesis that the worker selection approach is important and affects the scheduler's behavior, we have also created extended versions of the *blevel*, *tlevel* and *mcp* schedulers. These modified versions use a worker selection heuristic called "greedy transfer". We have not applied this heuristic to other list-based schedulers, because it would fundamentally change their behavior.

The greedy transfer heuristic assigns the selected task to a worker that has a sufficient number of free cores on which the task may be executed, and that requires the minimal data transfer (sum over all sizes of data objects that have to be transferred to that worker). It also adds support for clusters where some machines have a different number of cores than others. When a task t that needs c cores cannot be scheduled because of an insufficient number of free cores, the list-scheduling continues by taking another task in the list instead of waiting for more free cores. This task will only consider workers that have fewer than c cores. This allows for scheduling more tasks while it does not modify the priority of tasks because t cannot be scheduled on such workers anyway. Note that when all workers have the same number of cores, the behavior is identical to ordinary list-scheduling.

5.1.5 Task graphs

To facilitate task scheduler experiments, Estee contains a task graph generator that is able to generate parametrized instances of various categories of task graphs. Graphs from each category can be generated using several parameters that affect their resulting size and shape. To increase the variability of the graphs, properties like task durations or data object sizes are sampled from a normal distribution. Below is a description of the three categories of task graphs that can be generated:

elementary This category contains trivial graph shapes, such as tasks with no dependencies or simple "fork-join" graphs. These graphs can test the behavior of scheduler heuristics on basic task graph building blocks that frequently form parts of larger workflows. Examples of these graphs can be found in Appendix A (Figure A.1).

irw This generator creates graphs that are inspired by real-world task graphs, such as machine-learning cross-validations or map-reduce workflows.

pegasus This category is derived from graphs created by the Synthetic Workflow Generators [132]. The generated graphs correspond to the *montage*, *cybershake*, *epigenomics*, *ligo* and *sipht* Pegasus workflows. The graphs have been extended with additional properties required for testing information modes (notably expected task durations and data object sizes for the *user* information mode).

5.2 Task scheduler evaluation

We have carried out an extensive analysis of the performance of several task scheduling algorithms on various task graphs using the Estee simulator. The aim of the analysis was to explore the behavior of various schedulers in a complex simulation environment. In addition to comparing the schedulers among each other, we also wanted to test how their performance differs between various communication models and scheduler parameters.

Note that since the used simulation environment is focused on simulating different task graph schedules and network transfers, and it does not model the actual execution of the scheduler nor the task runtime in a cycle-accurate way, the term *scheduler performance* refers to the simulated makespan of task graphs executed using schedules provided by the given scheduler, as defined by Definition 6. In other words, our experiments estimate how quickly a given task graph would be fully computed on a cluster, using a given network model, while being scheduled by a specific scheduling algorithm.

5.2.1 Benchmark configuration

Below, you can find descriptions of the cluster, scheduler and task graph configurations that we have used for our benchmarks.

Task graphs The Estee graph generators were used to generate a collection of task graphs that were used in the benchmarks. The properties of all used graphs are summarized in Appendix A (Table A.1). The generated task graph dataset is available as a reproducible artifact [133].

Schedulers We have benchmarked all schedulers described in Section 5.1.4. Schedulers that use the greedy transfer heuristic are labeled in the benchmark results with a *-gt* suffix.

Scheduler parameters To evaluate the effect of minimal scheduling delay, we have used a baseline value of zero, where the scheduler is invoked immediately after any task status update, and then a delay of 0.1, 0.4, 1.6 and 6.4 seconds. In the cases where MSD is non-zero, we have also added a 50 milliseconds delay before sending the scheduler decision to workers, to simulate the time taken by the scheduler to produce the schedule. For experiments that do not focus on MSD, we always use an MSD of 0.1 seconds and the 50 milliseconds computation delay. To evaluate information modes, we have used the *exact*, *user* and *mean* imodes. For experiments that do not focus on imodes, we always use the *exact* mode.

Network models The simple (labeled *simple*) and max-min (labeled *max-min*) network models were used, with bandwidth speeds ranging from 32 MiB/s to 8 GiB/s. For experiments that do not focus on the network model (e.g. when imodes are being compared), we always use the *max-min* network model.

Clusters We have used the following five cluster (worker) configurations (where $w \times c$ means that the cluster has w workers and each worker has c cores): 8×4, 16×4, 32×4, 16×8, 32×16.

5.2.2 Evaluation

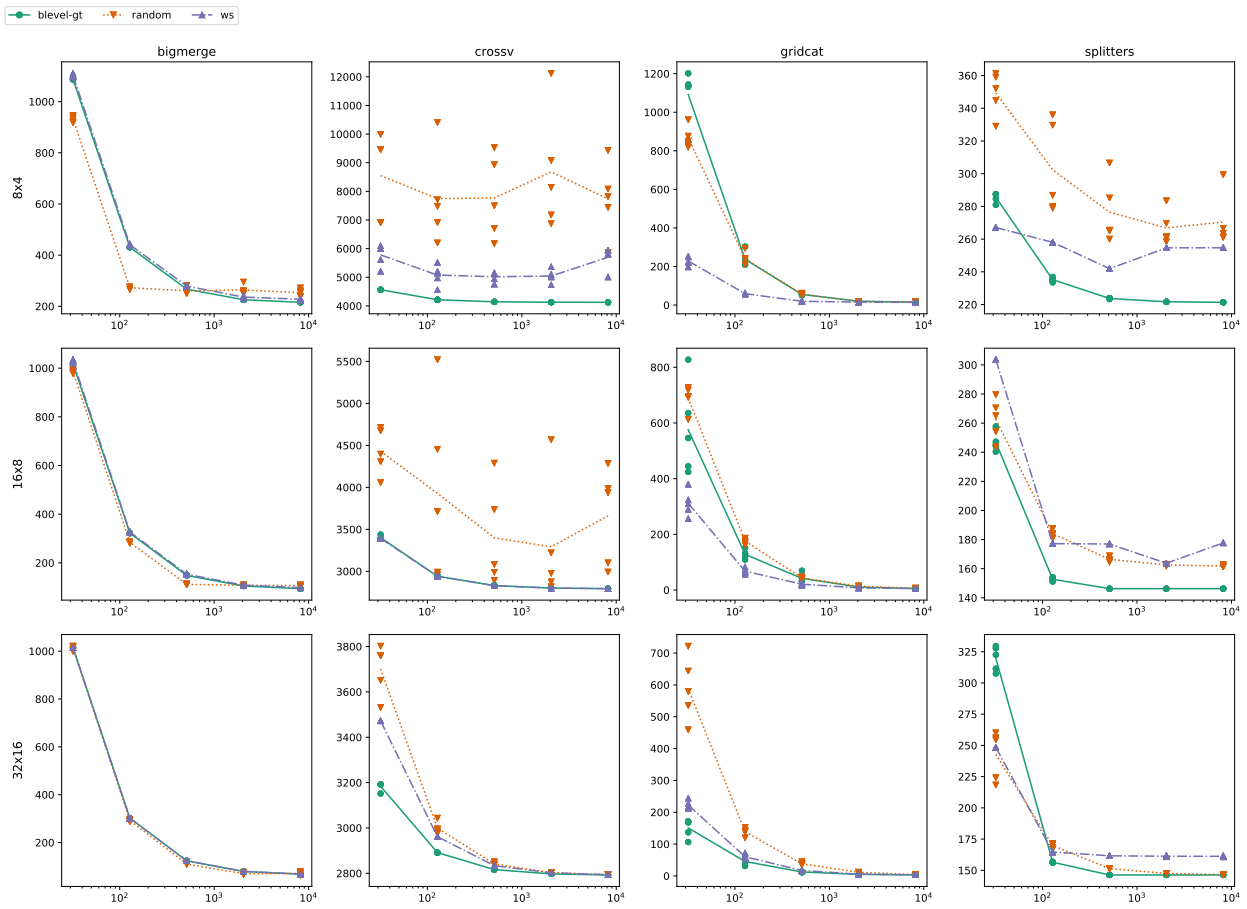
This section discusses selected results of the described benchmarks. Complete benchmark results and overview charts can be found in [1]. The benchmarking environment, input task graph datasets, all benchmark configurations, results and charts are also freely available as reproducible artifacts [134] for further examination.

The benchmarks were executed on the clusters of the IT4Innovations supercomputing center [135]. The actual characteristics of the cluster hardware is not important, because all benchmarks were executed using the Estee simulator, so the benchmark results do not depend on the used hardware. Each benchmark that was non-deterministic in any way (e.g. because it used a pseudorandom number generator) was executed twenty times. Unless otherwise specified, the individual experiments were performed with the default benchmark configuration that uses the *max-min* network model, the *exact* information mode and a Minimal Scheduling Delay of 0.1 s.

Note that the vertical axis of some charts presented in this section does not start at zero, as the goal was to focus on the relative difference between different scheduling algorithms rather than on the absolute makespan durations.

Random scheduler

Given the fact that task scheduling is an NP-hard problem, it would seem that a random scheduling approach should produce unsatisfying results. Therefore, we wanted to examine how a completely



horizontal axis: bandwidth [MiB/s]; vertical axis: makespan [s]; row: cluster

Figure 5.2: Performance of the *random* scheduler

random scheduler holds up against more sophisticated approaches. Figure 5.2 compares the simulated makespan durations of the *random* scheduler vs. two other competitive schedulers (*blevel-gt* and the work-stealing *ws* scheduler) on several task graphs.

While there are indeed cases where random scheduling falls short (for example on the cross-validation *crossv* task graph, or in situations with many workers and a slow network), in most cases its performance is similar to other schedulers, and in a few situations it even surpasses them. Its performance improves with increasing worker count and network speed. This makes intuitive sense, because if there are enough workers and the network is fast enough to overcome the cost of exchanging many data objects between them, the specific assignment of tasks between workers becomes less important. As long as the scheduler is able to keep the workers busy (which can be ensured even by a random schedule for some task graphs), then the resulting performance might be reasonable.

We have been able to validate these results in [2], where we have shown that as the worker count becomes larger, scheduling decisions can in some cases become less important, and other factors (like the overhead of the task runtime) might start to dominate the overall task graph execution cost. This will be described in more detail in Chapter 6.

Worker selection strategy

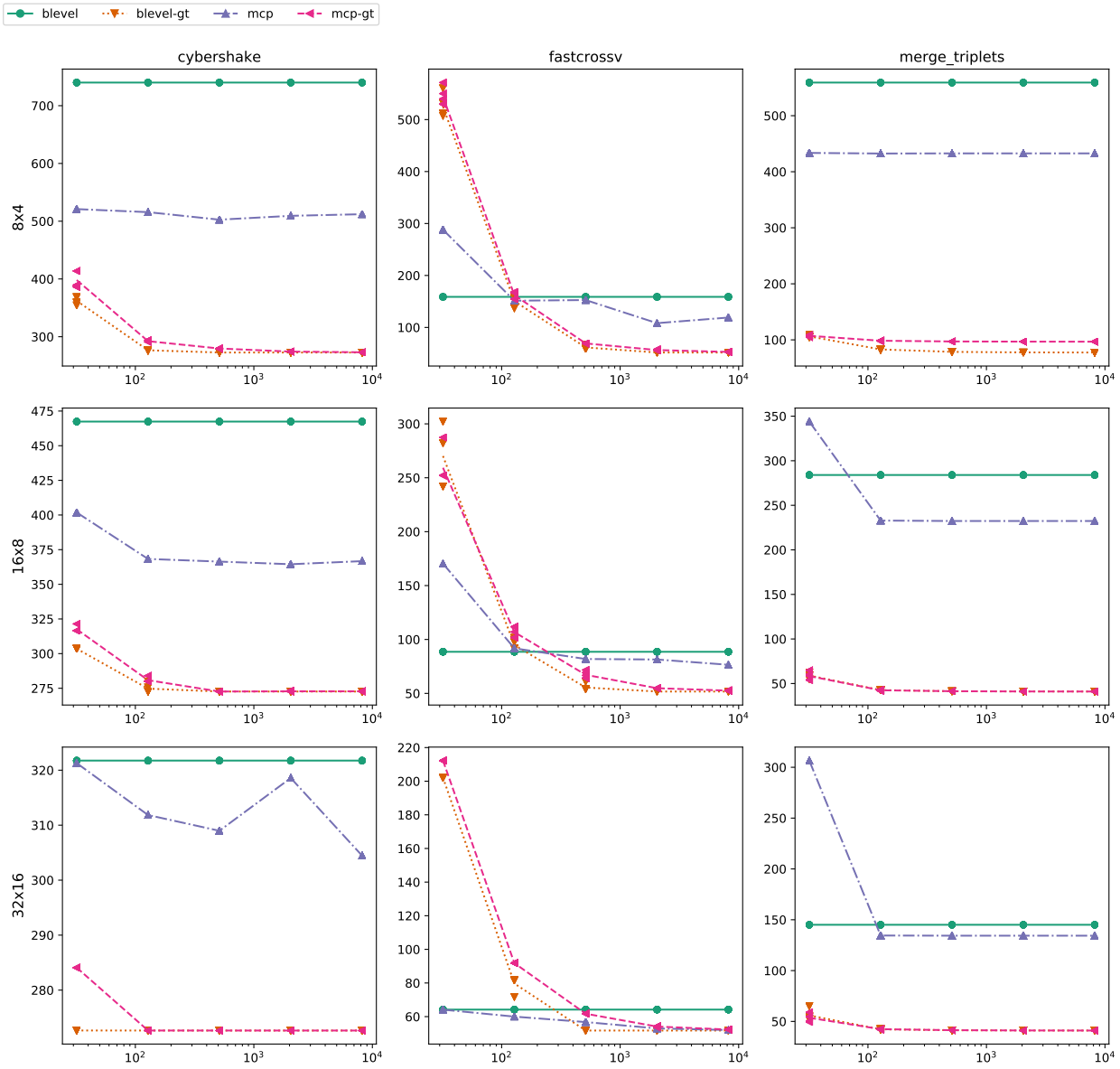
As was explained in Section 5.1.4, the descriptions of several schedulers that we have implemented in Estee do not specify the concrete strategy for selecting a worker that can start executing a given task as soon as possible. Yet, as we can see in Figure 5.3, this implementation detail is crucial. This chart shows the performance of two scheduling algorithms (*blevel* and *mcp*), each in two variants, with the simple selection strategy and with the greedy transfer strategy (the used worker selection strategy was the only difference between the simple and the *-gt* suffixed variants).

It can be seen that there is a large difference between these two strategies. In fact, the results suggest that in these specific scenarios, the worker selection strategy had a larger effect on the overall performance than the used scheduling (task selection) algorithm, as the variants using greedy transfer were highly correlated. This suggests that the used worker selection strategy is an important detail of list-scheduling algorithms that should not be omitted from their descriptions.

Network models

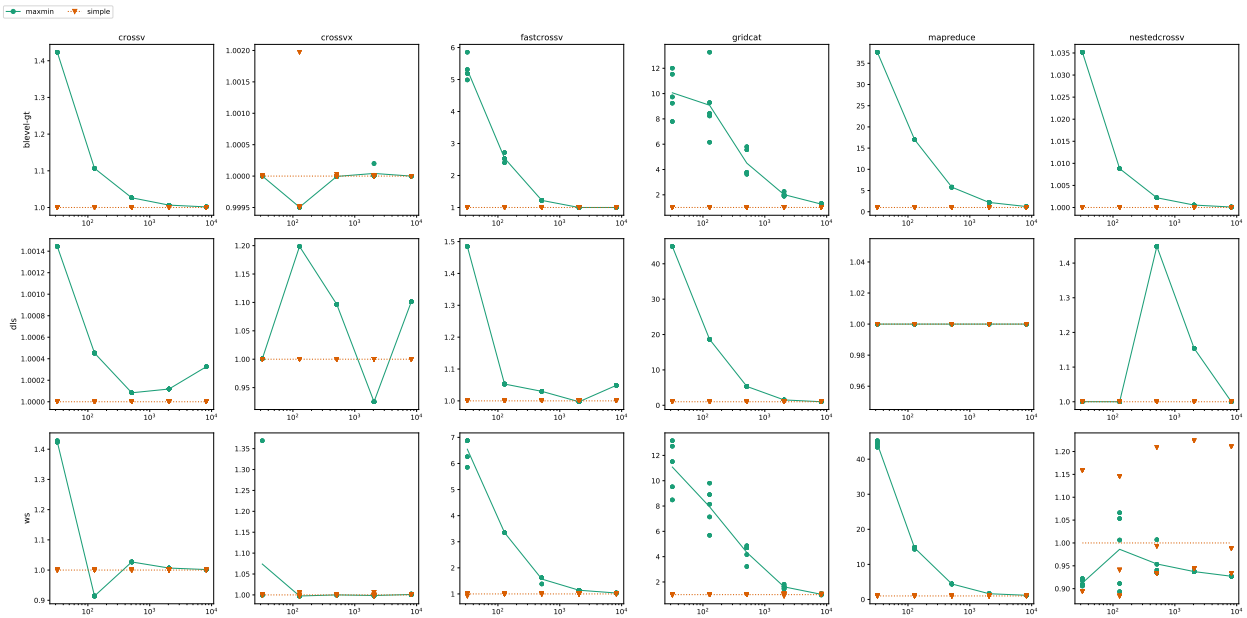
Figure 5.4 demonstrates how the used network model affects simulated task graph makespans for a selected set of task graphs and schedulers, using task graphs from the *irw* dataset on the $32x4$ cluster with 32 workers. The Y axis is normalized with respect to the average makespan of simulations performed with the *simple* network model.

It is clear that especially for slower network bandwidths, the naive *simple* model often underestimates the resulting makespan. This is caused by the fact that it does not take network contention into account at all, which causes the overall network transfer duration estimation to be overly optimistic. As network bandwidth goes up, the difference is reduced, since there is less overall contention and the transfers are faster in general.



horizontal axis: bandwidth [MiB/s]; vertical axis: makespan [s]; row: cluster

Figure 5.3: Comparison of worker selection strategy



horizontal axis: bandwidth [MiB/s]; vertical axis: makespan normalized to average of the *simple* model; row: scheduler; cluster $32x4$

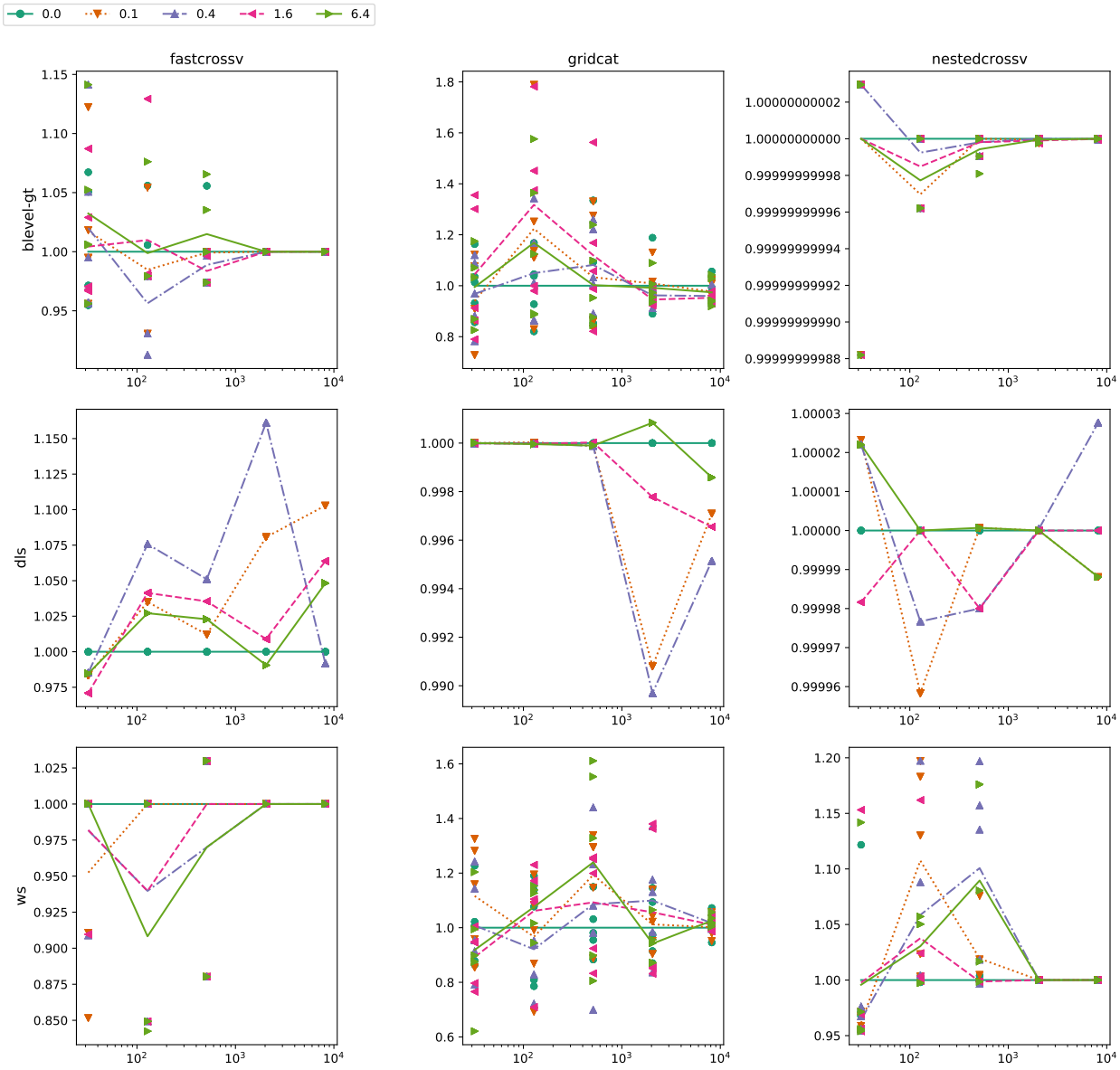
Figure 5.4: Comparison of *max-min* and *simple* network models (*irw* dataset)

The makespans of simulations with these two network models are sometimes up to an order of magnitude apart. This is quite significant, because the difference between the performance of schedulers (with a fixed network model) is otherwise usually within a factor of two, which was demonstrated both in [68] and by results of our other experiments. The gap between the two network models depends heavily on the used task graph.

Minimal Scheduling Delay

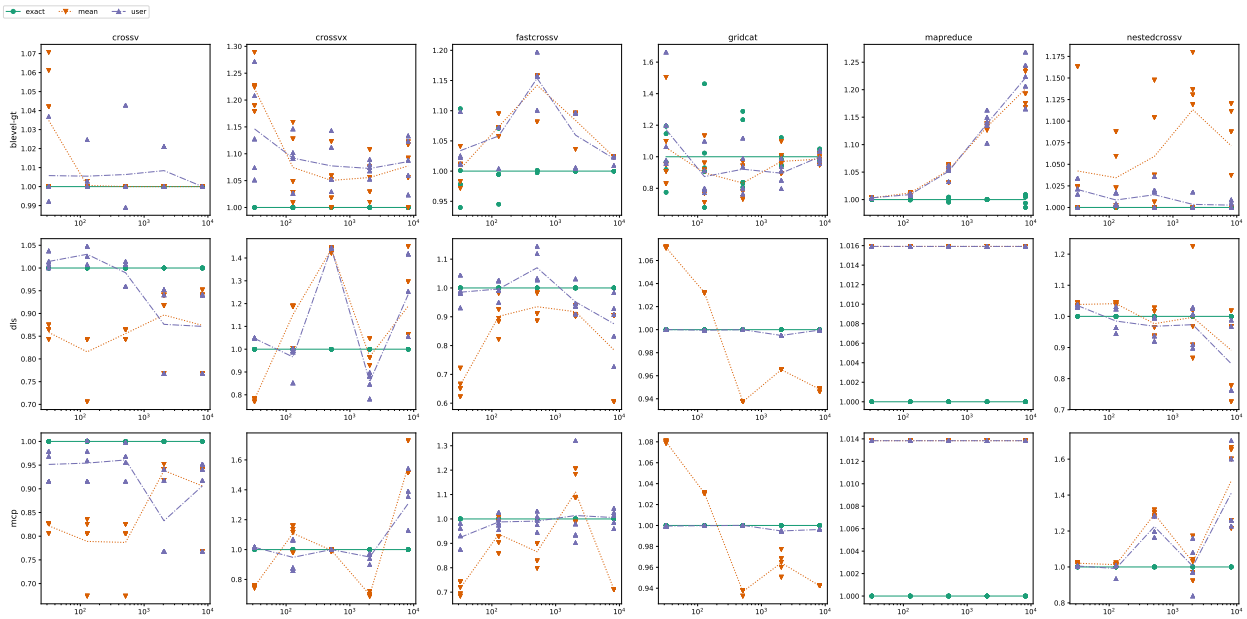
In Figure 5.5, we can observe the effect of MSD on graphs from the *irw* dataset, with the $32x4$ cluster configuration. The Y axis is normalized with respect to the configuration where MSD is zero. The results show that the effect of MSD is relatively limited. There does not seem to be any clear correlation or pattern that would suggest that a smaller MSD consistently improves performance of a scheduler. Although interestingly, a higher MSD value caused several makespan improvements, especially on the *gridcat* task graph.

This is another example of the non-trivial effect of scheduler heuristics. Increasing the MSD leads to a batching effect, where the scheduler is allowed to make decisions less often, but it has knowledge of more task events (that have arrived during the delay) during each decision. Whether this helps its performance, or hurts it, depends on the specific scheduler implementation and the task graph that it executes.



horizontal axis: bandwidth [MiB/s]; vertical axis: makespan normalized to $MSD = 0$

Figure 5.5: Comparison of MSD; cluster 32x4



horizontal axis: bandwidth [MiB/s]; vertical axis: makespan normalized to *exact* imode; row: scheduler; cluster $32x4$

Figure 5.6: Comparison of information modes (*irw* dataset)

Information modes

Figure 5.6 compares makespans of several scheduler and task graph combinations from the *irw* dataset on a $32x4$ cluster, with different information modes being used. The results are normalized to the mean makespan of the default *exact* information mode. In general, the effect of information modes is more significant than the effect of the Minimal Scheduling Delay.

An intuitive expectation would be that with more precise task duration information, the scheduler will be able to produce a shorter makespan, and this is indeed what happens in several cases, e.g. on the *mapreduce* and *nestedcrossv* task graphs with the *blevel-gt* scheduler, where the makespan is up to 25% longer when task durations are not exactly known.

However, there are also opposite cases, for example the *dls* and *mcp* schedulers produce better results on several task graphs when they take only the *mean* task duration into account. This further shows the effect of scheduler heuristics, which can produce worse results even when presented with more accurate data input (and vice versa).

One factor that makes it more difficult for the scheduler to accurately estimate the network transfer durations and thus make optimal use of the knowledge of task durations is that with the max-min network model, the scheduler knows only a lower bound on the communication costs, even if it knows the exact data size in advance. While it has access to the maximum bandwidth of the network, it does not know the current (and most importantly, future) network utilization; thus it has only a crude estimation of the real transfer duration.

5.2.3 Validation

It is challenging to validate the performance of different task schedulers in actual (non-simulated) task runtimes. Schedulers tend to be deeply integrated into their task runtime in order to be as performant as possible. That makes it difficult, or even infeasible, to replace the scheduling algorithm without also modifying large parts of the task runtime. Furthermore, some scheduling approaches might not even be compatible with the architecture of the runtime as a whole. For example, work-stealing schedulers perform a lot of communication between the server and the workers (potentially even between the workers themselves), and if the runtime does not implement the necessary infrastructure for facilitating these communication patterns, then implementing a work-stealing scheduler into such a runtime might amount to rewriting it from scratch.

In order to at least partially validate our simulation results, we have decided to use a modified version of the Dask [43] task runtime as a validation framework. Apart from validating results from the Estee simulations, we have also used this modified version of Dask to perform other experiments and benchmarks that are described in Chapter 6, which also depicts the architecture of Dask and our performed modifications in detail.

Dask is written in Python, which makes it relatively easy to modify and patch. It uses a work-stealing scheduler by default, and even though it is relatively deeply integrated within the Dask runtime, we were able to implement three simple alternative scheduling algorithms into it[3], which correspond as closely as possible to the *random*, *blevel* and *tlevel* schedulers from Estee. The default work-stealing scheduler was compared with our work-stealing implementation of the *ws* scheduler.

Apart from implementing new schedulers into Dask, there were several issues that we had to solve to make sure that the comparison between the simulated and the real environment is as fair and accurate as possible.

The absolute makespans of task graphs simulated by Estee and task graphs executed by Dask cannot be compared directly, because there are many aspects of the operating system, network, implementation of Dask itself and system noise that Estee can not fully simulate. Therefore, since the primary goal of our task scheduler experiments was to compare the relative performance of individual schedulers, we have decided to compare the relative makespans normalized to a reference scheduler (*blevel*), to test if the makespan ratios between the schedulers is similar in simulation and in real execution.

In the scheduler benchmarks, we have used many task graphs generated by the Estee task graph generator. However, it would not be possible to perfectly replicate task durations of these generated graphs in Dask. Therefore, we have approached this problem from the other direction. We have executed several task graphs in Dask, and recorded their execution traces, so that we would have a completely accurate representation of all the executed task durations and data object sizes, which we could then accurately replicate in the simulated Estee environment. The recorded task graphs will be described in Section 6.2 and they can also be found in [2]. We have executed these workflows with a $24x2$ cluster (24 cores on two nodes), which corresponds to two nodes of

[3]The modified version of Dask with these implemented schedulers can be found at `https://github.com/Kobzol/distributed/tree/simple-frame-sched`.

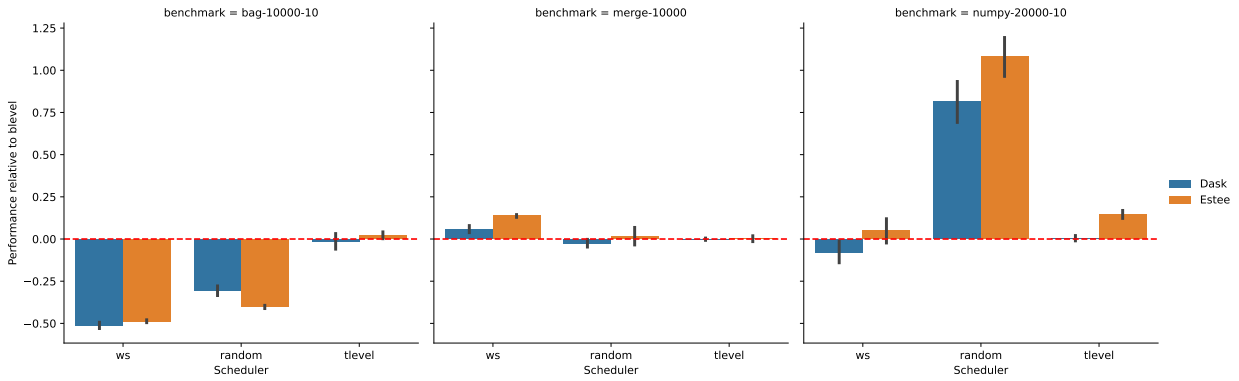


Figure 5.7: Scheduler performance relative to *blevel* in Dask and Estee

the Salomon [136] cluster, on which were the Dask workflows executed. Each actual execution and simulation was performed three times.

Figure 5.7 shows the results of the validation comparison for three selected task graphs[4]. The performance of each scheduler was normalized to the makespan of the *blevel* scheduler within the same environment (either Estee or Dask). Note that the relative ratios were centered around zero by subtracting 1 from them, to focus on the relative differences. For example, if a task graph execution had a makespan of 100 s with the *blevel* scheduler, but 110 s with the *ws* scheduler, the ratio of the *ws* scheduler would be 0.1. If the simulation was perfect, the two columns for each scheduler would have the same height.

The first chart shows a situation where changing the scheduler resulted in large changes in makespans, and Estee was able to simulate these changes relatively accurately, and reflect the results measured in Dask. The second chart demonstrates a situation where all schedulers produce similar makespans; therefore, in this case the scheduling algorithm does not seem to be that important. Estee was again able to estimate that the differences between schedulers will be small. In the third chart, we see that Estee was systematically overestimating the makespans of all three schedulers (with respect to the reference scheduler). The most important difference was in the *ws* scheduler, where the simulated result claims that it is slower than *blevel*, while in reality it was slightly faster. The work-stealing implementation in Dask is complex, and in this case it was able to outperform *blevel* in a way that Estee was not able to simulate.

To summarize the average error of the simulated results, we took the relative makespans of the individual schedulers w.r.t. the reference *bevel* scheduler, and calculated the difference between the executed and simulated relative makespan. The geometric mean of these differences across all measured benchmarks was 0.0347, which suggests that the differences between the execution and simulation were relatively small; the simulated makespans were usually within just a few percent off the actual makespan duration.

[4]Extended validation results can be found in [1].

Summary

We have implemented a set of well known scheduling heuristics, prepared a benchmark dataset containing task graphs of different types and scales and designed a simulation environment for task scheduler experimentation. We have conducted a series of reproducible benchmarks using that environment, in which we have analyzed the effect of network models, minimal scheduling delays, information modes and worker selection strategy on the behavior of the implemented schedulers, and also compared their relative performance.

Our attempts to implement existing scheduling algorithms in Estee and reproduce the results of previous scheduler benchmarks have shown that various implementation details which are often missing from the algorithm's description (like the precise worker selection strategy) can have a large effect on the final performance of the scheduler. Furthermore, we have been able to confirm our hypothesis that the naive network model used in several existing works can result in very inaccurate simulation results.

We have demonstrated that even a completely random scheduler can be competitive with other scheduling approaches for certain task graphs and cluster configurations. This supports the conclusion made in [68], where the authors have also observed that relatively simple scheduling algorithms can be competitive, and more complex algorithms are useful mostly for special situations and edge cases, where the simple heuristics might fail to produce reasonable results.

The Minimal Scheduling Delay had a smaller effect in our simulations than we have expected. This hints that it might be possible to reduce the scheduling overhead (by invoking the scheduler less often) without sacrificing schedule quality, but this will be highly dependent on the specific task runtime implementation.

The effect of different information modes turned out to be significant, although it is unclear whether schedulers can leverage the additional information about exact task durations when facing unpredictable network conditions.

All our benchmarks, source code and also experiment results and charts are available in an open and reproducible form [133, 134], which should facilitate reproducibility of our experiments.

The implemented Estee simulator enables simple prototyping of task schedulers and also allows simulating execution of various task graphs. We hope that it could have further potential to simplify the development and evaluation of novel task schedulers.

Through our experiments presented in this chapter, we have learned more about the effect of scheduling algorithms on the performance of task graphs. We have identified a performant baseline scheduling approach, work-stealing combined with the *b-level* heuristic, which we have further leveraged in work described in the following two chapters.

Although task scheduling is an important factor, it is not the only aspect that contributes to the overall overhead and performance of task runtimes. The following chapter will examine the runtime overhead and scalability of a state-of-the-art task runtime in more detail.

Chapter 6

Task runtime optimization

This chapter delves further into the performance aspects of task graph execution on HPC clusters. After performing the task scheduling experiments using Estee, our next goal was to find out how do other parts of task runtimes (other than the scheduler) affect task graph makespans, in order to provide guidelines for efficient design of task runtimes. We also wanted to validate some of the rather surprising results of our experiments, for example the competitiveness of the random scheduler, in a non-simulated setting. In order to do that, we moved from a simulator to a task runtime that executes task graphs on an actual distributed cluster.

As was already discussed in Chapter 4, there is a large number of existing task runtimes that are being used on HPC systems, which could be used as a test vehicle for our experiments. After evaluating several of them, we have decided to examine the Dask [43] task runtime, for the following reasons.

- It is very popular within the scientific community [137], and thus any insights into how it could be improved could benefit a wide range of users.
- It is implemented in Python, which makes it relatively easy to modify.
- It is quite versatile, as it allows executing arbitrary task graphs with dependencies and performing data object exchanges between workers.
- It uses a fairly standard distributed architecture with a centralized server that creates task schedules and assigns tasks to a set of distributed workers. This maps well to the cluster architecture used by Estee and also to HPC clusters in general.

In terms of task scheduling, Dask uses a work-stealing scheduler, which has been tuned extensively over several years to support various use-cases and to provide better scheduling performance. Yet it is unclear whether additional effort should be directed into improving the scheduler or if there are other bottlenecks which should be prioritized.

To answer that question, we have analyzed the runtime performance and the bottlenecks of Dask in *Runtime vs Scheduler: Analyzing Dask's Overheads*[1] [2]. This work provides the following contributions:

[1]Note that this line of research follows after the task scheduler analysis described previously in Chapter 5, even though it was published at an earlier date.

1. We have created a set of benchmarks containing diverse task graphs implemented in Dask. This benchmark set was then used to analyze Dask's performance in various HPC-inspired scenarios. We have evaluated the per-task-overhead and scalability of Dask and compared how different task scheduling algorithms affect its performance.

2. We demonstrate that even a naive (completely random) scheduling algorithm can be in some situations competitive with the sophisticated hand-tuned work-stealing scheduler used in Dask.

3. We provide RSDS, an alternative Dask server that is backwards-compatible with existing Dask programs and provides significant speed-up vs the baseline Dask server in various scenarios despite using a simpler task scheduler implementation.

Various descriptions of task graph benchmarks, Dask and RSDS used in this chapter were adapted from our publication [2].

I have collaborated on this work with Ada Böhm; we have both contributed to it equally. I have designed and performed all the experiments described in this chapter. Source code contribution statistics for RSDS can be found on GitHub[2].

6.1 Dask task runtime

Dask is a distributed task runtime implemented in Python that can parallelize and distribute Python programs. It offers various programming interfaces (APIs) that mimic the interfaces of popular Python packages. For example, *Dask DataFrame* copies the `pandas` [138] interface for table processing and database operations, *Dask Arrays* copies the `numpy` [139] interface for tensor computations and *Dask ML* copies the `scikit-learn` [140] interface for machine-learning. Thanks to this interface compatibility, existing Python programs leveraging these libraries can often be parallelized with Dask by changing only few lines of code.

This is demonstrated in Listings 6.1 and 6.2, which show two small Python programs that leverage the *Dask Arrays* and *Dask DataFrame* API, respectively. Notably, the only difference between these programs (which leverage Dask and thus can be parallelized) and a standard sequential version is the change of imports from `numpy` and `pandas` to Dask Python modules.

```
# import numpy as np
import dask.array as np

x = np.random.random((10000, 10000))
y = (x * 2 + x.T).mean(axis=1)
```

Listing 6.1: Example of a Python program that leverages the Dask Array API

[2] `https://github.com/it4innovations/rsds/graphs/contributors`

```python
# import pandas as pd
import dask.dataframe as pd

df = pd.read_csv("data.csv")

df2 = df[df.y > 0]
df3 = df2.groupby("name").x.std()
```

Listing 6.2: Example of a Python program that leverages the Dask DataFrame API

Programming model

Dask automatically transforms Python code leveraging these APIs into a task graph, which is then executed in parallel, possibly on multiple nodes of a distributed cluster. This enables almost transparent parallelization of sequential Python code. However, apart from these high-level interfaces, it is also possible to build a task graph manually, using the *Futures* interface, to define complex computational workflows.

The core computational abstraction of Dask is a *task graph*, which corresponds closely to the definition of task graphs that was defined in Chapter 3. Each task represents a single invocation of a Python function. The return value of the function forms its *output* (a data object), and the arguments of the function invocation define the *inputs* of the task.

One important aspect of the mapping between Python code and task graphs in Dask is the concept of *partitions*. It is a configuration parameter that essentially controls the granularity of tasks created by Dask out of Python code that uses its APIs. For example, a single line of Python code that performs a query over a `pandas`-like table (also called DataFrame) will eventually be converted to a set of tasks; each such task performs the query on a subset of the table's rows. The selected number of these tasks (or partitions) is crucial, since it determines how effectively the operation will be parallelized. Too few (large) tasks can cause computational resources to be underutilized, while too many (small) tasks can overwhelm the Dask runtime.

Architecture

Dask supports multiple computational backends that can execute the task graphs generated from Python code. The default backend is able to execute the task graph in a parallelized fashion on a local computer, but there is also a distributed backend called *dask/distributed*[3] (or simply *distributed*), which is able to execute task graphs on multiple nodes. Since this backend is most relevant for task graph execution on distributed and HPC clusters, our experiments focus solely on this backend, and any further reference to Dask in this text will assume that it uses the *distributed* backend.

In terms of architecture, Dask is composed of three main components: the *client*, the *server* and the *worker*. A single server and an arbitrary number of workers deployed together (e.g. on a local machine or a distributed system) form a Dask cluster.

[3] `https://distributed.dask.org`

The *client* is a user-facing library that offers various APIs used to define computations in Python that can be converted into a task graph. Once the user defines the computation, the client can connect to the Dask cluster (more specifically, to the server), submit the task graph, wait for it to be computed and then gather the results. The client can build the whole task graph eagerly on the user's machine and then send it to the server for processing; however, this can consume a lot of memory and network bandwidth if the task graph is large. For certain types of task graphs, clients are able to send a much smaller, compressed abstract representation of the task graph that is only expanded on the server lazily, which can help reduce memory and network transfer bottlenecks.

The *server* is the central component of the cluster, which communicates with the workers and clients through a network, usually a TCP/IP (Transmission Control Protocol/Internet Protocol) connection, handles client requests, coordinates task execution on workers and manages the whole Dask cluster. Its main duty is to process task graphs submitted by clients by assigning individual tasks to workers, in order to parallelize the execution of the submitted task graph and in turn efficiently utilize the whole cluster. It uses a sophisticated work-stealing scheduler that uses many heuristics, which have been tuned for many years. Some of them are described in the Dask manual[4].

The scheduler works in the following way: when a task becomes *ready*, i.e. all its dependencies are completed, it is immediately assigned to a worker according to a heuristic that tries to minimize the estimated start time of the task. This estimate is based primarily on any potential data transfers that would be incurred by moving the data objects of tasks between workers, and also the current occupancy of workers. When an imbalance occurs, the scheduler tries to steal tasks from overloaded workers and distribute them to underloaded workers. The scheduler also assigns priorities to tasks, which are used by workers to decide which tasks should be computed first.

The *worker* is a process which executes tasks (consisting of serialized Python functions and their arguments) that are assigned and sent to it by the server. Workers also communicate directly among themselves to exchange task outputs (data objects) that they require to execute a task. Tasks assigned to a worker are stored in a local queue; tasks are selected from it based on their priorities. Each worker can be configured to use multiple threads, some of which handle network (de)serialization and I/O while the rest are assigned for executing the tasks themselves. However, there is an important caveat that can limit the parallel execution of tasks within a worker, which is described below.

Bottlenecks

The primary bottlenecks that limit the efficiency of Dask are related to the programming language used for its implementation. All described components (server, worker and client), are implemented in Python, which has non-trivial consequences for its performance, because the Python language is not well-suited for implementing highly performant software. The most commonly used Python interpreter (CPython[5]) does not generally allow programmers to make optimal use of hardware in an easy way due to its indirect memory layout and automatic memory management that pervasively uses reference-counting.

[4] `https://distributed.dask.org/en/latest/work-stealing.html`
[5] `https://github.com/python/cpython`

Crucially, there is a specific quirk of this interpreter that can severely reduce the performance of task graph execution in Dask; specifically, it affects the performance of its workers. The CPython interpreter infamously uses a shared, per-process lock called GIL (Global Interpreter Lock), which synchronizes access to the internal data structures of the interpreter. This means that when a Python program is interpreted using CPython, only a single thread that executes Python code can make progress at the same time. It is still possible to achieve concurrency with multiple Python threads (e.g. by using blocking I/O, which releases the GIL while the thread is blocked), but not parallelism, under this design. There are some caveats to this, notably code written using native Python extensions (most commonly written in `C`, `C++` or `Rust`) can run truly in parallel with other Python threads, but only if it opts into releasing the global lock; this prevents the native code from interacting with the Python interpreter while it is running.

The GIL issue does not impact only Dask, of course. It is a pervasive issue across the whole Python ecosystem. There have been multiple attempts over the years to remove the GIL from CPython, but they have not been successful yet. The most recent attempt has progressed the furthest, and it has been accepted as a PEP (Python Enhancement Proposal) 703 [141], so it is possible that the GIL will eventually be removed from CPython. However, even if this proposal is adopted, it will probably take years before Python packages and programs will fully adapt to the change.

The presence of the GIL poses a challenge for Dask workers. Unless a task executed by a worker releases the GIL (which essentially means that the task either has to be implemented in native code or it needs to block on an I/O operation), it will block all other tasks from executing at the same time. Therefore, a single Dask worker can only execute at most one non-native and non-blocking task at once, which can potentially severely limit its throughput and the efficiency of task graph execution. It is worth noting that many commonly used data science and data analysis tasks executed with Dask will likely be implemented in native code (such as *numpy*, *pandas*, or their Dask equivalents). However, for tasks written in pure Python, the worker will essentially behave in a single-threaded fashion. This has been observed for example in [142], where several workflows did not benefit at all from multi-threaded Dask workers.

To alleviate this limitation, Dask workers can be executed in multiple instances on the same node, each running inside a separate process. In this configuration, each worker has its own copy of the CPython interpreter, and thus also its own copy of the GIL; therefore, tasks running inside one worker do not affect (and most importantly block) tasks running on other workers. However, this also means that certain overhead (a TCP/IP connection to the server, a worker entry in the scheduler, management of worker tasks) is multiplied by the number of spawned workers on each node. In certain cases, it is thus necessary to carefully balance the trade-off between too few workers (which can hamper task execution parallelism) and too many workers (which can reduce the overall efficiency of the Dask runtime).

6.2 Dask runtime overhead analysis

We have designed a series of experiments to evaluate the inner overhead of Dask and to find out which factors affect its runtime performance the most.

6.2.1 Benchmarks

To evaluate the runtime, we have prepared a diverse set of benchmarks that span from simple map-reduce aggregations to text processing workloads and table queries. The properties of the task graphs used in our experiments along with the Dask API that was used to create them are summarized in Appendix A (Table A.2).

Most of the task graphs are heavily inspired by programs from the Dask Examples repository[6]. The definitions of the benchmarks are available in a GitHub repository[7]. A short summary of the individual benchmarks is provided below.

merge-n creates n independent trivial tasks that are merged at the end (all of their outputs are used as input for a final merge task). This benchmark is designed to stress the scheduler and the server, because the individual tasks are very short (essentially they perform almost no work).

merge_slow-n-t is similar to `merge-n`, but with longer, t second tasks.

tree-n performs a tree reduction of 2^n numbers using a binary tree with height $n - 1$.

xarray-n calculates aggregations (mean, sum) on a three-dimensional grid of air temperatures [143]; n specifies size of grid partitions.

bag-n-p works with a dataset of n records in p partitions. It performs a Cartesian product, filtering and aggregations.

numpy-n-p transposes and aggregates a two-dimensional distributed `numpy` array using the *Arrays* interface. The array has size (n, n) and it is split into partitions of size $(n/p, n/p)$.

groupby-d-f-p works with a table with d days of records, each record is f time units apart, records are partitioned by p time units. It performs a groupby operation with an aggregation.

join-d-f-p uses the same table, but performs a self-join.

vectorizer-n-p uses Wordbatch[8], a text processing library, to compute hashed features of n reviews from a TripAdvisor dataset [144] split into p partitions.

wordbag-n-p uses the same dataset, but computes a full text processing pipeline with text normalization, spelling correction, word counting and feature extraction.

6.2.2 Benchmark configuration

Our experiments were performed on the Salomon supercomputer [136]. Each Salomon node has two sockets containing Intel Xeon E5-2680v3 with 12 cores clocked at 2.5 GHz (24 cores in total),

[6] `https://examples.dask.org`
[7] `https://github.com/it4innovations/rsds/blob/master/scripts/usecases.py`
[8] `https://github.com/anttttti/Wordbatch`

128 GiB of RAM clocked at 2133 MHz and no local disk. The interconnections between nodes use an InfiniBand FDR56 network with 7D enhanced hypercube topology.

Unless otherwise specified, by default we spawn 24 Dask workers per node, each using a single thread for task computations. We chose this setting because of the CPython GIL issue described earlier. Since our benchmarks are mostly compute-bound and not I/O-bound, a single worker cannot effectively use more than a single thread. Not even the popular `numpy` and `pandas` libraries used in our benchmarks are multi-threaded by default, which is also why Dask provides direct API support for their parallelization. Using the same number of workers as the available cores ensures that no more than a single task per core is executed at any given moment, to avoid oversubscription of the cores.

To validate our choice of this default configuration, we have benchmarked a configuration using a single worker with 24 threads per each node. We have found that it provides no benefit in comparison to a single worker with only one thread in any of our benchmarks.

For each of our experiments we state the number of used worker nodes which contain only the workers. We always use one additional node which runs both the client and the server. For our scaling experiments, we use 1 to 63 worker nodes (24 to 1512 Dask workers), for the rest of our experiments we use either 1 or 7 worker nodes (24 or 168 Dask workers). We have chosen these two cluster sizes to represent a small and a medium sized Dask cluster. The number of workers is fixed; it does not change during the computation.

We have executed each benchmark configuration five times (except for the scaling benchmarks, which were executed only twice to lower our computational budget) and averaged the results. We have measured the makespan as a duration between the initial task graph submission to the server and the processing of the final task graph output by the client. The whole cluster was reinitialized between each benchmark execution.

The following abbreviations are used in figures with benchmark results: *ws* marks the work-stealing scheduler and *random* represents the random scheduler.

6.2.3 Evaluation

We have designed three specific experiments that focus on Dask's scheduler, its inner overhead, and the effect of GIL on its performance.

Random scheduler

Our original goal of this experiment was to test how different schedulers affect the performance characteristics of Dask. However, it turned out that plugging a different scheduling algorithm into it is complicated, because it uses a work-stealing scheduler implementation that is firmly ingrained into its codebase across multiple places. There was not a single place where the scheduler could be swapped for a different implementation without affecting other parts of the runtime, unlike in Estee. Integrating a different complex scheduler into Dask would thus require making further changes to it, which could introduce bias stemming from arbitrary implementation decisions.

We have thus decided to implement perhaps one of the simplest schedulers possible, which does not require a complex state and which could be implemented relatively easily within Dask – a

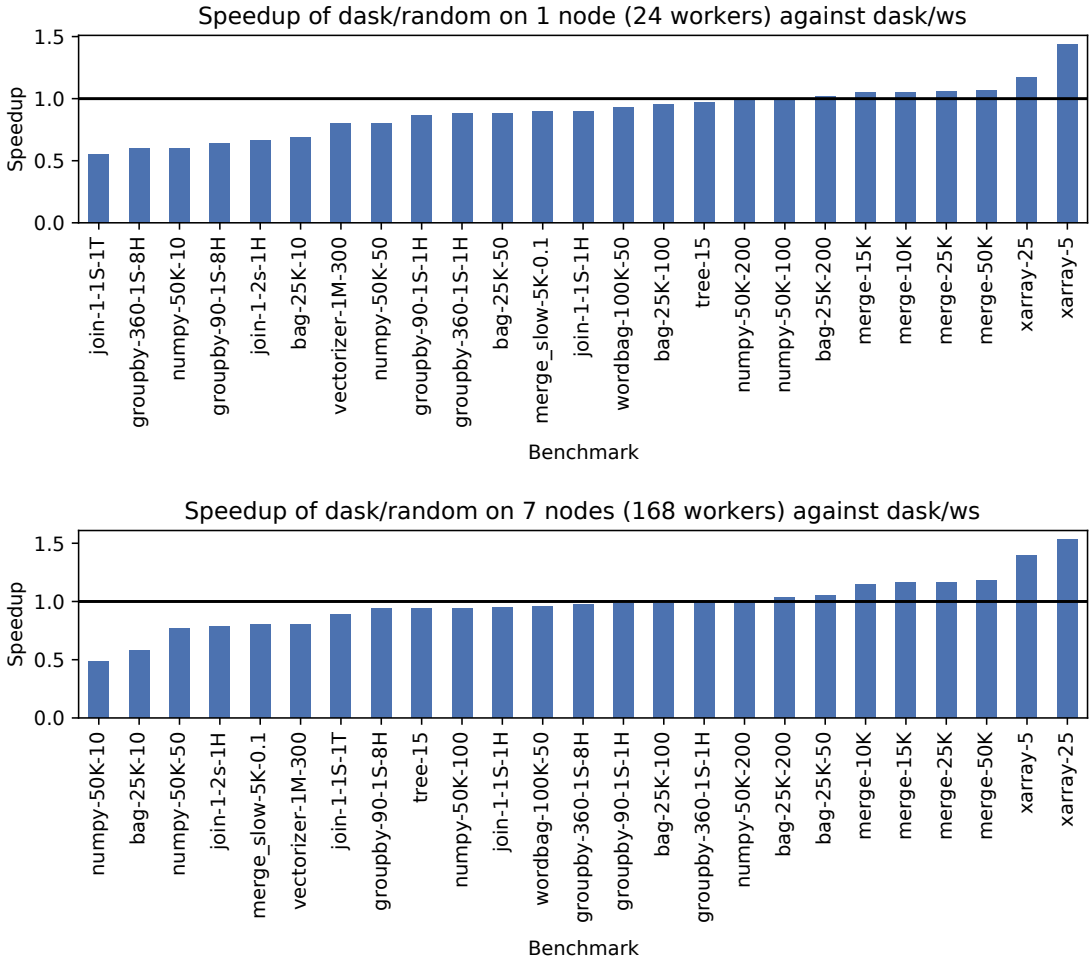


Figure 6.1: Speedup of Dask/random scheduler; Dask/ws is baseline.

fully random scheduler. This scheduler simply uses a PRNG engine to assign tasks to workers at random. Our Estee experiments have shown that a completely random scheduler could still achieve competitive performance in some cases, which was quite a surprising result to us. However, since these results were only tested in a simulated setting, it was interesting to examine how it would fare on an actual distributed cluster, while scheduling real-world Dask task graphs.

We can observe the results of benchmarks that compare Dask using its baseline work-stealing scheduler vs a completely random scheduler in Figure 6.1. Based on these results, it is clear that the random scheduler fares relatively well on both small and medium sized clusters. At worst, it produces a makespan twice as long, but overall it is close to the performance of the work-stealing scheduler, and in some cases it even outperforms it with a 1.4× speedup.

If we aggregate the results with a geometric mean, the random scheduler achieves 88% of the performance of the work-stealing scheduler with a single node and 95% with seven nodes. The performance of the random scheduler thus gets closer to the performance of work-stealing when more workers are used.

The fact that the random scheduler's performance improves with a larger number of nodes is not surprising. With more nodes, it is easier to fully saturate the potential parallelism contained in each task graph. Furthermore, a random scheduler produces less network traffic and has a much smaller computational cost on the server; as we will see in the next experiment, the work-stealing scheduler's computational cost increases notably when more workers are present.

Furthermore, for certain task graphs, a complex scheduling algorithm might not be needed, and a random schedule is sufficient to achieve reasonable results. However, all of that combined is still a rather weak explanation for why a random scheduler is so competitive with the work-stealing scheduler in Dask. In the following experiments, we will show that Dask has considerable runtime overhead, which might introduce a bottleneck that is more impactful than the effect of the used schedule. In Section 6.3, we will see that with a more efficient runtime and less server overhead, random schedules will become less competitive.

Overhead per task

We have seen that a random scheduler can be surprisingly competitive with the work-stealing scheduling implementation in Dask. In this experiment, we further examine this effect by estimating the inner overhead of Dask per each executed task. In order to isolate the effects of the specific executed tasks, as well as network communication and worker overhead, we have created a special implementation of the Dask worker that we label *zero worker*.

Zero worker is a minimal implementation of the Dask worker process, written in Rust. Its purpose is to simulate a worker with infinite computational speed, infinitely fast worker-to-worker transfers and zero additional overhead. It actually does not perform any real computation; when a task is assigned to a zero worker, it immediately returns a message that the task was finished. It also remembers a set of data-objects that would be placed on the worker in a normal computation. When a task requires a data object which is not in this list, the worker immediately sends a message to the server, claiming that the data object was placed on it – this simulates an infinitely fast download of data between workers.

No actual worker-to-worker communication is performed; zero workers respond to every data object fetch request with a small mocked constant data object. Such requests come from the server when a client asks for a data object, usually at the end of the computation. Since there is no worker-to-worker communication when the zero worker is used, fetch requests never come from other workers.

This idealized worker implementation helps us understand the fundamental overhead of the Dask runtime, independent of the specific tasks that are being executed. Note that even though all tasks in this mode are basically the same, the shape and size of the benchmarked task graph are still important, since they affect the performance of the scheduler and also the bookkeeping overhead of the runtime.

Using this special mode, we have evaluated the average runtime overhead per each task, which is calculated as the total makespan divided by the number of tasks in the executed task graph. Figure 6.2 shows the average overhead per task for the `merge` benchmark; we can see how the average overhead per each task (vertical axis) changes with an increasing number of tasks (horizontal axis).

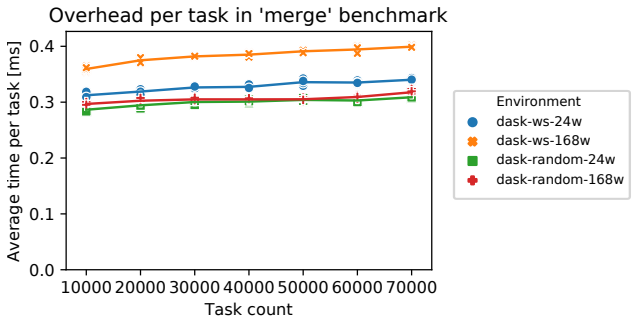


Figure 6.2: Overhead per task in Dask with an increasing number of tasks.

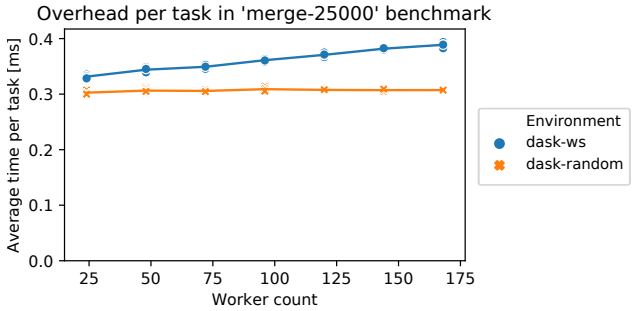


Figure 6.3: Overhead per task in Dask with an increasing number of workers.

We can observe that the overhead of the random scheduler is smaller than the overhead of the work-stealing scheduler, as expected. We can also see that the overhead per task increases with an increasing number of tasks for both schedulers. There is also a distinct increase of overhead between the configuration with 24 workers (one node) and with 168 workers (seven nodes) for the work-stealing scheduler.

This effect can be examined in more detail in Figure 6.3, which shows how the overhead increases when more workers are available. It is clear that the overhead of the work-stealing scheduler increases when more workers are added to the cluster, while the overhead of the random scheduler stays almost constant, independent of the number of workers.

The Dask manual states that "Each task suffers about 1ms of overhead. A small computation and a network roundtrip can complete in less than 10ms."[9]. Our experiment shows that the overhead is less than 1 ms for most of our benchmarks.

The overhead per task is an important property of the task runtime, since it determines the minimal duration of a task that is still viable for parallel execution with that runtime. If the duration of a task is similar or smaller than the overhead of the runtime itself, executing such task with the runtime will probably not yield any speedup. This is demonstrated in Figure 6.4, which shows the strong scaling of Dask on the *merge_slow* benchmark. The three charts demonstrate how it scales with tasks that take 1000 ms, 100 ms and 10 ms, respectively. With tasks that take one second, Dask is able to scale relatively well up to 1512 workers. However, when tasks take one

[9] `https://distributed.dask.org/en/latest`

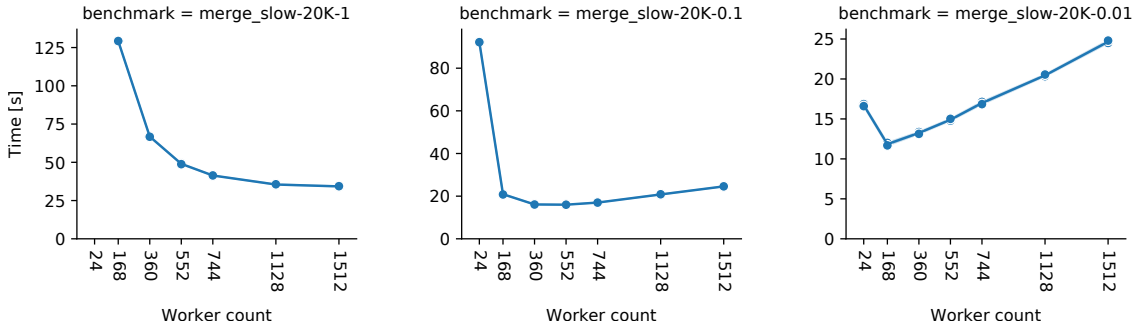


Figure 6.4: Strong scaling of Dask with different task durations (1000 ms, 100 ms and 10 ms).

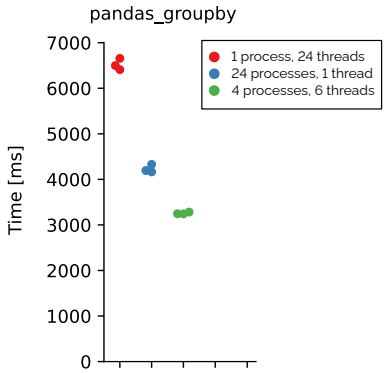


Figure 6.5: Effect of GIL on the performance of the *pandas_groupby* benchmark

tenth of the time, it only scales up to approximately 360 workers, and when tasks take only 10 ms, Dask scales to approx. 168 workers (7 nodes); adding more workers makes the makespan longer.

The results of this experiment indicate that the general runtime overhead of Dask mainly grows with an increasing number of tasks, no matter which scheduler is used. On the other hand, overhead of the work-stealing scheduler grows primarily with the number of workers. This is consistent with the results of the zero worker experiment presented earlier. They also show that the minimum duration of tasks executed by Dask should be taken into account in order to avoid introducing too much runtime overhead.

The fact that Dask might struggle with a larger number of workers would not be an issue on its own, as every task runtime will have its scaling limit. However, due to the already mentioned effect of GIL, Dask users might be forced to use more workers than would be otherwise necessary to achieve reasonable performance. We will examine this in the following experiment.

The effect of GIL

As noted earlier, the GIL can have a non-trivial effect on the performance of Python programs. This can also be seen in Dask, where the configuration of workers might need to be tuned in order to achieve optimal performance.

Figure 6.5 demonstrates the effect of GIL on the *pandas_groupby* benchmark. We have executed the same benchmark using three Dask worker configurations on a single node. In the default

configuration, with a single worker that uses 24 threads (one for each core), the pipeline finishes in approximately 6.5 seconds. If we instead create a single worker per each core (24 processes, each with a single thread), the performance improves significantly, by approximately 35%. This makes it clear that the GIL is a bottleneck in this case, and more Dask workers are needed to improve performance by enabling parallel task execution.

However, it is not so simple as to always use a single Dask worker per core. As we have discovered with our earlier benchmarks, more workers introduce non-negligible overhead for the Dask server. This can be seen from the result of the third configuration, which uses 4 Dask workers (processes), each leveraging 6 threads. This configuration actually achieves the best performance out of the three tested configurations. It is thus clear that for some Dask workflows, users might need to carefully tune the configuration of workers to achieve optimal performance.

Summary

Our experiments have shown that Dask does not scale optimally to a large number of workers and it might require manual tuning of worker configuration to achieve optimal performance. This led us to the idea of improving the implementation of the Dask server in order to reduce its overhead in HPC scenarios.

6.3 RSDS task runtime

In order to examine how much the performance of Dask could be improved if we were able to reduce its runtime overhead, we have developed RSDS (Rust Dask server), an open-source drop-in replacement for the Dask server [145]. We had the following goals for its design and implementation:

- Keep backwards compatibility with existing Dask programs, so that it can be used to speed up existing workflows. This also enables us to compare the performance of RSDS and Dask on the benchmarks described in the previous section.
- Design an efficient runtime that could scale to HPC use-cases, to find a baseline for how much fast Dask could become if its overhead was reduced.
- Use a modular architecture that would enable easily replacing the implementation of the scheduler, to enable easier experimentation with scheduling algorithms on non-simulated distributed clusters.

Architecture

RSDS is implemented in the Rust programming language, which has a minimal runtime and does not dictate automatic memory management. This reduces the ubiquitous overhead of reference counting and data indirection present in Python. It also has direct support for asynchronous I/O and provides strong memory safety guarantees. Therefore, it is well-suited for writing distributed applications.

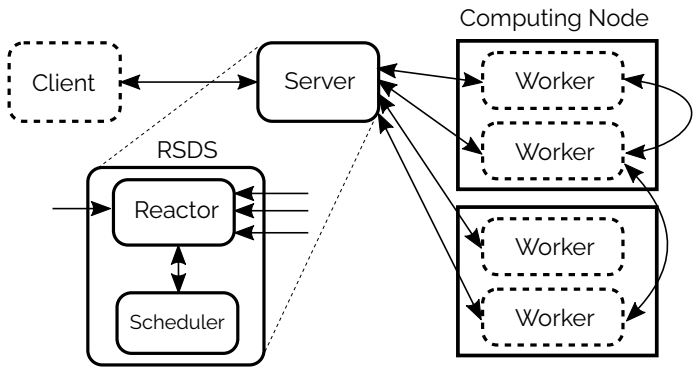


Figure 6.6: Architecture of RSDS
(Dask components are dashed)

The architecture of RSDS is shown in Figure 6.6. It reuses the client and worker components from Dask, but replaces its server (and thus also the scheduler). The main architectural difference between the two implementations is that RSDS separates the server into two parts: *reactor* and *scheduler*. The reactor manages worker and client connections, maintains bookkeeping information and translates scheduler assignments into Dask network messages which are then sent to the workers.

The scheduler is a process which receives a task graph and outputs assignments of tasks to workers. It does not know about worker network connections, Dask network messages or any other bookkeeping that is not relevant to scheduling. Since it is isolated, it can be easily swapped, and thus it is trivial to experiment with different schedulers in RSDS. Another benefit is that we can easily run the scheduler in a separate thread to enable concurrent scheduling and runtime management. This is possible because the scheduler does not share any data structures with the reactor. A disadvantage of this scheme is that both the reactor and the scheduler need to maintain their own copies of submitted task graphs, which increases their memory usage. However, we have not found this to be an issue in practice.

Same as Dask, the RSDS scheduler also ensures that task dependencies and resource requirements are always satisfied and that worker resources are not being oversubscribed, as defined by Definition 4 and Definition 5.

Compatibility with Dask

Dask components communicate together using a custom communication protocol that contains dozens of different message types. Messages are represented as dictionaries, where a single field specifies the type of the message and the remaining fields contain parameters specific to the given message type.

Dask splits messages into multiple parts called *frames* and serializes each frame with the MessagePack[10] binary serialization format. The messages are split mostly for performance reasons. If a client sends large binary data to a worker (or vice-versa), the server can deserialize only the first

[10] `https://msgpack.org`

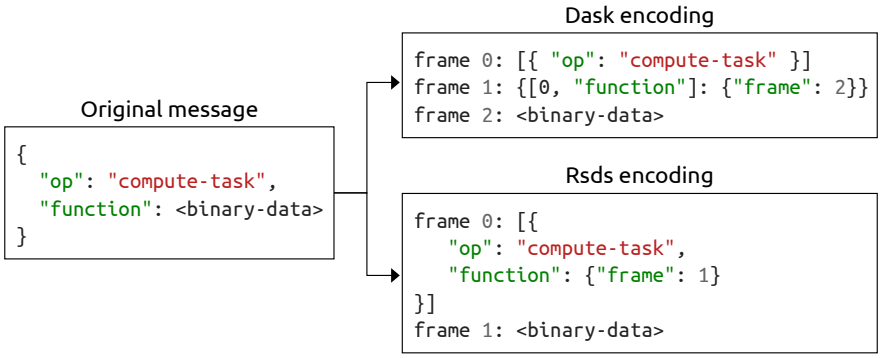


Figure 6.7: Dask message encoding

frame containing the message metadata, and it then does not need to further process the binary data; it merely forwards it to the correct destination (worker or client). The protocol is designed in such a way that any Python specific objects sent within the protocol are fully opaque to the server. For example, the server does not need to understand Python function definitions; it only forwards them from clients to workers, where they are deserialized and executed. This makes it possible to implement the central server in a different language than Python.

To retain compatibility with existing Dask programs, RSDS must understand this protocol and be able to both read and also generate its messages. This is relatively challenging, for two reasons. The first reason is that the protocol is unfortunately mostly undocumented[11], and its exact definition depends solely on Dask code that creates the protocol messages ad-hoc in many different source code locations. Therefore, we had to reverse-engineer its functionality.

The second reason is that the frame encoding scheme of Dask makes heavy use of the fact that Python is a very flexible and dynamic language. The specific rules of this encoding grew organically over time and they have become rather complicated and dynamic, as Dask can split messages into frames almost arbitrarily. Specifically, it sometimes extracts values out of arbitrary array indices or dictionary keys into a separate frame during serialization, and then puts them back into the original place during deserialization. The encoded frame metadata then contains a series of array index and dictionary key accessors. For example, `[0, "attr1", 1]: <data>` tells the deserializer to put the specified data into the second element of an array located in the `"attr1"` key of a dictionary that is located at the first element of the input array contained within the frame. This dynamic frame (de)fragmentation is relatively straightforward to perform in a dynamic language like Python, but it becomes incredibly difficult to handle in a statically-typed language with a strict type system, such as Rust.

An additional issue is that Dask sometimes uses completely different frame encoding schemes for the same message type, which further complicates frame deserialization. In order to overcome these challenges, we made a small change to the Dask protocol, so that it is feasible to implement the server in statically typed languages that make it difficult to parse the flexible messages generated by Dask clients and workers.

[11] `https://github.com/dask/distributed/issues/3357`

Our modification changes the dynamic encoding of Dask messages into frames in a way that always keeps the original message structure, so that it is easier to deserialize. This modification is described in Figure 6.7, which depicts a simplified scheme of a Dask message and its encoding in Dask and in RSDS[12]. On the left, we can see the message (dictionary) that we want to serialize, which contains two fields (`op` and `function`). The original Dask encoding, shown in the top right corner, would fragment this dictionary by removing its `function` field and moving it to a separate frame. During deserialization, the field would have to be put back into the original message, transmitted in the first frame. Our modified encoding, shown in the bottom right corner, keeps the original message structure instead, and replaces the field that has to be transferred in a separate frame with a placeholder. During deserialization, the placeholder is replaced with the contents of the second frame, which is relatively easy to implement in a statically typed language, because it avoids the need to dynamically change the message structure during deserialization.

This protocol modification only changes low-level message handling, and it is thus fully transparent to the rest of the code. It is also relatively small; it spans less than 100 modified lines of Dask source code. Crucially, it has no effect on the functionality of clients, workers and the server, so it does not require any modifications in Dask user programs. Our modified version of Dask is open-source and available online[13]. Note that all evaluations presented in this chapter use the modification described above. We have benchmarked this modification and found that there are no performance differences with respect to the original Dask message encoding.

It should be noted that RSDS does not implement all Dask message types. Some of them are rarely used and their implementation would be highly complex for a minimal gain. Others cannot be implemented in a straightforward way in a different language than Python. For example, Dask contains an API that allows clients to run a Python function on the server; this functionality does not have a direct counterpart in a server implemented in a different programming language. However, the fact that RSDS is able to execute all the benchmarks described in Section 6.2 demonstrates that it supports a wide variety of Dask programs.

Schedulers

We have implemented two schedulers in RSDS in order to allow us to compare its performance to Dask – a work-stealing scheduler and a completely random scheduler.

Even though it was not possible to exactly replicate the work-stealing implementation used in Dask, since it is affected by a very large number of implementation choices and details, our implementation is heavily inspired by its design. However, it is also deliberately kept simple, to avoid the need to perform extensive hand-tuning of constants and parameters within the scheduler. Some of the heuristics used by Dask were changed, simplified, or dropped in our implementation. For example, RSDS does not estimate average task durations and does not use any network bandwidth estimates. The work-stealing scheduler uses the *b-level* heuristic, as we have found that it provides reasonable performance in our experiments described in the previous chapter.

[12]Note that the message and the encoding is depicted using a JSON (JavaScript Object Notation) notation for improved readability. In reality, the data is encoded using MessagePack.

[13]`https://github.com/kobzol/distributed/tree/simplified-encoding`

The RSDS work-stealing scheduler works as follows: when a task becomes ready (i.e. all its dependencies are already finished), it is immediately assigned to a worker. The scheduler chooses a worker where the task may be executed with minimal data transfer costs, while it deliberately ignores the load of the worker. The load is ignored to speed up the decision in optimistic situations when there are enough tasks to keep all workers busy. When this is not the case, it is solved by balancing, which is described below.

When a new task is scheduled or when a task is finished, the scheduler checks if there are workers that are underloaded. In such case, balancing is performed; the scheduler reschedules tasks from workers with a sufficient number of tasks to underloaded workers. The scheduler simply passes the new scheduling state to the reactor, which performs the complex rescheduling logic. It tries to retract rescheduled tasks from their originally assigned workers. If retraction succeeds, the task is scheduled to the newly assigned worker. When the retraction fails because a task was already running or has been finished, the scheduler is notified; it will then initiate balancing again if necessary.

For computing transfer costs, we use a heuristic that takes into account inputs that are already present on the worker's node, and also inputs that will eventually be present because they are in transit or they are depended upon by another task assigned to the same worker. The transfer cost is estimated to be smaller for data transfers between workers residing on the same node.

Our random scheduler mirrors the random scheduler implementation that we have added to Dask – it assigns a random worker to each task as soon as the task arrives to the server, using a uniform random distribution. It ignores any other scheduling mechanisms, such as work-stealing.

6.4 Performance comparison of Dask and RSDS

We have performed a set of experiments to evaluate how performant is RSDS in comparison to Dask. The experiments were executed using the same hardware and benchmarks that were used for the previous Dask experiments described in Section 6.2. Our experiments are structured in the same way as the Dask experiments; they focus on the comparison of the two schedulers, the number of workers that each server implementation can scale to, and the inner overhead per task. Unless stated otherwise, all experiments use the original Dask client and worker implementations with the previously described protocol modification.

Server comparison

In the first experiment we compare the efficiency of the RSDS and Dask server implementations on a diverse set of benchmarks, using both the work-stealing and the random scheduler. The results for the work-stealing schedulers are shown in Figure 6.8. The data confirms our expectation that reducing the overhead of the server can help improve the makespan of executed task graphs by a non-trivial amount. Even though RSDS uses a much simpler work-stealing scheduler, its more efficient runtime achieves a higher performance in most cases. This effect is accentuated for a larger cluster with more workers.

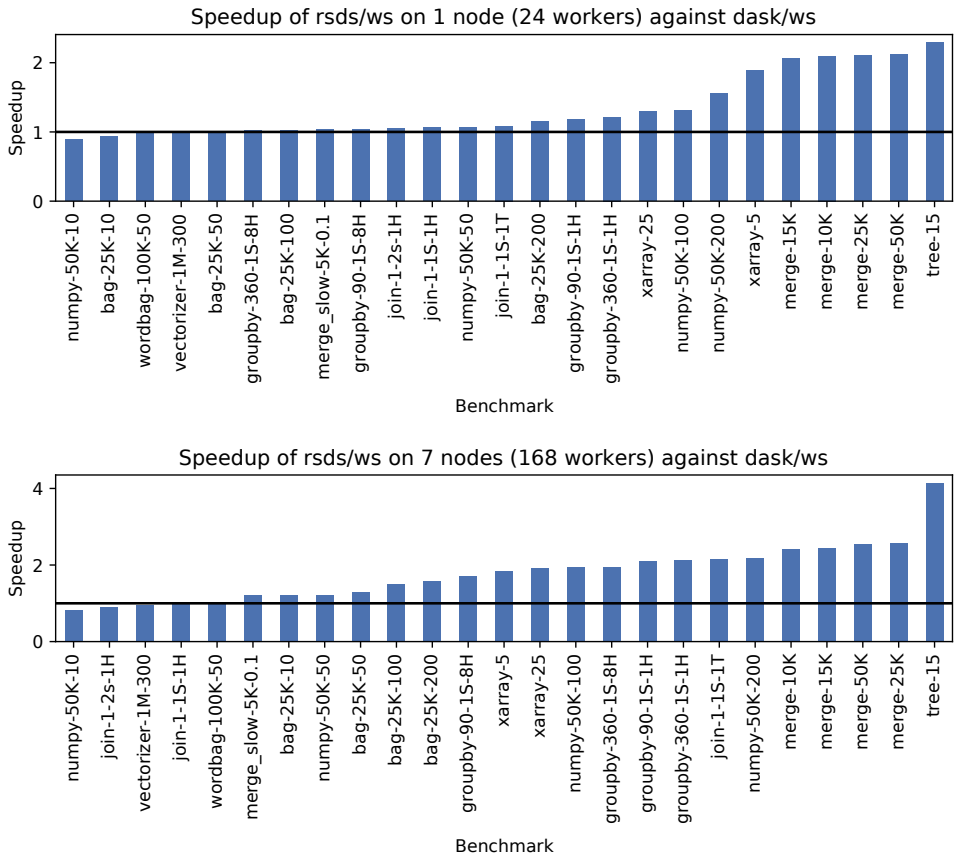


Figure 6.8: Speedup of RSDS/ws scheduler; baseline is Dask/ws.

Since the work-stealing implementations of RSDS and Dask are not exactly identical, the result above could also be explained by a difference in scheduling, rather than reduced runtime overhead. To eliminate that possibility, we also performed an experiment where we compare RSDS using a random scheduler with Dask using its sophisticated work-stealing scheduler. The results of this experiment can be seen in Figure 6.9. They confirm that the speedup is not caused by RSDS having a better scheduler, because even with a completely random schedule, it is still able to outperform Dask. This serves as evidence that the improved performance of RSDS with work-stealing is caused by better runtime efficiency and not by better schedules.

The results of this experiment are summarized in Table 6.1, which shows the geometrical mean of speedup of the tested RSDS configurations over a baseline using the Dask server with the work-stealing scheduler. The table also contains the results of the Dask server with a random scheduler, to provide a more complete picture. Even with the random scheduler, RSDS is on average faster than Dask using work-stealing. With more workers, this effect is further increased.

In Section 6.2, it was noted that the random scheduler used in Dask was relatively competitive with the work-stealing scheduler implementation. Our hypothesis was that this phenomenon is reinforced by the runtime overhead of Dask. To test this hypothesis, we have also compared the

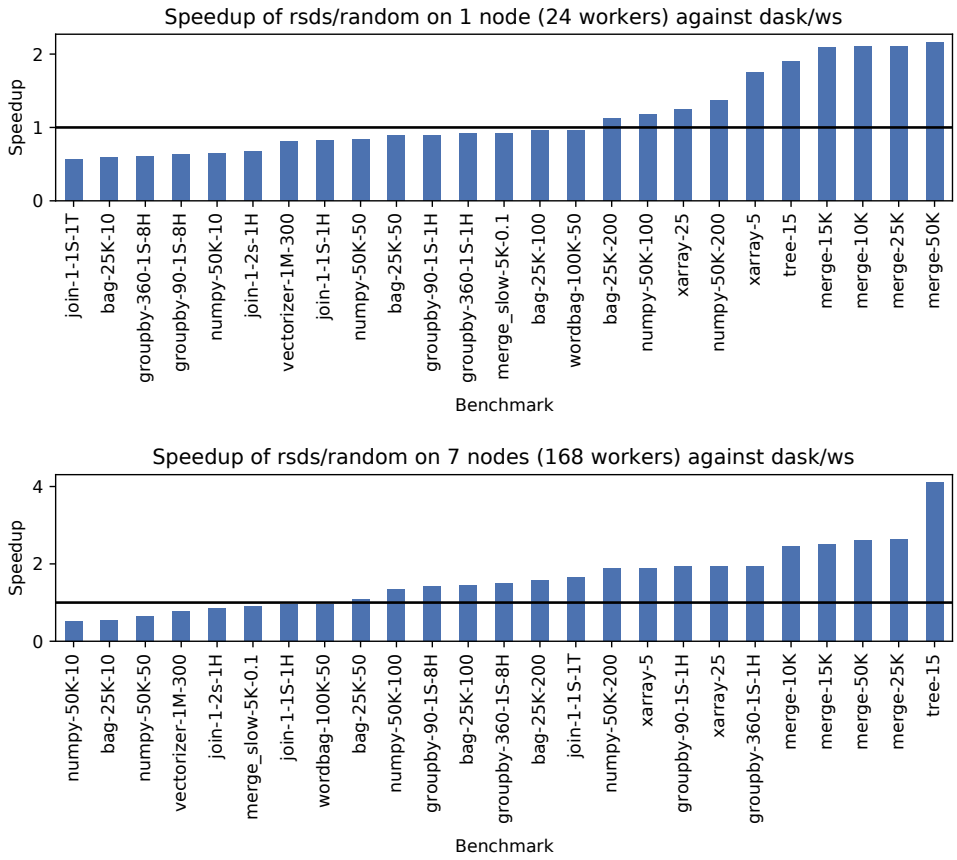


Figure 6.9: Speedup of RSDS/random scheduler; baseline is Dask/ws.

Server	Scheduler	Node count	Worker count	Speedup
dask	random	1	24	0.88x
rsds	random	1	24	1.04x
rsds	ws	1	24	1.28x
dask	random	7	168	0.95x
rsds	random	7	168	1.41x
rsds	ws	7	168	1.66x

Table 6.1: Geometric mean of speedup over the Dask/ws baseline

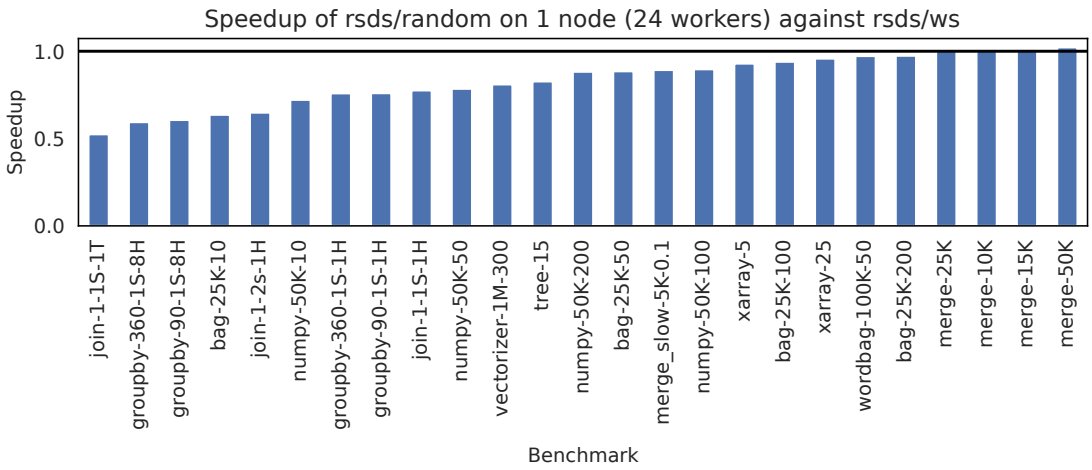


Figure 6.10: Speedup of RSDS/random scheduler; baseline is RSDS/ws.

Server	Scheduler	Baseline	Node count	Worker count	Speedup
dask	random	dask/ws	1	24	0.88x
rsds	random	rsds/ws	1	24	0.82x
dask	random	dask/ws	7	168	0.95x
rsds	random	rsds/ws	7	168	0.85x

Table 6.2: Geometric mean of speedup for random schedulers

performance of the RSDS random scheduler with its work-stealing implementation. The results can be observed in Figure 6.10. We can see that in the case of RSDS, the random scheduler is less competitive than it was in Dask (cf. Figure 6.1 and Figure 6.10) and it does not significantly outperform work-stealing in any of the benchmarked cases. This suggests that with a more reasonable runtime overhead of the server, a work-stealing scheduler is strictly better than a random scheduler, which is a much more intuitive result.

To provide a clearer comparison of the random scheduler results, Table 6.2 shows the geometric mean of speedup of the random scheduler, compared to a work-stealing scheduler baseline of a corresponding server implementation. It is clear from the results that in RSDS the random scheduler is less performant (relative to its work-stealing counterpart) than in Dask.

Scaling comparison

One of the primary motivations for creating RSDS was to improve the performance of Dask workflows in HPC scenarios. In these use-cases, it is important for the runtime to scale to a large number of workers that can be provided by an HPC cluster. We have designed an experiment which tests the strong scaling of both server implementations on several cluster sizes, ranging from 1 node (24 workers) to 63 nodes (1512 workers). The default work-stealing scheduler was used for both server implementations.

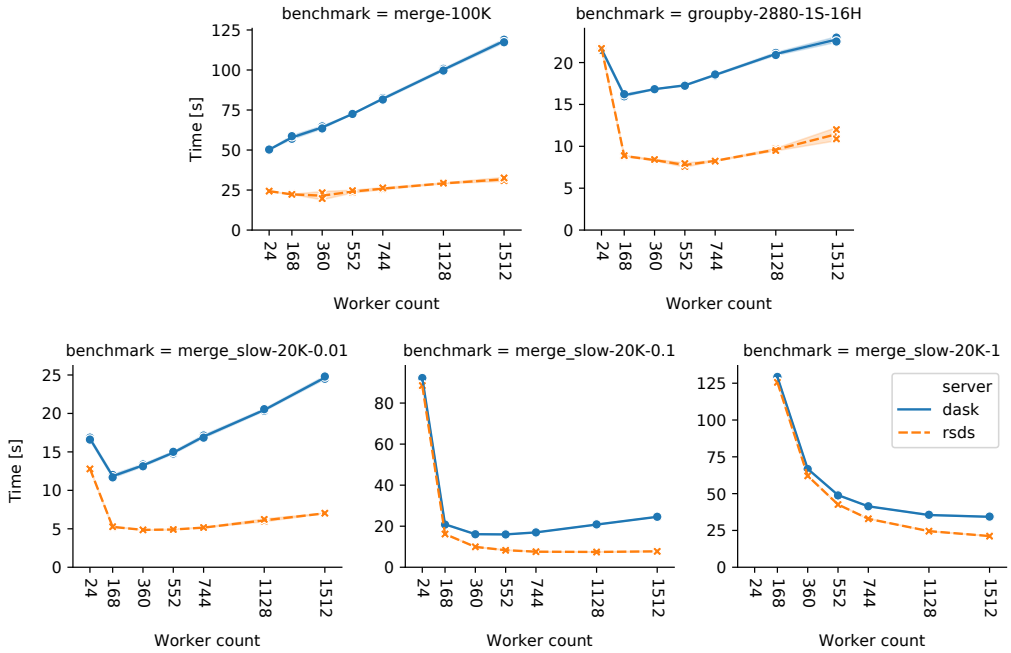


Figure 6.11: Strong scaling of RSDS vs Dask on selected task graphs

The results of this experiment are shown in Figure 6.11. The first examined task graph is `merge-100K`, which executes a hundred thousand trivial, almost instant tasks. It is an adversarial case for a scheduler, as the tasks are short and thus the overhead of scheduling and network transfers will overcome most parallelism gains. Therefore, increasing the number of workers will probably not provide a large speedup for this task graph. However, it should ideally not slow down the computation to a large extent. We can see that RSDS scales up to 15 nodes (360 workers). This is due to the fact that the cost associated with worker management and work-stealing increases with an increasing number of workers, and from some point it starts to dominate, because the tasks are too short. However, Dask fares much worse. It is twice as slow when compared to RSDS with a single worker node, but four times slower with 63 nodes (1512 workers). Here we can see that the inner overhead of Dask adds up, and its performance is reduced significantly with each additional worker node. On the other hand, with RSDS, the total makespan stays relatively constant, even after it stops scaling.

The next tested benchmark was `groupby-2880-1S-16H`, which computes an analysis of table data using a task graph automatically generated by Dask using the `DataFrame` API. This task graph provides many opportunities for parallelization, as the individual tasks work on a subset of rows and thus have more computational density compared to the `merge` task graph. However, Table A.2 (in Appendix A) shows that the average computation time is still only around 10 ms, while the average task output is 1 MiB. This benchmark thus produces considerable network traffic. While both Dask and RSDS have identical performance with a single worker node, Dask stops scaling at 7 nodes; with more nodes its performance degrades and eventually becomes slower than the single node case. RSDS scales up to 23 worker nodes, hence it is able to utilize three times more workers.

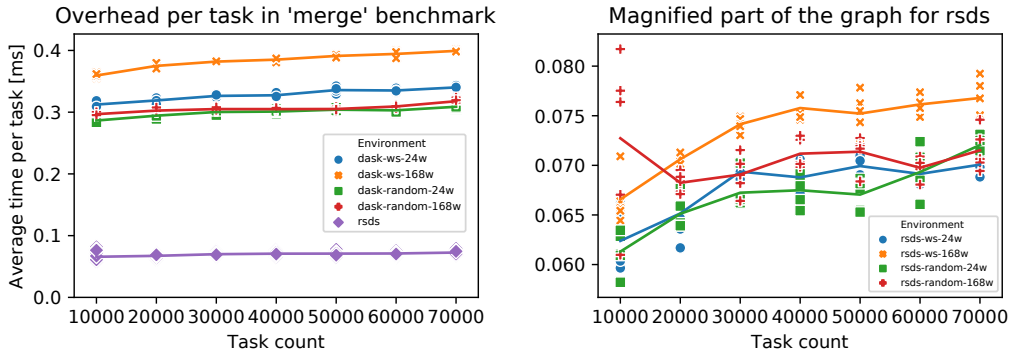


Figure 6.12: Overhead per task for RSDS and Dask with an increasing number of tasks.

With more worker nodes, the performance of RSDS also degrades, as the network communication caused by task output transfers and work-stealing messages starts to dominate the overall execution time. In this case, Dask and RSDS follow a very similar scaling pattern; however, in absolute terms, the makespans are twice as short with RSDS, and it does not become slower with more workers than with just a single worker.

The third examined task graph is `merge_slow-20K`, which executes twenty thousand tasks, where each task has a fixed duration, specified by a parameter (note that the `merge` and `merge_slow` benchmarks have the exact same task graph shape, the only difference is the duration of each task). We have benchmarked three variants of this task graph, with 0.01, 0.1 and 1 second tasks, same as for the previous similar experiment that we have performed with Dask only. This gives us a better idea of the task granularity required for Dask and RSDS to scale effectively. With 10 millisecond tasks, Dask scales to 7 workers; further its performance follows a similar shape as for `merge-100K`. RSDS stops scaling at 15 nodes, then its performance drops slightly with more added nodes. With 100 millisecond tasks, RSDS is able to scale up to 47 worker nodes (1128 workers); from that point on its performance stagnates. Dask scales only up to 23 worker nodes, then the makespan again starts to increase when additional workers are added. For the last task graph, with one second tasks, both RSDS and Dask scale up to 63 nodes (1512 workers). However, RSDS is consistently faster on all cluster sizes, and its performance with respect to Dask increases with added worker nodes; it is 1.03× faster than Dask with 7 nodes and 1.6× faster with 63 nodes.

In general, RSDS is able to scale to a larger number of workers than Dask, thanks to its reduced runtime overhead. Furthermore, it is also able to keep its performance relatively steady with an increasing number of workers after it stops scaling, even when executing very short tasks.

Overhead per task

In Section 6.2.3, we have seen that the average overhead of Dask for each task is approximately 0.3 ms. In this experiment, we have measured the per-task overhead of RSDS, so that we could qualify how much its overhead was reduced, relative to the previously measured baseline. All the benchmarks in this experiment were performed with the *zero worker* implementation.

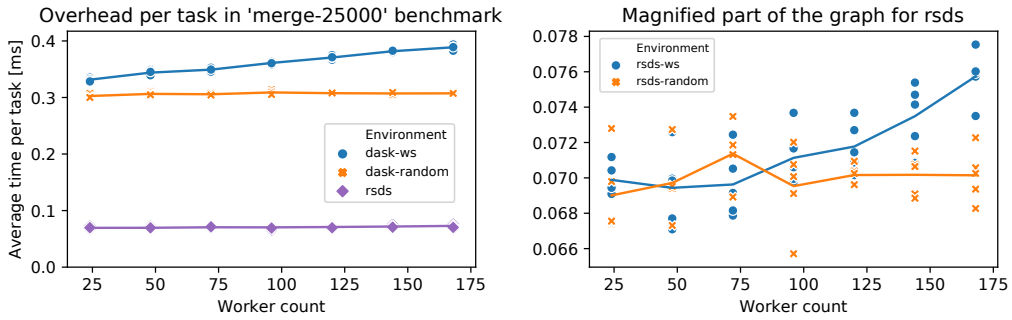


Figure 6.13: Overhead per task for RSDS and Dask with an increasing number of workers.

We have performed the same experiment as with Dask, to calculate the per-task overhead on the *merge* benchmark. Figure 6.12 shows how the per-task overhead changes for larger task graphs. From the chart on the left side, we can see that the per-task overhead of RSDS is approximately 3–4 times smaller than the overhead of Dask. From this chart, it looks like its overhead stays constant with an increasing number of tasks, yet when we zoom in on the section containing RSDS results (which we can see in the chart on the right side), we can observe a similar pattern that we see for Dask – the overhead of the work-stealing scheduler increases slightly with more tasks and the work-stealing scheduler generally has a larger overhead than the random scheduler. However, it happens on a much smaller absolute scale than with Dask.

Figure 6.13 depicts the results of per-task overhead with an increasing number of workers. The results are almost identical; the overhead of RSDS is several times smaller than with Dask. The overhead of the work-stealing scheduler also slightly increases with more added workers, although the increase is so small that it is almost negligible, at least for the evaluated number of workers.

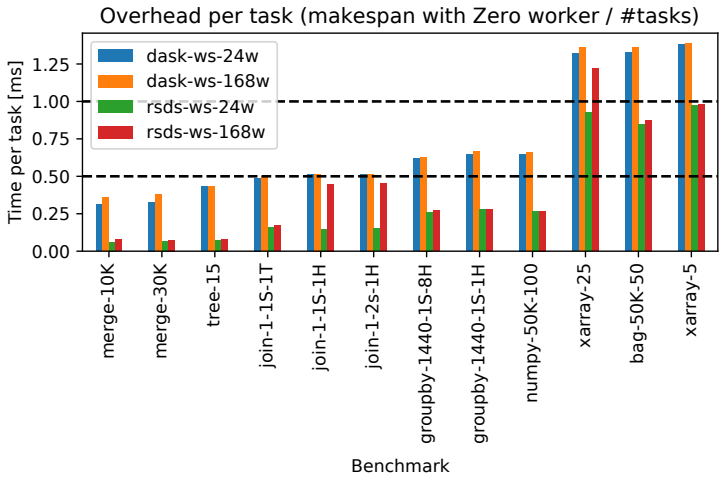


Figure 6.14: Overhead per task for various cluster sizes and benchmarks

In order to confirm that the per-task overhead results from the *merge* benchmark generalize to other benchmarks, we have also performed a similar experiment for several other task graphs. The

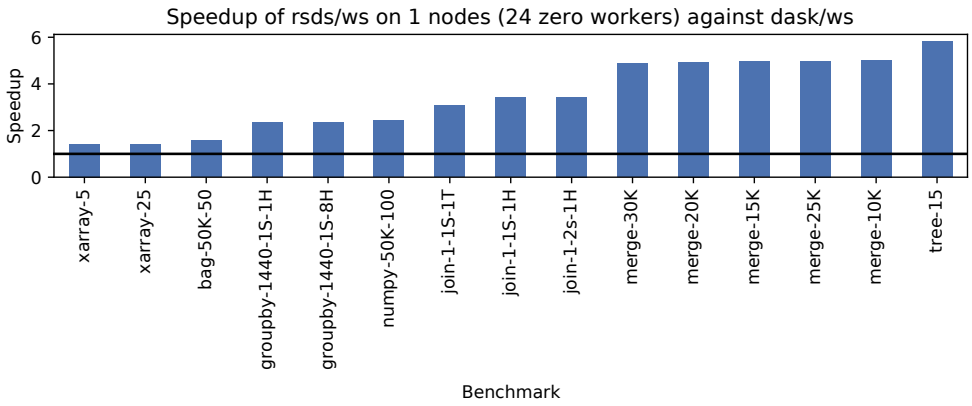


Figure 6.15: Speedup of RSDS/ws over Dask/ws with the zero worker implementation

results of this experiment are shown in Figure 6.14. We can see that the overhead for the *merge* benchmark is on the lower end of the spectrum; it reaches up to 1.25 ms for some other benchmarks. The results show that the per-task overhead of RSDS is always smaller than the overhead of Dask, although the difference is not that large for the *xarray* and *bag* benchmarks.

To evaluate the end-to-end effect of overhead on task graph makespan, we have executed several benchmarks with both Dask and RSDS (still using the *zero worker* implementation) and compared their overall makespans. Note that we could not execute all of our benchmarks with the zero worker mode, since some of the task graphs depend on the computed contents of data objects generated by tasks. Because the zero worker implementation does not actually execute tasks, it is unable to generate the correct data.

The results of this experiment can be found in Figure 6.15. For all evaluated benchmarks, RSDS produced shorter makespans than Dask; in some cases it was up to six times faster. This speedup is larger than with the standard worker implementation, which shows that if the overhead of the worker was reduced, RSDS could further improve its performance advantage over Dask. In other words, RSDS would benefit more from a faster worker implementation than the Dask server could.

All these experiments have shown that RSDS exhibits similar performance behavior and scaling patterns as Dask, which is not surprising given that it reuses its client and worker implementations. However, the most important result is that in absolute terms, it tends to be faster than Dask across various task graphs and cluster sizes, even though it uses a much simpler scheduling algorithm. This suggests that optimizing the runtime overhead of the Dask server is a more worthwhile effort than improving its scheduling algorithm.

The overall overhead present in the whole RSDS cluster could be further reduced by also implementing the worker (or even the client) in a language other than Python; however, that would probably make it challenging to keep backwards compatibility with existing Dask programs.

Summary

We have analyzed the performance of the Dask task runtime on a set of diverse benchmarks in an HPC setting. Our analysis has demonstrated that Dask is heavily bottlenecked by the runtime overhead of its Python implementation and by other aspects related to the usage of Python, notably the presence of the GIL. This suggests that further optimizations should focus mainly on the overhead of its server rather than its scheduler, as even a completely random scheduler is able to achieve reasonable performance in certain cases, because scheduling is not the primary bottleneck.

To improve the performance of Dask workflows, we have developed RSDS, an open-source drop-in replacement for the Dask server. 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 original Dask protocol, which means that it is backwards-compatible with existing Dask programs and can be used to speed them up.

We have performed a series of experiments that compared the performance of RSDS vs Dask. The results of our experiments indicate that optimizing the runtime is definitely a worthy effort to pursue, as RSDS has been able to outperform Dask in various scenarios even though it uses a much simpler work-stealing scheduling algorithm.

The RSDS task scheduler and parts of its runtime that were not Dask-specific have been used as a basis for the implementation of the HyperQueue task runtime, which is described in the following chapter.

Chapter 7

Task graph meta-scheduling

The previous two chapters focused primarily on the performance aspects of task graph execution, by examining various task scheduling algorithms and also bottlenecks that can limit the scalability of existing task runtimes. However, while performance is naturally crucial for HPC, it should not be the only factor to focus on. Chapter 4 discussed several other challenges affecting HPC task graphs that are related to the second main focus of this thesis, namely the ergonomics of task graph execution.

This chapter deals with these ergonomic challenges through a meta-scheduling design that provides a set of guiding principles for implementing task runtimes. It takes into account heterogeneous resource management, fault tolerance, dynamic load balancing, interaction with allocation managers and other aspects required to enable effortless execution of task graphs on supercomputers.

This design is then combined with the performance-related insights into task scheduling and minimizing runtime overhead presented in Chapters 5 and 6 in an implementation of an HPC-tailored task runtime called HyperQueue. This task runtime was created with the goal of enabling transparent, ergonomic and efficient execution of task graphs on HPC clusters. It is the culmination of the research and work presented in this thesis, which was made possible thanks to the experience gained from our work on Estee and RSDS and also from interacting with many HPC workflows and use-cases over the course of several years.

The work presented in this chapter makes the following contributions:

1. We propose a meta-scheduling design that provides a unified method for executing tasks in the presence of allocation managers and allows load balancing across different allocations, by disentangling the definition of tasks and the hardware resources that executes them.
2. We describe a novel resource management scheme designed to improve hardware utilization of heterogeneous clusters.
3. We provide an implementation of the meta-scheduling and resource management design in HyperQueue, a task runtime that enables ergonomic and efficient execution of task graphs on HPC clusters.
4. We evaluate the performance, scalability and achieved hardware utilization of task graphs executed using HyperQueue on several benchmarks and use-cases.

This chapter first introduces the proposed general meta-scheduling design and also several concepts useful for improving the utilization of heterogeneous resources. Then it describes the architecture of HyperQueue, which implements the proposed design, and discusses how its features alleviate the challenges described in Chapter 4. Finally, it presents several use-cases where HyperQueue was successfully used, provides a performance analysis of HyperQueue on various use-cases and stress tests and compares it with state-of-the-art meta-scheduling task runtimes.

We have described the design and key ideas of HyperQueue in *HyperQueue: efficient and ergonomic task graphs on HPC clusters* [3]. The architecture of HyperQueue presented in this chapter was adapted from this publication.

I have collaborated on this work with Ada Böhm; we have both contributed to it equally. I am the sole author of the design and implementation of the automatic allocator component, and I have also significantly contributed to the design and implementation of most remaining parts of HyperQueue. I have designed and performed all the experiments described in this chapter. While Ada and I are the primary contributors to HyperQueue, it should be noted that other people have also contributed to it, as its development is a team effort. Source code contribution statistics for HyperQueue can be found on GitHub[1].

7.1 Meta-scheduling design

This section describes a design for executing task graphs that aims to alleviate the challenges described in Chapter 4. It consists of a set of general guidelines that should help drive the design of task runtimes so that they will be able to resolve these challenges. Note that this section focuses on general principles that do not depend so much on a specific implementation. There are some task runtime aspects, such as simple deployment or high runtime efficiency, that have to be resolved with specific implementation choices rather than general principles. These will be mentioned in a following section that will detail the implementation of the HyperQueue task runtime, which implements the described meta-scheduling design.

The primary challenge that affects the simplicity of task graph execution on HPC clusters is the presence of an allocation manager, because it forms a barrier; on such clusters, it is not possible to compute anything at all without interacting with allocations. In order for task graph execution to become truly ergonomic, it is the first and foremost challenge that needs to be resolved. It is also tied to most of the other mentioned challenges; heterogeneous resource requirements, fault tolerance and efficient hardware utilization on supercomputers are all affected by the way the task runtime handles allocations. It will thus be the main focus of the described guidelines.

Section 4.1 has described several approaches that can be used to map tasks to allocations, as well as their various shortcomings. Due to the limits imposed by allocation managers, it might be necessary to partition task graphs into multiple subgraphs in order to fit them within an allocation, which can be challenging. Furthermore, it can result in non-optimal usage of hardware resources, because tasks from (sub)graphs submitted in separate allocations will only be load balanced within their own allocation rather than also across different allocations.

[1] `https://github.com/it4innovations/hyperqueue/graphs/contributors`

It is important to note what is uniquely challenging about the interaction with allocations. The fact that the computation needs to go through a queue is not an issue by itself, as we are dealing with batch processing anyway. Therefore, some form of a delay between submitting the task graph and receiving the results is expected, even if we did not have to submit allocations at all.

The main issue of allocations is that they strictly tie together two separate aspects; *what* the user wants to compute (the computation performed once the allocation starts) and *where* should the computation take place (specific hardware resources and computational nodes reserved for the allocation). As was already described previously, both of these things have to be specified together in an allocation. This is a very inflexible design because of several reasons:

- The user needs to consider both aspects at the same time. Ultimately, the main thing that the user cares about is what they want to compute. With allocations, they also need to think about specific hardware resources that should be allocated for computing their tasks, and how should tasks be mapped to these resources, which can be challenging.
- Hardware resources are allocated for the whole duration of the allocation. This can result in inefficient resource usage, especially for heterogeneous task graphs that consist of many types of tasks that use different resources. In situations where not all resources can be used at the same time, some of the resources can unnecessarily remain idle.
- The amount of used resources has to be decided up front; new hardware resources cannot be added or removed from existing allocations. Apart from the potential resource waste, this can also limit load balancing. If load balancing only happens within (and not across) allocations, then resources from different allocations cannot be pooled together to make the task graph execution faster, and users also cannot easily add new hardware resources during the computation of a task graph.
- Granularity of the allocated resources might not match the granularity of tasks. Unless the allocation manager supports very fine-grained allocations (e.g. on the level of individual CPU cores rather than whole computational nodes), there can be a large gap between the resources required by a task graph and the resources provided to an individual allocation. This can again lead to resources sitting idle.
- Binding computation with specific hardware resources up front complicates handling task failures. When the user submits thousands of tasks on a specific set of hardware resources and some of these tasks fail, the user might need to create a new allocation to recompute the failed tasks. This requires once again figuring out a new set of hardware resources that should be requested, as the original resources might not be a good fit for just a subset of the original computation. A fault-tolerant design would ideally allow recomputing tasks on any compatible hardware resource transparently, which is incompatible with forcing computations and resources to be defined together.

As an aside, it is interesting to note that some of these challenges uniquely affect task graphs, and in general programming models that are different from the traditional ways of defining HPC computations. Consider a distributed application implemented using MPI, which has historically

been a common way of defining computations on supercomputers. MPI applications typically assume that they will run on a set of computational nodes for a relatively long duration (hours or even days). This set of nodes (corresponding to MPI processes) usually does not change during the computation; it is not trivial to add new processes, and if some of the processes crash, it typically leads to the whole computation being aborted, as fault tolerance is not a default property of MPI applications [31].

These properties of MPI applications are similar to the mentioned properties of allocations, so they fit together well; using allocations to execute them is relatively straightforward. In fact, it is clear that the allocation model itself was designed with MPI-like use-cases in mind. Therefore, it is not very surprising that the concept of allocations is not a good match for programming models that are very different from MPI, such as task-based workflows.

In order to remove this problematic aspect of allocations, the key idea of the proposed meta-scheduling design is to completely disentangle the definition of what the user wants to compute (tasks) from the hardware resources where the computation should take place (computational nodes, CPU cores, etc.). By separating these two concerns, we enable users to focus on what they care about the most (their task graphs), instead of having to think about mapping tasks to allocations and graph partitioning.

In order to disentangle these two concepts, we have to rethink what kind of computation is submitted in allocations. Instead of making allocations execute tasks or task graphs directly, they should execute generic computational providers, which will then be dynamically assigned work (tasks to execute) based on the current computational load. This both improves the achievable hardware utilization and simplifies the allocation submission process. The following principles describe a meta-scheduling approach based on this idea, along with several additional guidelines designed for ergonomic task execution.

Task runtime runs outside of allocations In order to allow fully dynamic load balancing, the task runtime should not be tied to a specific allocation. Instead, it should run at some persistent location of the cluster (e.g. on a login node). This enables users to submit tasks to it independently of allocations, which is a crucial property. This removes the need to decide which tasks should be computed on which concrete hardware resources and in which allocations up front. It can also help improve hardware utilization, because it provides the task runtime the possibility to load balance tasks across all active allocations.

This is where the term *meta-scheduling* comes in; the essence of the idea is to use a task runtime as a sort of a high-level scheduler on top of an allocation manager. Instead of forcing users to think in terms of allocations, they can submit task graphs in the same way they would on a system that is not supervised by allocation managers and let the task runtime automatically decide in which allocation a task should be executed.

With this approach, the task runtime will most likely run in an environment that is shared with other users of the cluster, rather than being executed inside secured and isolated allocations. It should thus offer its users an option to encrypt its communication to avoid other users of the cluster reading sensitive data (e.g. task outputs) sent from workers.

This should improve both **ergonomics**, by automating the partitioning of tasks to allocations, and also **efficiency**, by allowing the scheduler to load balance tasks across different allocations.

Allocations are uniform Even with the task runtime running outside an allocation, it is still necessary to submit *some* allocations to provide hardware resources that will actually execute tasks. Instead of defining specific tasks that should be computed within an allocation, allocations should execute a generic computational provider (worker) that will be assigned tasks to execute dynamically by the task runtime. This assignment can happen e.g. by a centralized component pushing tasks to workers over the network, or by the workers pulling work from a queue or a distributed database.

With this approach, allocations become trivial and completely uniform, because they all have the same structure; each allocation simply starts a (set of) computational provider(s). This makes it much easier for users to submit allocations. Instead of thinking about how to partition their task graphs, users simply decide how many computational resources they want to spend at any given moment and then they start the corresponding number of allocations.

This should improve **ergonomics**, by making it easier for users (and the task runtime) to provide computational resources.

Allocations are submittable automatically A corollary of the previous principle is that it becomes possible to submit allocations fully automatically by the task runtime. It can use its knowledge of the current computational load to submit workers whose resources would help execute tasks that are currently waiting for hardware resources. This saves users from performing a laborious manual process (allocation submission) and can further help improve hardware utilization, since the task runtime can use its knowledge to dynamically increase and decrease the number of active workers.

This approach helps improving **ergonomics** by removing the need to manually submit allocations.

Failed tasks are retried automatically Once allocations become generic computational providers, their lifetime is no longer bound to a specific set of tasks. And since allocations often end abruptly, they might disappear in the midst of a task being computed. This is not a failure of the task per se; it is a mostly unavoidable consequence of allocations being transient. Such tasks should thus be automatically reassigned by the task runtime to a different worker without the user's intervention, in order to ensure uninterrupted execution of the task graph.

The fact that the task runtime resides outside of allocations further facilitates fault tolerance, because an allocation failure will not affect the state of tasks stored in the task runtime. In order to also be resilient against e.g. login node failures, the task runtime should be able to restore its state from persistent storage (e.g. a filesystem or a database).

Automatic task re-execution improves **ergonomics**, because users do not have to find failed tasks and manually resubmit them.

Dependencies are independent of allocations The allocation manager should not act as a task runtime and thus it should not handle dependencies between tasks. These should be expressed

solely on the level of the task runtime itself and they should be fully independent of allocations. This approach also facilitates execution of the task runtime on a local computer, where there most likely will not be any allocation manager available.

Defining task graphs without any relation to allocations improves **ergonomics**, because it makes it simpler to prototype task graph execution on users' personal computers.

Tasks are paired with workers using abstract resources The first two principles described above separate the two core pillars of task graph execution; the definition of the task graph and the provisioning of hardware resources. However, it is in fact required to somehow tie these two aspects together, because tasks can have resource requirements that constraint the environment in which they can be executed.

We saw that submitting tasks directly as allocations, which explicitly ties tasks to a set of hardware resources, had many disadvantages. Instead, the task runtime should provide a resource system that will allow users to pair tasks with workers in a general and abstract way. Each task can describe a set of abstract resources that have to be available so that the task can be executed and each worker will in turn provide a set of abstract resources that it manages. The task runtime will then dynamically assign tasks to workers by matching these resources together, while making sure that worker resources are not being oversubscribed.

Furthermore, it should be possible to define arbitrary kinds of resources to support heterogeneous clusters that might have custom hardware resources, such as special accelerator devices or network interconnects. Complex heterogeneous resource management use-cases should also be supported; these will be described in more detail in Section 7.2.

Crucially, this approach allows users to avoid considering the granularity of hardware resources provided in allocations. Even if the allocation manager always provides at least a whole node in each allocation, multiple granular tasks (even from completely independent task graphs) that require e.g. only a single core can be load balanced onto that same node.

A generic resource system should improve **efficiency**, by providing the scheduler with more opportunities to load balance computation and by allowing tasks to express precise resource requirements that can be useful especially on heterogeneous clusters.

This set of guidelines should enable execution of heterogeneous task graphs on supercomputers in a straightforward way. Thanks to moving the task runtime outside of allocations, we can sidestep the limits associated with executing tasks in allocations directly, such as limited support for dependencies, limited scale of task graphs that can be executed, and potentially non-optimal load balancing and hardware utilization. In combination with a generic resource system, automatic allocations and automatic re-execution of failed tasks by default, this design should provide an ergonomic experience for task graph authors.

As with any approach, there are certain trade-offs associated with these guidelines. For example, running the task runtime on a login node might be problematic if the login nodes are severely computationally constrained or if they are not even able to connect to compute nodes of the HPC cluster through the network. Performance and security aspects of this deployment method also

need to be considered. These trade-offs will depend on a specific implementation of a task runtime that will leverage the described guidelines; they will be evaluated in Section 7.3.

In addition to these meta-scheduling guidelines, we will also describe a novel approach for managing heterogeneous resources in the following section.

7.2 Heterogeneous resource management

As was already discussed in Section 4.2, support for heterogeneous resource management in existing task runtimes is relatively limited. Most tools support defining task resource requirements for a known set of resources (such as CPU cores or GPUs), some task runtimes like Dask or SnakeMake allow defining arbitrary resource kinds, and several task runtimes, such as PyCOMPSs or Ray, also provide the ability to specify multi-node, with various constraints.

However, there are various use-cases that could benefit from a more comprehensive resource management support. Below, we describe four concepts for advanced management of heterogeneous resources, which should improve hardware utilization on heterogeneous clusters and make it more flexible to define heterogeneous task graphs.

7.2.1 Non-fungible resources

Existing task runtimes treat resources as sets of fungible elements without an identity. For example, if we specify that a Dask worker provides two GPUs, the Dask scheduler will make sure (in accordance with Definition 5) that it executes a single task that requires two GPUs or at most two tasks that require a single GPU on that worker at any given time. However, it does not provide the task with information about *which* specific *resource elements* of the given resource kind were assigned to it, because resources are treated merely as numbers by the Dask scheduler.

Yet in some cases, it would be quite useful to make resources non-fungible by assigning identities to them, so that tasks could leverage knowledge about which specific resource elements were assigned to them. As an example, assume that we have a worker that provides two GPUs with separate identities (A and B) and a task that requires a single GPU. If a task runtime would track the identities of the individual resources, it could tell the task that it was assigned e.g. the GPU B. It would then be much easier for the task to execute on the specific resource that it was assigned; in the case of an NVIDIA GPU, it could leverage the `CUDA_VISIBLE_DEVICES=B` environment variable to make sure that it executes on GPU B. Without support for resources with identities, the task would need to use a different method to determine which specific GPU it should use.

We can formally define non-fungible resources by modifying Definition 2 and also Definition 5 to add support for resources with identities.

[**Definition 7**] A *computational environment with non-fungible resources* is a tuple $C^{id} = (W, RK_w^{id}, Res_w^{id})$, where:

- W is a set of workers.
- RK_w^{id} is a set of resource kinds; each resource kind is a set of *resource elements*, non-fungible resources that can be provided by workers.

- Let $RE_{C^{id}}$ be a set of all resource elements of C^{id}:
  $RE_{C^{id}} = \bigcup_{r \in RK_w^{id}} r$
- $Res_w^{id} \colon W \times RK_w^{id} \to \mathcal{P}(RE_{C^{id}})$ is a function that defines a set of non-fungible resource elements provided by a worker for a specific resource kind.

  In the original definition of a computational environment (Definition 2), Res_w assigned a numerical amount to each resource kind. In this definition, the worker manages a specific set of unique resource elements; therefore, Res_w^{id} returns a set of elements instead.
- Res_w^{id} has to return resource elements belonging to the correct resource kind:
  $\forall w \in W, \forall r \in RK_w^{id} : Res_w^{id}(w, r) \subseteq r$
- Workers do not share resource elements between each other:
  $\forall w_1 \in W, \forall w_2 \in W, \forall r \in RK_w^{id} \colon w_1 \neq w_2 \Rightarrow Res_w^{id}(w_1, r) \cap Res_w^{id}(w_2, r) = \emptyset$
- Workers do not share resource elements between different resource kinds:
  $\forall w \in W, \forall r_1 \in RK_w^{id}, \forall r_2 \in RK_w^{id} \colon r_1 \neq r_2 \Rightarrow Res_w^{id}(w, r_1) \cap Res_w^{id}(w, r_2) = \emptyset$

This definition describes a computational environment where each worker manages a set of unique resource elements that are non-fungible; each one has a separate identity and cannot be interchanged with another resource element.

[**Definition 8**] A *task graph execution with non-fungible resources* is a tuple $E^{id} = (G, C^{id}, X, RA)$, where:

- $G = (T, O, A, RK_w^{id}, Res_t)$ forms a *task graph.*
- $C^{id} = (W, RK_w^{id}, Res_w^{id})$ forms a *computational environment with non-fungible resources.*
- X is a function that maps a task to a worker that is executing it at a given point in time; this corresponds to X defined in Definition 3.
- $RA \colon T \times W \times RK_w^{id} \times \mathbb{R}_{\geq 0} \to \mathcal{P}(RE_{C^{id}})$ is a function that returns a set of specific resource elements of a given resource kind that were assigned to a task running on a specific worker at a given point in time in E^{id}.

  Note that in the previous definition of a computational environment (Definition 2) we did not need the RA function to define worker and task resource constraints (Definition 5); Res_t was used instead. However, that is no longer enough in the presence of non-fungible resources.
- When a task is not being executed at a given point in time, no resource elements are assigned to it:
  $\forall tp \in \mathbb{R}_{\geq 0}, \forall t \in T, \forall w \in W, \forall r \in RK_w^{id} \colon X(t, tp) = \perp \Rightarrow RA(t, w, r, tp) = \emptyset$
- Resource elements allocated to a task that is being executed have to be drawn from the correct resource kind:
  $\forall tp \in \mathbb{R}_{\geq 0}, \forall t \in T, \forall r \in RK_w^{id} \colon X(t, tp) \neq \perp \Rightarrow RA(t, X(t, tp), r, tp) \subseteq r$

Using these modified definitions, we can now redefine the worker and task resource constraints (originally defined in Definition 5) for non-fungible resources.

[**Definition 9**] A *worker resource constraint with non-fungible resources* in the context of execution $E^{id} = ((T, O, A, RK_w^{id}, Res_t), (W, RK_w^{id}, Res_w^{id}), X, RA)$ is defined as follows:
$\forall tp \in \mathbb{R}_{\geq 0}, \forall w \in W, \forall r \in RK_w^{id} \colon \left(\bigcup_{t \in WX_{E^{id}}(w,tp)} RA(t, w, r, tp)\right) \subseteq Res_w^{id}(w, r)$

This property ensures that resource elements assigned to a task belong to the worker that is executing it. If we want to also ensure that a given resource element is not being oversubscribed, i.e. that is used by at most a single task at any given point in time, we could also add the following constraint:

$\forall tp \in \mathbb{R}_{\geq 0}, \forall w \in W, \forall r \in RK_w^{id}, \forall t_1 \in T, \forall t_2 \in T \colon t_1 \neq t_2 \Rightarrow RA(t_1, w, r, tp) \cap RA(t_2, w, r, tp) = \emptyset$

However, in a following section that describes fractional resource requirements, we will see that there are some situations where this constraint might not be desirable.

In addition to the modified *worker resource constraint* we will also need to define a new version of the *task resource constraint*, which ensures that all task resource requirements were satisfied in a given execution. Note that with fungible resources, this property was a trivial consequence of Definition 5; however, that is no longer the case with non-fungible resources.

[**Definition 10**] A *task resource constraint with non-fungible resources* in the context of execution $E^{id} = ((T, O, A, RK_w^{id}, Res_t), (W, RK_w^{id}, Res_w^{id}), X, RA)$ is defined as follows:
$\forall tp \in \mathbb{R}_{\geq 0}, \forall w \in W, \forall t \in WX_{E^{id}}(w, tp), \forall r \in RK_w^{id} \colon Res_t(t, r) = |RA(t, w, r, tp)|$

This property ensures that at any given point in time when it is being executed, each task has been assigned the required amount of resource elements for each resource kind. Note that this definition allows the individual resource elements assigned to tasks to change over time; some task runtimes might leverage this to implement a very dynamic form of resource management.

It is important to note that even with non-fungible resources, tasks should still express their resource requirements using numerical amounts of required resources, as defined by Definition 1, rather than asking for specific resource elements. As an example, a task should ideally not ask for a single specific GPU based on its identity. This is necessary to keep the task graph portable, but crucially also to provide the scheduler with more opportunities for load balancing. If tasks required specific resource elements to execute, it would limit the scheduler's ability to choose resources dynamically in response to computational load.

Taking resource identity into account improves **ergonomics**, because it makes it easier for tasks to correctly use the exact resources assigned to it by a scheduler. This benefit is especially useful when combined with fractional resource requirements, which are described later below.

7.2.2 Related resources

Even though tasks should ideally not ask for specific resource elements in order to remain general, in certain cases it could be useful for a task to ask for a set resource elements that are somehow *related* to each other, primarily for performance reasons. Assume that there is a computational node with 128 CPU cores and we want to execute a task that requires four cores. It might seem that it does

not matter which specific four cores are assigned to that task by the scheduler. However, modern processors are divided into so-called NUMA nodes, where each such node contains a subset of cores and a subset of RAM that they can access very quickly. On the other hand, accessing memory across different NUMA nodes introduces a performance penalty [146]. Assigning four cores that belong to the same NUMA node to a task could thus improve its performance, while assigning cores from different NUMA nodes could reduce its performance. It should be noted that the inverse can also be true; it depends on the nature of the executed program. The important point is that the NUMA placement of cores can significantly affect the performance of programs that are executed with them.

In order to support this use-case, a task runtime should allow tasks to define certain relations between resource elements. Ideally, it should be possible to express these relations in a generic way, because they can affect performance in other situations than just with NUMA, e.g. GPU accelerators can have similar characteristics.

We will not provide a formal definition for this concept, as it is very general and could be designed and implemented in various ways. Section 7.3.4 will describe how it is implemented in HyperQueue.

While it is always possible to configure the CPU or NUMA affinity of workers, most existing task runtimes do not allow configuring resource relations for individual tasks. The Ray [76] task runtime supports a general mechanism called *placement groups* that allows specifying groups of related resources and the PyCOMPSs was extended in [147] to support NUMA placement of tasks.

Specifying related resources is designed to improve **efficiency**, as it should allow the scheduler to understand more precise resource requirements of individual tasks and thus help them achieve better performance.

7.2.3 Fractional resource requirements

As you may recall from Section 3.1 and Definition 1, we have originally defined a resource requirement of a task to for a given resource kind in the form of an integer. However, in some situations this might be too coarse-grained. To support a more fine-grained resource specification, we can leverage a *fractional resource requirement*, which would allow a task to require only a fraction of some resource.

[**Definition 11**] A *task graph with fractional resource requirements* is a tuple $G^{fr} = (T, O, A, RK_t, Res_t^{fr})$, where:

- $Res_t^{fr} \colon T \times RK_t \to \mathbb{Q}_{\geq 0}$ is a function that defines the *fractional resource requirement* of a task for a given resource kind.

The remaining properties of G^{fr} are identical to the *task graph* properties defined in Definition 1. Note that we can reuse both the task and worker resource constraints provided by Definition 5 for *task graphs with fractional resource requirements*, as they still hold even with fractional resource requirements if we simply replace Res_t by Res_t^{fr}.

Support for fractional resource requirements can be useful to express situations where a task is not able to make full use of a resource. As an example, for some programs it can be challenging to achieve full utilization of a GPU accelerator. If the used software can utilize e.g. only 25% of a single GPU, a part of that accelerator's hardware would sit idle while the task is being executed, which would result in resource waste. With fractional resource requirements, we could instead state that the task t requires only a fraction of a GPU: $Res_t^{fr}(t) = 0.25$. A task runtime that supports fractional resource requirements should then be able to schedule up to four such tasks on a single GPU in order to improve its utilization.

Note that in theory, fractional resources could be simulated in a task runtime without explicit support for them. If we would like to use e.g. a fractional resource requirement $\frac{1}{n}$, we could simply multiply all other resource requirements and resource amounts provided by workers by n. However, this would not work in combination with non-fungible resources, as ideally we would like to assign multiple tasks to the exact same resource (e.g. a single specific GPU), and thus have a one-to-one mapping between physical hardware devices and the corresponding resource elements that represent them.

Fractional resources are not supported by most existing task runtimes, although they are present in Ray, which allows assigning fractional resources to individual tasks. The Slurm allocation manager supports *resource sharding* [148], which allows sharing a specific resource by multiple tasks. However, this only works for a known set of resources (notably GPUs) and the resource has to be specially preconfigured by its administrators for it to work. It also works mostly on the level of individual allocations; as was already discussed in Section 4.1, using Slurm allocations directly as tasks causes several issues.

Fractional resource requirements are designed to improve **efficiency** of task graph execution, as they allow achieving higher hardware utilization of powerful hardware resources that are difficult to fully utilize by a single task.

7.2.4 Dynamic resource selection

Existing task runtimes allow specifying a specific amount of a given resource kind that is required by each task, as was defined by Definition 1. This resource requirement is determined before the task graph is executed by the task graph author and it typically does not change during the execution of the workflow. Some task runtimes, such as SnakeMake, allow tasks to define their resource requirements dynamically based on their computed inputs. However, each task can still choose only a single required amount for each resource kind.

There are situations where it could be beneficial to allow tasks to specify several *variants* of resource requirements with which they can be executed. The task runtime (and its scheduler) could then dynamically select the variant that should result in the best utilization of hardware based on the current computational load.

As an example, assume that we have a typical modern HPC cluster that has several powerful GPU accelerators along with tens of CPU cores per computational node, and we want to execute some tasks on its accelerators, so we assign a resource requirement that requires a GPU to them. When these tasks will be executing, the available CPU cores might remain idle if there are no other

tasks to compute at the same time. However, software frameworks that are GPU-accelerated also commonly support execution on a CPU, albeit at a reduced speed [32, 34]. Computing some of these tasks on the CPU while others are being computed on the accelerators could improve the achieved hardware utilization and reduce the total makespan of the executed task graph.

The question then becomes how to determine which tasks to run on the CPU and which on the GPU. We could determine this statically before executing the workflow, by dividing the tasks into two groups. However, we usually do not know exact task durations in advance; it is thus infeasible to estimate the ideal fraction of tasks that should run on the accelerators in order to achieve optimal hardware utilization. What we could do instead is to leverage *dynamic resource selection*. For example, we could specify that a task needs either a single GPU and a single CPU or that it needs 16 CPU cores in the case that a GPU is not currently available, and let the scheduler decide whether to execute that task using CPU or GPU resources dynamically during task graph execution. This allows the scheduler to react to computational load in a flexible way, and thus potentially improve its load-balancing capabilities.

To support this use-case, we can generalize task resource requirements so that each task can specify multiple *resource variants* of resource requirements, where each variant is a separate set of resource requirements. The task runtime can then dynamically choose which variant to use for a task based on current computational load and worker resource usage.

[**Definition 12**] A *task graph with resource variants* is a tuple $G^{var} = (T, O, A, RK_t, Res_t^{var})$, where:

- $Res_t^{var} \colon T \times RK_t \times \mathbb{N}_{>0} \to \mathbb{N}_{\geq 0}$ is a function that defines the resource requirement of a task for a given resource kind and a given resource variant index.

  The individual variants are identified with consecutive numeric indices starting at 1.

- Let $VC_{G^{var}}$ be a function that determines the number of resource variants for a task of G^{var}:

  $$VC_{G^{var}}(t) = \begin{cases} 1, & \text{if } \forall v \in \mathbb{N}_{>0}, \forall r \in RK_t \colon Res_t^{var}(t, r, v) = 0 \\ \max\limits_{v \in \mathbb{N}_{>0}} \exists r \in RK_t \colon Res_t^{var}(t, r, v) > 0, & \text{otherwise} \end{cases}$$

  If a task does not define any resource requirements, then $VC_{G^{var}}(t) = 1$. In that case, the task behaves as if it had a single variant that does not require any resources.

The remaining properties of G^{var} are identical to the *task graph* properties defined in Definition 1.

[**Definition 13**] A *task graph execution with resource variants* is a tuple $E^{var} = (G^{var}, C, X, RV)$, where:

- $G^{var} = (T, O, A, RK, Res_t^{var})$ forms a *task graph with resource variants*.
- $C = (W, RK, Res_w)$ forms a *computational environment*.
- X is a function that maps a task to a worker that is executing it at a given point in time; this corresponds to X defined in Definition 3.
- $RV \colon T \times \mathbb{R}_{\geq 0} \to \mathbb{N}_{>0}$ is a function that returns a specific resource variant index that was assigned to a task running on some worker at a given point in time.

- RV has to uphold the following condition:
  $\forall tp \in \mathbb{R}_{\geq 0}, \forall t \in T\colon X(t, tp) \neq \perp \Rightarrow RV(t, tp) \in \{1, \ldots, VC_{G^{var}}(t)\}$

  Informally, when a task is executing, it has to be assigned a valid variant index. Note that when a task is not being executed at a given point in time, RV is allowed to return an arbitrary non-zero natural number.

Using the two previous definitions, we can now also define a modified version of both the worker and task resource constraints:

[**Definition 14**] A *task resource constraint with resource variants* and a *worker resource constraint with resource variants* in the context of execution $E^{var} = (G^{var}, C, X, RV)$, where $G^{var} = (T, O, A, RK, Res_t^{var})$ is a *task graph with resource variants* and $C = (W, RK, Res_w)$ is a *computational environment*, are defined as follows:
$\forall tp \in \mathbb{R}_{\geq 0}, \forall w \in W, \forall r \in RK\colon \left(\sum_{t \in WX_{E^{var}}(w,tp)} Res_t^{var}(t, r, RV(t, tp))\right) \leq Res_w(w, r)$

Informally, each task has an assigned resource variant, whose resource requirements all need to be satisfied at any given point in time the task was being executed in a given task graph execution. Same as with Definition 5, this definition satisfies both the task and the worker resource constraints.

To our knowledge, there is no existing task runtime that would support all the described advanced resource management concepts at once; we are also not aware of other work that would support non-fungible resources and resource variants at all. Implementing these concepts could improve the flexibility of task graph execution on heterogeneous clusters; this will be described in more detail and evaluated in the following sections.

Dynamic resource selection should improve **efficiency**, by allowing tasks to provide the scheduler with several alternative resource options that they are able to work with, which then allows the scheduler to dynamically select the best option during load balancing based on current computational load.

7.3 HyperQueue

To evaluate how the proposed meta-scheduling and heterogeneous resource management approaches work in practice, what their performance and usage implications are and how they compare with existing meta-schedulers, we have implemented them in a task runtime called HyperQueue, which will be described in the rest of this chapter.

HyperQueue is a distributed task runtime that aims to make task graph execution a first-class concept on supercomputers. It does that by implementing the meta-scheduling design described in the previous section and leveraging insights gained from our task scheduling experiments and Dask runtime efficiency analysis. In terms of architecture and implementation, it is an evolution of the RSDS task runtime described in Chapter 6; however, instead of using the Dask components and APIs, it implements its own task-based programming model. HyperQueue is developed in the Rust programming language [149] and it is provided as an MIT-licensed open-source tool [150].

This section describes the high-level design of HyperQueue, its programming model and its most important features. The following sections present several use-cases where it has been successfully leveraged and evaluate its performance and other aspects with various experiments.

As a reminder, Slurm will be used as a default representative of an allocation manager throughout this whole chapter; however, it could be replaced by PBS or any other commonly used allocation manager without loss of generality.

7.3.1 Architecture

HyperQueue uses a fairly standard distributed architecture. It consists of two main components, a central management service called a *server* and a component that serves as a computational provider which executes tasks, called a *worker*. Users then interact with the server using a *client* interface. Figure 7.1 displays a high-level view of the HyperQueue architecture for a typical deployment on an HPC cluster with an allocation manager, where the server runs on a login node and meta-schedules tasks to workers that run on computational nodes in various allocations. The individual components are described in more detail below.

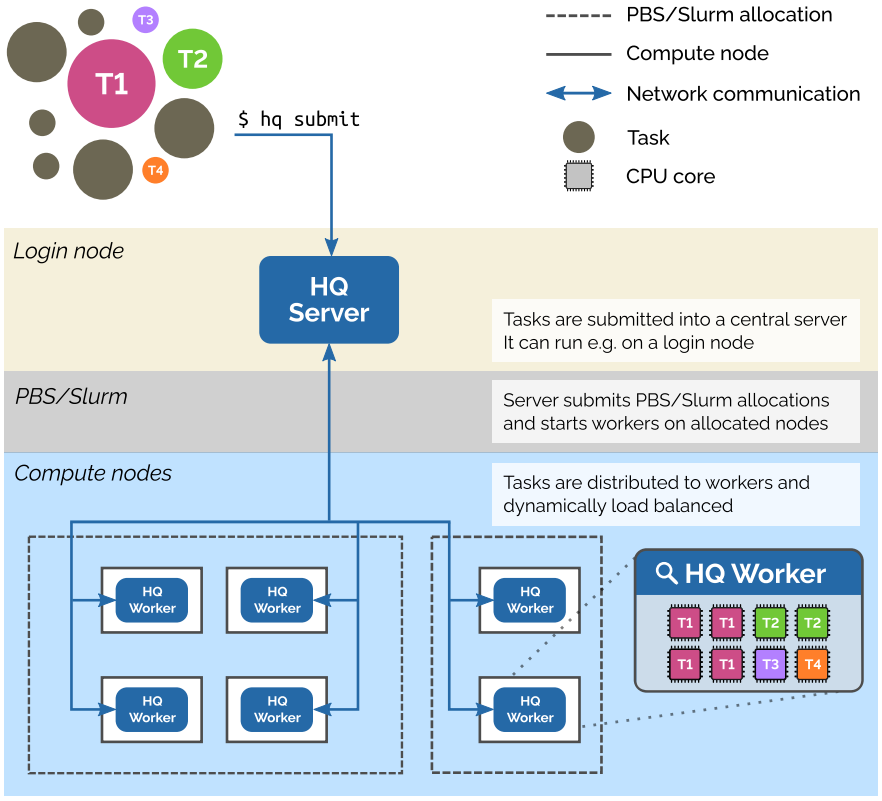


Figure 7.1: Architecture of HyperQueue.

Server

The *server* is the most important part of HyperQueue. Its main goal is to manage the lifecycle of both task graphs and workers; it keeps track of them and allows users to query their state and also send commands to it through a client interface. The server contains a work-stealing task scheduler that is based on the RSDS scheduler implementation described in Section 6.3. The scheduler assigns individual tasks to workers based on task resource requirements and current computational load of workers, with the goal of maximizing hardware utilization of the connected workers.

The server is designed to be executed as a long-running background process that should ideally be executed in a persistent location not managed by allocation managers (i.e. outside transient allocations). Typically, it is executed on a login node of a cluster, but it can also run elsewhere, for example in a cloud partition. It should keep running until all tasks that the user wants to execute are finished, although it can also restore its state from disk, so the computation of a task graph can be interrupted and resumed at a later time, if needed.

The fact that the server runs outside allocations allows it to load balance tasks across workers running in multiple allocations concurrently, and work around the various limits of allocation managers. This adheres to the first principle of the meta-scheduling design described in the previous section.

The server itself does not execute any tasks, and it consumes a relatively modest amount of resources; it is thus not computationally demanding. Section 7.5.11 evaluates how many CPU resources the server consumes in extreme scenarios.

Apart from basic responsibilities related to task scheduling and worker management, the server also provides additional functionality. For example, it can submit allocations fully automatically on behalf of the user. This feature will be described in detail in Section 7.3.6.

The inner architecture of the server is displayed in Figure 7.2. The server is divided into two parts; frontend and backend. The frontend part is responsible for managing a database of task graphs and workers and providing access to it to clients. It also stores all important events (such as a task being completed or a task graph being submitted) into a journal file, from which it can restore its state in case of an unexpected failure. The frontend also runs the automatic allocator, which submits allocations into Slurm or PBS to spawn new workers.

The backend is the performance-critical part of the server. It does not separate individual task graphs submitted by users, it simply works with a set of tasks that it schedules to workers. The *Reactor* component passes task events received from workers to a separate *Scheduler* component that computes the assignments of tasks to workers using the *b-level* heuristic and work-stealing (this implementation is based on the RSDS scheduler). It then returns the schedule back to the reactor, which takes care of sending tasks to workers that were assigned to them. The backend also contains a component for streaming standard (error) output generated by tasks from workers into binary files, which can be used to avoid putting too much stress on distributed filesystems. This will be described in Section 7.3.3.

Even though both parts (frontend and backend) currently run inside the same Linux process, they are strictly separated and communicate together using a well-defined interface. Therefore, it would be possible to deploy the backend in a separate process or even run it on a different node, in

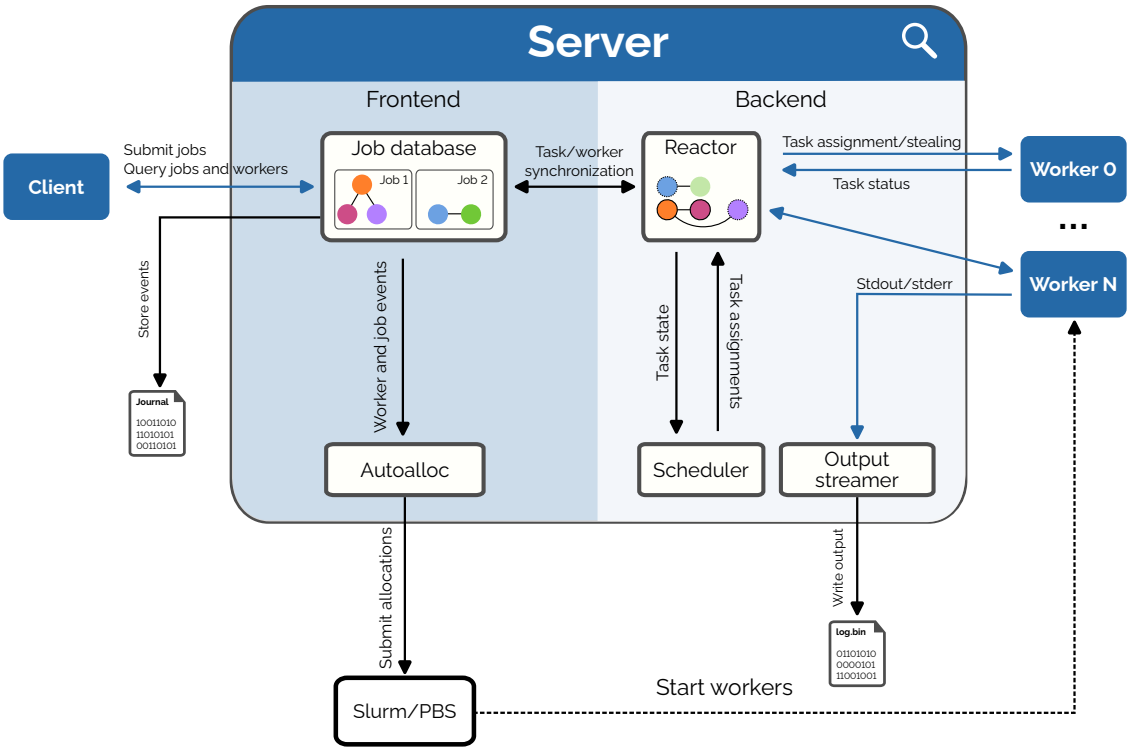


Figure 7.2: Architecture of the HyperQueue server.

which case the two parts would communicate over the network. This could be useful for example to run the backend on a more powerful computational node.

Worker

The *worker* component is a general computational provider that executes tasks assigned to it by the server. A single worker typically manages all hardware resources (CPU cores, GPU or FPGA accelerators, memory, etc.) of a given computer, usually a single computational node of a supercomputer.

The worker is able to automatically detect all relevant hardware resources that are available on the node where it is started and advertise them to the server. Therefore, in a typical case, users can simply start the worker on a given node with a single uniform command, and from that moment on the resources of that node will be used for task execution. This adheres to the second principle of the proposed meta-scheduling design, where computational providers should be generic (not tied to any specific task or a set of tasks) and it should be possible to start them using a uniform command, to make their deployment trivial.

Each worker also participates in task and resource scheduling. The server makes high-level scheduling decisions, such as which tasks will be executed on a given worker, while the worker then performs micro decisions, such as in which order it will execute tasks assigned to it or which specific

resource elements will be assigned to a given task. Resource management will be described in more detail in Section 7.3.4.

HyperQueue workers were designed with fault tolerance in mind. On an HPC cluster, workers will be typically executed inside allocations that only run for a limited amount of time, which implies that workers can disappear at any time during the computation of a task graph. Both the server and workers thus operate with the assumption that workers can be potentially short-lived; if a worker disconnects while it is executing a task, the task will be transparently reassigned to a different worker without user intervention. Worker management is thus dynamic and flexible; new workers can also be added at any time, after which they will immediately start contributing to the execution of the submitted task graphs. This property enables users of arbitrarily scale the computation of a task graph both up and down even while it is being executed, by dynamically adding or removing workers as needed.

While workers can be deployed manually by users, HyperQueue can also automatically submit allocations to an allocation manager that start workers on computational nodes based on current computational needs of submitted task graphs, as was already mentioned. This system, called *automatic allocation*, will be described in more detail later in Section 7.3.6.

Client interfaces

Users can use two main interfaces for deploying the HyperQueue server and workers, submitting task graphs and managing and monitoring the computation. They can either use a CLI, which has been designed to feel familiar to users of Slurm in order to make it easier to migrate existing HPC workflows to HyperQueue, or a Python API, which is useful for more complex use-cases that are challenging to express using the command line. These interfaces are relatively comprehensive and their specific shape is not crucial for this thesis; the complete set of supported CLI commands and their parameters and the Python API is described in the HyperQueue documentation [151]. In the rest of the text, we will show various examples of these interfaces (primarily of various CLI commands), to provide an idea of how ergonomic it is for users to interact with HyperQueue.

The CLI has been designed with the "Simple things should be simple. Complex things should be possible."[2] approach in mind. It allows submitting various kinds of workflows in an easy way, but also allows users to configure their execution extensively. This is in contrast to some other task runtimes, which require using Python code (e.g. Dask) or workflow files (e.g. SnakeMake) for executing even the simplest of workflows. HyperQueue also supports defining task graphs using workflow files, although it is optional and usually not needed for simple use-cases. Listing 7.1 shows three basic commands that leverage the `hq` binary to deploy an instance of a HyperQueue server and a worker, and then execute a single task through it. More examples will be shown later throughout this chapter. For simple existing workflows defined using Bash scripts, it is often enough to replace uses of the `sbatch` command with `hq submit` to let HyperQueue manage task execution instead of Slurm.

[2]Originally coined by Alan Kay.

```
# Start a HQ server
$ hq server start &

# Start a HQ worker
$ hq worker start &

# Submit a HQ job (task graph) with a single task
$ hq submit ./my-program
```

Listing 7.1: Examples of HyperQueue CLI commands

Deployment

Even though it is often overlooked, deployment of software on supercomputers can be challenging, as was described in Section 4.6. Since HyperQueue aims to provide seamless support for HPC use-cases, it was also designed to be trivial to deploy. It is distributed as a single, relatively small (approximately 15 MiB), statically linked executable called `hq`, which implements all the provided functionality (server, worker and the whole CLI). It does not depend on any external services or dynamic libraries, apart from the ubiquitous `C` standard library, and it runs fully in userspace and does not require any elevated privileges. It also does not require any installation nor configuration.

The HyperQueue Python API is then distributed as a Python package that is available on the standard Python Package Index [152]. This package contains all the relevant code inline; it is thus self-contained and does not require the `hq` binary.

Both clients and workers communicate with the server using the standard TCP/IP protocol, which is ubiquitously available on most clusters. By default, all communication is encrypted, so that other users of the cluster cannot read the data sent between the HyperQueue components. This is performed out of an abundance of caution, because the server typically runs on a login node shared by multiple users, rather than on computational nodes that are usually isolated between users by the allocation manager. The performance overhead of encryption will be evaluated in Section 7.5.12.

In order to connect the individual components using TCP/IP, normally it would be required for users to specify network hostnames and ports. While this process is relatively simple, it still presents a minor ergonomic hurdle, because users would need to reconfigure their worker deployment scripts and client commands every time the server would start with a new TCP/IP address and port. To make this easier, HyperQueue removes the need to specify hostnames and ports by default, by exploiting a useful property of HPC clusters, which commonly use a shared networked filesystem. When a server is created, it creates a configuration file in the user's home directory, which contains all information necessary for exchanging handshakes and connecting the components, including encryption keys. When workers and clients discover the presence of this file (which can be accessed due to the filesystem being shared), they are able to connect to the server without the user having to specify any addresses. However, it is still possible to deploy multiple concurrent instances of HyperQueue under the same account; users just need to provide separate directories for storing the configuration file.

Thanks to these properties, it is trivial to deploy and use HyperQueue on supercomputers even

without the involvement of cluster administrators. The deployment benefits extend beyond HPC clusters; users can trivially deploy HyperQueue on their own personal computers. This enables them to prototype their HPC workflows locally, which can accelerate their development process and is another step towards improving the ergonomics of using task graphs on supercomputers. This is in contrast with using allocation managers for submitting tasks directly, which makes local experimentation difficult, as allocation managers are typically not straightforward to deploy.

7.3.2 Programming model

The task-based programming model used by HyperQueue builds upon the task model defined in Chapter 3. It adds some additional HPC-focused features on top of it, such as fine-grained resource management and support for multi-node tasks.

The core element of the programming model is a *task*, which has the following properties:

- It is a unit of computation. It can be executed by a worker (or a set of workers in the case of multi-node tasks). In the current implementation, each task represents either the execution of a binary executable or the invocation of a Python function.
- It is a unit of scheduling. The scheduler assigns tasks to workers, and can move tasks between workers using work-stealing if their load is unbalanced.
- It forms a failure boundary. When a worker crashes while executing a set of tasks, they are scheduled again onto a different worker and their execution is restarted from scratch.
- It is a unit of dependency management. Each task may depend on a set of other tasks; in this way tasks are composed in a DAG.
- It is a unit of resource management. Each task has a set of resource requirements; the scheduler matches them with resources provided by workers.

Each task also has various additional properties, such as a path to the executed binary (or a Python function), a set of environment variables, a working directory, and others.

Tasks are composed into task graphs, which are called *jobs* in the HyperQueue terminology. Each task belongs to exactly one job; jobs are completely independent, so there can be no dependencies between tasks belonging to different jobs. Jobs are units of monitoring and management, as they allow users to group many tasks together, observe their state or perform operations on all (or a selected subset of) tasks of a job at once. There are two kinds of jobs, *graph jobs* and *task arrays*. Graph jobs correspond to arbitrary DAGs, where tasks may depend on one another. These can be only created using the Python API or a TOML (Tom's Obvious Minimal Language) workflow file, since it would be difficult for users to express dependencies using a command-line interface.

Task arrays are a special case of graph jobs that use a compressed representation of a potentially large task graph with a regular structure. For example, a common use-case is to compute a task graph where each task runs the same computation, but on a different input. This can be achieved with a task array, which stores only a single instance of the task definition (what should be computed), but creates many copies of this task, each with a different input. This allows users

to easily execute large task graphs using the CLI, and it also reduces the memory usage of the server, because the task graph is stored in a compressed format.

Each task in the task array gets assigned a different *task ID*, which uniquely identifies a specific task, which allows it to perform its computation on a different input. For example, if a user creates a task array with a thousand tasks, then by default, the first will have ID 1, the second one will have ID 2, etc. The ID is passed to the task through an environment variable. Apart from assigning unique inputs to tasks through task IDs, HyperQueue also supports passing a unique binary blob to each task (again through an environment variable) that can contain arbitrary data. Listing 7.2 shows an example of how users may submit task arrays through the CLI.

```
# Task array with 10 tasks, each task gets a different ID
$ hq submit --array=1-10 ./my-script.sh

# Each task will receive a single line from the given file
$ hq submit --each-line=inputs.txt ./my-script.sh

# Each task will receive a single item from a JSON array stored in the given file
$ hq submit --from-json=items.json ./my-script.sh
```

Listing 7.2: Creating task arrays using the HyperQueue CLI

Task and job lifecycle

The server manages the state lifecycle of tasks and jobs and also reports them to the user, so that they can observe what is happening with their computation. At any given time, each task can be in a single *state*:

Waiting After the user creates (*submits*) a task, it starts in the *waiting* state, where it waits until all its dependencies are finished and until there is a worker that is able to fulfill its resource requirements.

Running After a task is scheduled and starts executing on a worker, it moves to the *running* state. If the worker that is executing the task stops or crashes, the task will move back to the *waiting* state, so that it can be rescheduled to a different worker. This is labeled as a *task restart*.

Finished When a task finishes successfully (a binary program exits with the exit code 0 or a Python function returns without throwing an exception), it moves to the *finished* state.

Failed When a task fails (a binary program exits with a non-zero exit code or a Python function throws an exception), it moves to the *failed* state.

Canceled When a waiting or a running task is canceled by the user, it moves to the *canceled* state, and it is no longer considered for execution. Users can cancel tasks and jobs that are no longer relevant and that should not continue executing.

Figure 7.3 displays a state diagram of the various task states and transitions that cause state changes. The *finished*, *failed* and *canceled* states are terminal; once a task reaches that state, it

cannot change its state anymore. Each job also has its associated state that is derived from the state of its tasks.

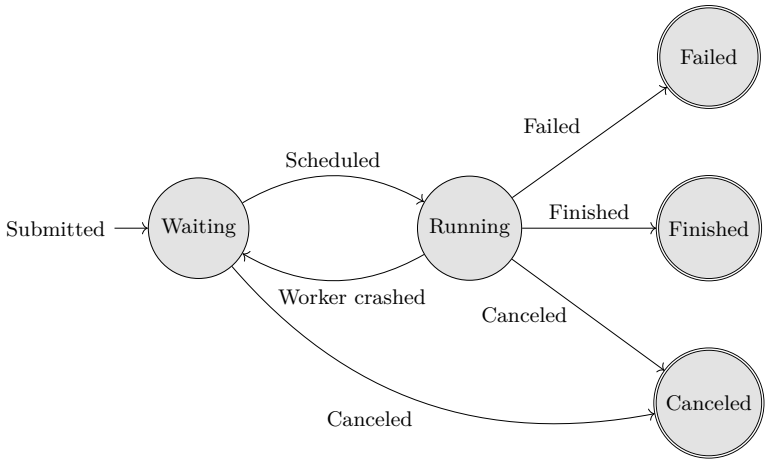


Figure 7.3: State diagram of HyperQueue tasks

Iterative computation

Iterative computation, such as running a simulation until it converges or training a machine-learning model, can be expressed using the concept of *open jobs*. Such jobs allow users to dynamically add tasks to them, even if some tasks were already computed. This can be used to perform several iterations until some condition is met within the same job, without unnecessarily creating a new task graph (job) for each iteration. This makes it easier to express arbitrarily dynamic task graph structures where the shape of the graph is not fully known before the computation starts.

7.3.3 Task input and output

By default, each HyperQueue task stores its *standard output* (*stdout*) and *standard error output* (*stderr*) streams to the filesystem. Users can override this to avoid storing these output streams, or to let HyperQueue remove them if they are empty after the task has finished executing, to reduce disk clutter. It is also possible to use various placeholders for configuring the paths of these streams. For example, the string `%{TASK_ID}` in a stream path will get resolved to the task ID of the task that produces the stream. This can be used to easily separate the output of different tasks.

Since task graphs can be large, generating two files per task, especially on shared network filesystems that are commonly used on supercomputers, could cause performance bottlenecks or run into limitations caused by user disk quotas. To alleviate this problem, HyperQueue supports an output management system called *output streaming*. If it is enabled for a given job, then *stdout* and *stderr* of tasks is streamed over the network from workers to the server, instead of being stored into separate files. The server then stores all this data into a single *logfile*. It also offers several commands for manipulating and filtering task output data from this file. This helps to reduce

the pressure put on the filesystem. Section 7.5.13 will evaluate how this system can help resolve potential I/O issues.

HyperQueue currently does not support data transfers between tasks. Arbitrary binary data can be attached as an input for any task (through its *standard input* stream), but task outputs are only used for specifying dependencies; they cannot be transferred between the workers directly. This is only relevant for the Python API, because tasks submitted using the CLI cannot easily express data transfers. The HyperQueue server is based on the RSDS server implementation, which supports data transfers between workers, so the transfer functionality itself is actually implemented. However, data transfers still need to be incorporated into the HyperQueue programming model, and a user-facing (Python) interface for consuming outputs of tasks needs to be designed. It should be noted that this is a limitation of the current implementation that we plan to lift in future work.

7.3.4 Resource management and scheduling

As has already been mentioned, the HyperQueue scheduler is derived from the RSDS work-stealing scheduler that was described in Section 6.3. It assigns tasks to workers eagerly, to optimize the fast path where there are enough tasks to utilize the available workers. When some workers are underutilized, the server uses work-stealing to steal tasks between workers to balance the computational load. Scheduling is performed on the level of tasks, regardless of which job they belong to; the scheduler does not even know anything about jobs, as they exist only in the frontend. The scheduler ensures that the *dependency constraint* (Definition 4) is always upheld.

What makes the scheduler different from RSDS and other state-of-the-art task runtimes is its support for the heterogeneous resource management concepts described in Section 7.2. It supports non-fungible resources, related resources, fractional resource requirements, resource variants and also multi-node tasks in a single unified resource system. When a task is being scheduled, the scheduler looks for a worker that can satisfy all resource requirements of that task, based on the amount of resource elements that the worker currently has available. When a task is scheduled onto a worker, the scheduler *allocates* a specific subset of the worker's resources to that task, which will not be available for other tasks until that task finishes its execution.

HyperQueue resources are *generic*; users can define their own arbitrary resource kinds for both tasks and workers. At the same time, workers also recognize several known resource kind names and provide specialized support for them. Resource management in HyperQueue consists of two main systems, resource pools provided by workers and resource requirements specified by tasks. They will be described in turn below.

Worker resource pools

Worker resources are configured when a worker is started through so-called *resource pools*, which represent individual resource kinds (named sets of resource elements). The most common resource pools, such as CPU cores and their NUMA sockets, the amount of RAM and the presence of GPU accelerators, are detected automatically. On top of that, users can specify an arbitrary set of additional pools. There are two kinds of resource pools:

Indexed pool represents a set of resource elements where each individual element is non-fungible and has a unique identity (represented either by an integer or a string). An example of such a resource pool could be a set of CPU cores or GPU devices. For example, if a worker has two NVIDIA GPUs, that worker could provide an indexed pool containing two elements (e.g. with indices 0 and 1), each representing one individual GPU device. These two resources are not interchangeable; the scheduler decides which specific element will be assigned to a task and it tracks the individual elements based on their identity. If a task t is currently using GPU 0, that element will not be assigned to any other task until t has finished being executed.

Individual elements of an indexed pool can be further subdivided into *groups*, which mark some form of a relationship between specific elements. This can be used to represent *related resources*, e.g. NUMA domains and CPU sockets; a worker might provide a single indexed pool of CPUs with a separate group for each NUMA socket present on its node. Tasks can then specify if they want to prioritize drawing resources that belong to the same group; this will be described later.

When a worker is started, by default it automatically divides available CPU cores into groups according to their NUMA nodes. However, the concept of groups is general; they can be used for any indexed pool, not only specifically for NUMA.

The scheduler ensures that the *worker resource constraint with non-fungible resources* (Definition 9) is upheld for indexed pools.

Sum pool is designed for resources that are fungible. Such resources are typically numerous and it would thus be impractical to track each individual resource element separately. A sum pool does not represent its individual elements with indices; it simply defines a single number that specifies the amount of resources available in the pool. A typical representative of a sum pool is the amount of RAM available on a worker node. Modern HPC nodes can contain hundreds of GiBs of memory; it would be infeasible to treat each byte as a separate resource element. At the same time, tasks usually do not need such fine-grained tracking for these kinds of resources; if they care about memory requirements, they can just specify how much memory they need.

The scheduler ensures that the *worker resource constraint* (Definition 5) is upheld for sum pools.

Listing 7.3 shows a few examples of how users can configure resources when starting workers through the CLI.

Task resource requirements

Resource pools define the resources that are available on workers. The scheduler then uses *resource requirements* attached to individual tasks to decide on which workers a given task can be executed and which specific resources it will consume while it is running. Task resource requirements are defined for each task separately. By default, each task requires a single CPU core, but users can specify a different CPU requirement and also add additional requirements.

A resource requirement consists of a name of the resource that the task needs (the resource kind) and an amount specifying how many such resources are needed. For example, a task might specify that it requires two GPUs for its execution. The resource requirement always specifies the

```
# Start a worker with automatically detected resources
$ hq worker start

# Start a worker that manages 16 CPU cores
$ hq worker start --cpus 16

# Start a worker with additional resources
# Adds an indexed pool of four GPUs and a sum pool of 1024 resources (bytes)
$ hq worker start \
    --resource 'gpus/nvidia=[0,1,2,3]' \
    --resource 'memory=sum(1024)'
```

Listing 7.3: Configuring worker resources using the HyperQueue CLI

amount of required resources, regardless of whether the given resource is drawn from an indexed pool or a sum pool.

Task resource requirements in HyperQueue also support related resources, fractional resources, and dynamic resource selection (resource variants). These allow tasks to describe more complex use-cases related to resource management. The implementation specifics of these resource management features are described below.

Fractional resource requirements Tasks can specify their resource requirements in the form of a fraction. The scheduler uses fixed-point arithmetic to manage fractional resource requirements, in order to avoid issues associated with floating-point arithmetic, such as rounding errors. Section 7.5.6 will examine hardware utilization improvements that can be achieved with fractional resource requirements.

When a task has a fractional resource requirement that is drawn from an indexed resource pool, then the fractional part of the requirement is always allocated from a single resource element. For example, if a task requires 0.5 of a GPU resource, it will always be assigned a half of a single device, not e.g. a quarter of two separate devices. Furthermore, the integer part of a resource requirement is always allocated from a single resource element. If a task requires a single GPU, it will not ever be provided with e.g. a half of two separate devices. This approach avoids excessive fragmentation of individual resource elements, and it also makes it easier for tasks to use the allocated resources; some tasks might be unable to leverage more than one resource element of a given resource kind.

Resource variants Each task can define a list of resource requirement sets (resource variants); the scheduler uses the first one that can be satisfied by a worker at the time of scheduling. The order of the variants in the list determines their priority. The load-balancing improvements that can be achieved using resource variants will be evaluated in Section 7.5.7.

Related resources Relations between resource elements can be modeled using a combination of indexed pool groups and a *group allocation strategy*, which can be used to choose which specific resource elements from an indexed pool will be used for a given task. A typical example where indexed pool groups are useful are NUMA nodes, where tasks might want to be executed on cores

that belong to the same NUMA node (group). Tasks can use one of the following three strategies for each resource kind that they request:

Compact is the default strategy. The scheduler tries to allocate all requested resources from as few groups as possible. However, if there are not enough resources currently available for that approach on a given worker, it will also try to draw resource elements from a larger number of groups in order to satisfy the resource requirement.

Strict compact is a stricter version of the *compact* strategy. With this strategy, the scheduler first figures out the minimum number of groups required to satisfy the resource requirement. It will then schedule the task only after it can provide all requested resources from the smallest number of groups possible.

Scatter is the opposite of the *compact* mode. It tries to allocate resources for the task from as many groups as possible.

The effect of group allocation strategies will be evaluated in Section 7.5.5. Note that these strategies are not relevant for sum pools nor for indexed pools with a single group.

Furthermore, resource requirements can also be specified without an upper bound, i.e. a task can specify that it wants to consume all available resources of the specified pool. If a worker has at least a single resource element of the given pool available and the scheduler assigns a task with this kind of resource requirement to it, that task will be allocated all the pool's resources that are available when the task starts to execute. This can be used for software that is scalable w.r.t. the amount of resources provided to it.

There is also one additional special resource that is relevant for scheduling, called a *time request*. If a task specifies a time request (a duration) tr, it tells the scheduler that it needs at least tr time units for its execution. This is important for avoiding needless task restarts. Because workers started in allocations cannot run for a longer time than their allocation's wall-time (which is a known value), the scheduler knows the remaining lifetime of each such worker at any given time. This information is used for scheduling; if a worker has remaining lifetime l, the scheduler will not assign it tasks with a time request that is longer than l, because it is probable that such tasks could not finish executing before that worker shuts down.

Once a task is scheduled to a worker and the scheduler decides which resources will be allocated to it, it passes this information to the task through environment variables. For example, if a task asks for two CPU cores and the scheduler allocates cores 10 and 14 of a specific worker to it, HyperQueue will pass the environment variable `HQ_RESOURCE_VALUES_cpus=10,14` to the task.

It is important to note that HyperQueue treats resources as opaque and virtual; they are used only for scheduling. For example, if a task is provided with cores 0 and 1, the worker that executes it does not enforce that it will actually run on these specific cores. This responsibility is left to the task itself, which can use the provided environment variable to decide which resources it should actually use for its execution. For example, a task could pin its threads onto cores that were allocated to it. In the case of CPU cores, HyperQueue can pin the allocated cores for the task automatically if the task is configured as such.

Listing 7.4 displays a few examples of how can task resource requirements be configured using the CLI.

```
# Submit a task requiring 4 CPU cores
$ hq submit --cpus=4 ./my-program

# Submit 1000 tasks, each requiring 2 CPU cores
$ hq submit --cpus=2 --array=1-1000 ./my-program

# Submit a task requiring 2 CPU cores, pin CPU cores using OpenMP
$ hq submit --cpus=2 --pin=omp ./my-program

# Submit a task that requires 2 FPGA devices and needs at least 10 minutes to execute
$ hq submit --resource 'fpga=2' --time-request 10m ./my-program
```

Listing 7.4: Configuring task resource requirements using the HyperQueue CLI

Multi-node tasks

HyperQueue also supports a special kind of a resource that is especially useful for tasks that want to execute HPC applications designed to run on multiple nodes (e.g. MPI programs). A task may specify that it wants to be executed on multiple workers at the same time. In this case, HyperQueue assumes exclusive ownership of all the resources of each worker (node), which typically matches the requirement of MPI-like applications. A multi-node task requiring n nodes can only be scheduled when n workers are idle (not executing any tasks) at the same time. When a multi-node task starts to execute, HyperQueue passes it a *nodefile* which contains the hostnames of all workers allocated to the task. This nodefile can then be used by common MPI program launchers to start an MPI program.

Users that execute multi-node tasks running MPI applications might want to make sure that all workers (nodes) contributing in this computation are started in the same allocation, primarily for performance reasons (e.g. they might configure allocations to use nodes that reside close together in the networking topology of the cluster). By default, HyperQueue tries to adhere to this constraint. It uses an abstract concept of *worker groups*; multi-node tasks are always scheduled to workers belonging to the same group. At the same time, the group of a worker is automatically configured based its allocation (if there is any). This ensures that multi-node tasks are by default executed only on workers that reside within the same allocation. However, it is possible to override the group of a worker and e.g. assign the same group to all workers. In that case, workers from all active allocations can participate in executing multi-node tasks.

7.3.5 Fault tolerance

The execution of a task in HyperQueue can end in a failure because of two main reasons. The worker that executes the task might crash, be stopped by the user, or an allocation inside which it is executed can run out of its wall-time. In that case, the task will be rescheduled by the server

to a different worker that upholds its resource requirements. This is made possible due to the fact that tasks are not tied to any specific computational resource or allocation. When such a task restart happens, it is important to let the executed program know that it is not being executed for the first time, so that it can react to it (for example clean up files that were created during a previous execution of the same task or restore execution from a previously created checkpoint). This is achieved using an *instance ID*. It is a non-negative number attached to each task that is increased every time the task is restarted. This number is then passed to the executed task so that it can decide whether it should react to the restart in any specific way.

In rare cases, the crash of the worker might be caused by the task itself, e.g. if it allocates too much memory and causes an OOM (Out Of Memory) failure. If this happens repeatedly, such task could hamper the execution of the whole task graph, since it would be repeatedly crashing workers. For that reason, it is possible to configure the maximum number of worker crashes per task (it is configured to a low number by default). If a task exceeds this number (i.e. it is present during too many worker crashes), it will be automatically canceled.

The second reason why a task might fail is that its computation simply does not succeed, for example due to the executed process returning a non-zero exit code. In that case, the task is marked as failed and any tasks that (transitively) depend on it are canceled. HyperQueue allows users to filter tasks by their state and resubmit only failed or canceled tasks in a straightforward way, which is possible due to the job state being kept outside of transient allocations. Listing 7.5 demonstrates how users can use the CLI to resubmit failed tasks from a previously submitted job. It also shows that several HyperQueue commands are designed to be composable, adhering to the Unix philosophy of creating commands that do a single thing well and that can be easily combined together.

```
# Submit task array with 1000 tasks, wait until it completes
$ hq submit --array=1-1000 --wait ./my-script.sh

# Find task IDs that have failed in the last job
$ FAILED_TASKS=$(hq job task-ids last --filter=failed)

# Resubmit only the failed tasks
$ hq submit --array=${FAILED_TASKS} ./my-computation
```

Listing 7.5: Handling task failure using the HyperQueue CLI

Too many tasks failing can signal the presence of some catastrophic condition, e.g. the user failed to properly configure their program, forgot to load some required runtime dependencies or used the wrong filesystem path for inputs. If this happens for a very large job, it could lead to a lot of wasted computation, where tasks will start to execute, but they will probably fail very soon after that. HyperQueue thus allows users to configure a parameter that specifies the maximum number of task failures per job. If the configured number of failures is exceeded, the whole job will be preemptively canceled to avoid wasting computation.

A special case of a failure is when the server itself crashes or it is stopped by the user or when a worker loses connection to the server due to networking issues. In this case, the worker cannot

reasonably continue executing long-term without having a communication channel with the server. However, if it manages tasks that are already being executed, it could be wasteful to stop their computation just because the connection to the server has been lost. The worker can thus be configured in a way that it will finish executing such running tasks before shutting itself down. However, the completion of these tasks will not be reported back to the server (as the worker cannot reach it anymore), so users will need to process their outputs manually.

It is possible to configure the server to store a journal containing all necessary information about its task database to the filesystem. The server can then be restored from this journal in case of a crash or a failure, so that it can resume computation of previously unfinished tasks.

7.3.6 Automatic submission of allocations

The meta-scheduling design employed by HyperQueue resolves most of the challenges associated with using allocation managers for executing tasks, since it fully automates the mapping of tasks to allocations. It also makes creating allocations simple, so that users can scale the computational resources used for computing their task graphs at their will. However, creating allocations manually can still be relatively demanding for users, who might want to employ a more automated approach for scaling computational resources. Since HyperQueue knows the state of all tasks and workers and it is commonly executed on login nodes that have access to allocation managers, it is natural to provide it with the ability to automatically submit new allocations on behalf of the user, based on current computational load. This system has been implemented in HyperQueue under the name *Automatic allocator* (shortened as *autoalloc*). The automatic allocator was created with the following design goals:

Allow computational resources to scale up At any given moment, if there are tasks that are waiting to be executed and there are no free computational resources, *autoalloc* should attempt to add more resources by creating new allocations. At the same time, it should respect backpressure from the allocation manager to avoid overloading it.

Allow computational resources to scale down Keeping allocations running on a supercomputer can be quite costly. *Autoalloc* should thus shut down allocations that are not performing any useful work (because they do not have anything to compute) in order to avoid wasting resources.

Be flexible Allocation managers typically provide various configuration knobs that can affect the behavior of allocations. While it will not ever be possible to support all possible implementations of allocation managers out-of-the-box, *autoalloc* should provide extension points that would enable users to configure the submitted allocations to their liking.

Autoalloc has been implemented as a background service within the HyperQueue server. To use it, users first need to create at least one *allocation queue*. Such a queue describes a template for creating new allocations that will be submitted when there is a demand for more computational resources. Each queue contains several properties that can be configured by users:

Allocation manager *Autoalloc* needs to communicate with a given allocation manager to be able to submit allocations into it. Users thus need to tell HyperQueue which allocation manager it should communicate with. Currently, two managers are supported, Slurm and PBS.

Time limit This value determines the wall-time of each allocation submitted by *autoalloc*. Knowing the maximum duration of each allocation helps it decide when it makes sense to create allocations. For example, if the only tasks waiting to be computed have a *time request* that is longer than the time limit of a given allocation queue, it does not make sense to create allocations in this queue, because workers started in these allocations could not execute such tasks anyway.

Backlog This parameter specifies the maximum number of allocations that can be queued in the allocation manager at any given time. It is designed to ensure that the automatic allocator does not overload the allocation manager.

Worker count per allocation Determines how many nodes (workers) should be requested for each allocation. Unless users want to execute multi-node tasks, it is usually convenient to use the default value (1); typically, the fewer nodes are requested in an allocation, the sooner it will be started by the allocation manager.

Max worker count This parameter can be used to limit the maximum number of running workers across all allocations started by this queue.

Idle timeout Time after which an idle worker (a worker that does not have any tasks to compute) will stop. The idle timeout mechanism helps to avoid resource waste by stopping workers that do not have anything to compute anymore. By default, this timeout is set to a few minutes.

Worker resources Users can override the resources assigned to each worker started by the given allocation queue. This can be used if the user knows that the allocations will be started in a specific partition of the cluster that contains resources that cannot be automatically detected by HyperQueue (e.g. FPGA devices).

Custom allocation parameters Users can also specify arbitrary command-line parameters that will be passed to the submit command sent to the corresponding allocation manager, like the computational project that should be used or the cluster partition where the allocation should be created. To make debugging easier, HyperQueue will by default submit a test allocation immediately after the allocation queue is created, in order to test that the specified allocation parameters can be handled by the manager. If this test allocation is successfully created, it is then immediately canceled to avoid wasting computational resources.

Listing 7.6 shows several examples of commands for creating allocation queues and observing allocation state using the HyperQueue CLI.

```
# Create an allocation queue for Slurm
$ hq alloc add slurm --time-limit 1h -- -Aproject-1 -pcpu_partition

# Create an allocation queue for PBS with GPU resources
$ hq alloc add pbs --resource "gpus/nvidia=range(0-1)" ...

# Display information about allocations submitted by the given queue
$ hq alloc info 1
```

Listing 7.6: Configuring automatic allocation using the HyperQueue CLI

Autoalloc is a reactive system. It observes events from the server (such as new jobs being submitted or new workers being connected) and manages allocations in response to these events. If it encounters a situation where there are tasks that are in the waiting state and no existing workers can execute them (either because there are no workers or all of them are fully occupied), it will start submitting allocations to the allocation manager. It will submit allocations up until the configured *backlog* value or until the allocation manager applies backpressure (whichever comes sooner). Note that *autoalloc* does not limit the total number of allocations that are "in-flight" (queued or running) by default, only the number of queued allocations. This allows it to potentially create a large number of allocations and thus make use of all available resources of the cluster if the allocation manager allows it. Users can use other configuration parameters (such as the total number of workers) to limit the amount of submitted resources. *Autoalloc* also contains an internal rate limiter that makes sure that it does not submit allocations too quickly and that it pauses the allocation process if the allocations start failing too often.

HyperQueue communicates with PBS or Slurm using their standard commands, such as `qsub` or `sbatch`, as they do not provide a completely machine-readable API. This means that in order to use automatic allocation, the server has to be executed on a node that has access to these commands and can communicate with the manager. If this is not possible, a proxy service could be used to forward communication between the node with the server and a node where the manager is deployed. HyperQueue implements communication with each manager in a custom way. While there have been some attempts to create a unified interface [153, 154] for submitting allocations to different allocation managers, they have not seen wide adoption so far. Furthermore, adopting these approaches would require HyperQueue to introduce a dependency on Python or an external HTTP (HyperText Transfer Protocol) service, which would make it more difficult to deploy.

It is crucial for *autoalloc* to maintain an up-to-date state of its created allocations, so that it can report their state to the user and also so that it knows if it should submit new allocations. Originally, it was using a polling approach to determine the latest state, where it was repeatedly querying the allocation manager about the state of allocations. However, this turned out to put a lot of pressure on the allocation managers, which were not very optimized for this use-case, especially if the polling frequency was very high (every few seconds). Some allocation managers are even configured in a way where they cache the state of allocations when they are queried too often; this reduces the chance of them being overloaded, but it also means that HyperQueue would not get the latest data, which is not ideal. However, we can observe that it is not actually required to poll the allocation manager. The only allocation events that are important for the allocator are the start and an end of an allocation. And this information can be learned through a proxy; the connections and disconnections of workers. When a worker is started in an allocation, it will connect to the server and tell it its allocation ID. This lets the automatic allocator learn about new allocations. On the other hand, when such a worker disconnects, it signifies that its containing allocation has also finished. Using this approach, the allocator is able to maintain the latest state of allocations without explicitly querying the allocation manager.

Autoalloc attempts to take certain task properties into account when creating allocations. For example, if a task has a *time request* of two hours, but a configured allocation queue only has a time

limit of one hour, the allocator will not submit allocations using this queue to provide resources for such a task, because workers started in these allocations could not compute such a task anyway. It should be noted that in general, it is very difficult to guess when and how many allocations should be created, because the allocator does not know in advance which tasks will be submitted in the future, nor for how long the currently submitted tasks will run. It also cannot precisely predict how long its submitted allocations will stay in the allocation manager queue. As a future extension, the allocator could be extended with a prediction of allocation queuing times [155] or with a prediction of task execution times [156].

7.4 Use-cases

This section describes several use-cases where HyperQueue has been successfully used to execute task graphs on HPC systems. Apart from these selected case studies, it has also been used in other projects and scenarios; these will be enumerated in Chapter 8.

The first two use-cases originate from the LIGATE (LIgand Generator and portable drug discovery platform AT Exascale) [4] project, whose goal was to integrate state-of-the-art CADD (Computer-aided Drug Design) tools into a unified platform designed for executing drug design experiments on petascale and exascale HPC clusters. HyperQueue was created within the LIGATE project; its use-cases served as one of the main driving forces behind the design of this task runtime. The usage of HyperQueue within the LIGATE project is also briefly described in [4].

7.4.1 Virtual screening and free-energy calculations

One of the MD (Molecular Dynamics) frameworks that were used in the LIGATE project was LiGen [157, 158], a suite of utilities that can be used for performing virtual screening. Virtual screening is a computational method for identifying molecules that can bind to drug candidates, which is used for computer-driven drug discovery.

LiGen was used to implement a high-throughput virtual screening pipeline, whose goal was to assign scores to a large number of input ligands and then perform molecular docking for the most promising ligand candidates. The primary input for the workflow is a set of ligands in the SMILES [159] format, where each line represents a single ligand. LiGen then expands each ligand into a 3D representation stored in a Mol2 [160] format and performs virtual screening to assign a score to it. The resulting ligand-score pairs are then stored in a CSV file. Finally, the most promising N candidates (where N is a configurable parameter) are selected based on their scores and molecular docking is then performed on them.

The virtual screening workflow has been implemented using HyperQueue, in order to leverage its heterogeneous resource management capabilities. It was designed using the Python API, due to its support for task dependencies and also because it was natural to implement several utility tasks within the workflow in Python rather than e.g. in Bash. The workflow consists of both very short running tasks (such as ligand expansion) and also more computationally demanding tasks (such as scoring and docking). The scoring and docking stages can also run on GPU accelerators, which provide much higher throughput than the CPU implementation. These constraints were encoded

using task resource requirements, which allowed selecting the proper hardware resources for each kind of task. Section 7.5.9 describes an experiment that measures the achieved hardware utilization when running this workflow with HyperQueue.

This workflow was executed on four computational sites; the Karolina [161], LUMI [162], LEONARDO [163] and E4 [164] HPC clusters. The only change required to port the workflow to a new cluster was to update the Slurm credentials used for submitting allocations. This demonstrates the simplicity of porting HyperQueue workflows to different supercomputers.

The output of the scoring and docking stages of the virtual screening workflow then serve as input for a more complex CADD workflow that is designed for performing free-energy calculations that estimate the RBFE (Relative Binding Free Energy) of ligands and protein-hybrid ligand complexes using AWH (Accelerated Weight Histogram) calculations [165]. The CADD workflow uses the GROMACS MD framework [32] along with tens of other bioinformatics tools and packages and is structurally quite complex, as it consists of tens of different Bash and Python scripts that depend on the outputs of one another. This workflow has been also implemented using the HyperQueue Python API.

The source code of the LiGen virtual screening and the CADD workflows is publicly available [166].

7.4.2 Pose selection

The *Pose selection* workflow was one of the supporting workflows of the LIGATE project. Its purpose was to generate a large training dataset that would then be used to train machine-learning models designed to improve the accuracy of protein-ligand complex pose scoring and docking performed by the LiGen virtual screening tool suite.

A single task of the workflow estimates the stability of an individual pose of a protein-ligand complex (a drug candidate). It also leverages GROMACS [32] for simulating a short (100 ps) trajectory for each pose, from which a final ABFE (Absolute Binding Free Energy) value is calculated. Because this computation is relatively lightweight (it only lasts for a couple of minutes), it was possible to run it on a large dataset of inputs. It was executed for more than four thousand complexes from the PDBbind [167] v2020 database; for each complex, up to 256 different poses were evaluated, and for each pose, 8 simulation replicas were executed to reduce statistical error (since the computation is randomized). This resulted in more than 6.5 million GROMACS simulations that had to be performed in total.

The workflow itself is not structurally complex; each task is independent and there are no dependencies between tasks. However, due to the sheer amount of simulations that needed to be performed, it would be quite challenging to execute such a large number of tasks directly using Slurm allocations. HyperQueue was thus used to define the computational task graph and manage the executions of all the simulations. Each task executed all eight replicas (independent simulations) for a given complex pose, either on 8 GPUs or 16 CPU cores in parallel. HyperQueue thus had to schedule and manage approximately 800 thousand tasks in total. The automatic allocator took care of automatically submitting Slurm allocations, users of the workflow thus did not need to concern themselves with creating them manually.

Another benefit of using HyperQueue for this workflow was its built-in support for fault-tolerance. With such a large number of tasks, some of them have not finished successfully, either because of a Slurm allocation ending in the middle of a computation or because of an instability of the used HPC cluster. Thanks to HyperQueue, such failed tasks were transparently rescheduled to different computational nodes without any user intervention. This was possible due to the fact that the server was executed on a login node and thus it kept the state of the submitted tasks in a persistent database that was not affected by shutdowns or failures of Slurm allocations.

The execution of this workflow has consumed a non-trivial amount of both CPU and GPU computational time. Concretely, its execution has used 240 thousand GPU hours on *LUMI-G* (the GPU partition of the LUMI [162] cluster) and 2 million CPU core hours on the MeluXina [168] cluster. This demonstrates that HyperQueue is able to scale to large use-cases and enables robust execution using a significant amount of computational resources. To our knowledge, the execution of this workflow was among the largest MD campaigns that were ever computed on HPC resources[3].

This use-case has benefited from several interconnected HyperQueue features. Due to its low overhead, it was possible to execute and schedule a very large number of tasks. These tasks were automatically meta-scheduled to many different allocations, respecting their (CPU and GPU) resource requirements. Thanks to the automatic allocator, all required Slurm allocations were submitted automatically in response to current computational load. And HyperQueue also automatically re-executed any tasks that have failed to successfully finish. Both the source code [169] and the datasets [170, 171] with the computed results of the Pose Selector workflow are publicly available.

7.4.3 Hyperparameter search

HyperQueue was used in the EXA4MIND [172] project to parallelize and distribute independent instances of a hyperparameter search workflow on the GPU partition of the Karolina cluster [161]. Hyperparameter search is a technique where a machine-learning model is trained multiple times with separate values for various hyperparameters (such as learning rate or batch size). The goal is to find a configuration of hyperparameters that produces a model with the highest possible prediction performance.

Parallelizing a hyperparameter search using HyperQueue is very simple, even using the CLI. The individual configurations can be stored either into a JSON array (where each configuration is a JSON object) or into a text file (where each configuration is a separate line), from which HyperQueue can then automatically build the workflow. Listing 7.7 shows an example of how easy is to use HyperQueue in this case. The user simply starts the server, configures automatic allocation, so that it starts allocations on their behalf, and then submits a job that is automatically generated from a JSON file containing the hyperparameter configurations. The executed script does not know anything about HyperQueue apart from reading its assigned input configuration from an environment variable set by the corresponding worker.

[3]This claim can also be found in LIGATE Deliverable D7.1, which has not been published yet at the time of writing of this thesis.

```
# Start the HyperQueue server
$ hq server start &

# Configure automatic allocation
$ hq alloc add slurm --time-limit 48h -- --partition=qgpu --gpus=1 --account=...

# Submit the computation. Each training requires a single GPU and 16 CPU cores
$ hq submit --cpus=16 --resource="gpus/nvidia=1" --from-json configs.json train.sh
```

Listing 7.7: Hyperparameter search using HyperQueue

7.4.4 Efficient load balancing of opportunistic computations

The CERN ATLAS [173] experiment generates vast amounts of data from its Large Hadron Collider, which are then further postprocessed using various simulations. Some of these simulations are executed on Czech supercomputing resources through the ARC-CE (Advanced Resource Connector Compute Element) [174] submission system. ARC-CE periodically connects over the network to IT4Innovations HPC clusters (such as Karolina) and submits allocations that perform the desired simulations. Its goal is to opportunistically leverage free cluster resources that are available when no other allocations with a higher priority can run at the same time.

Originally, ARC-CE was submitting simulation computations directly as PBS allocations[4]. While this approach was working, it was also running into some of the challenges presented in Section 4.1; notably, the submission system was hitting the limits of the maximum allowed number of allocations that could be submitted by a single user. Furthermore, it was encountering problems caused by communicating with the allocation manager too frequently [175], which we also had to deal with during the implementation of the HyperQueue automatic allocator, as was described in Section 7.3.6. In order to resolve issues with allocation manager communication and to improve hardware utilization, the users of ARC-CE have switched to using HyperQueue, which has provided two primary benefits.

The first benefit is that computation submission becomes much easier for ARC-CE. It simply submits its computations into a HyperQueue server running on the login node of Karolina, without having to deal with Slurm allocation count or communication frequency limits. It also configures the automatic allocator to submit allocations into several Slurm queues; the rest is handled fully automatically by HyperQueue.

The second advantage of using HyperQueue is an improved utilization of nodes. Previously, the computations were performed on the Salomon supercomputer [136], which had 24 CPU cores per node. This was a reasonable amount of cores for computing a single simulation per each node. However, once Salomon has been decommissioned, ARC-CE started submitting computations to Karolina [161], which has 128 cores per node instead. This resulted in a decrease of CPU utilization, because each simulation spends a non-trivial amount of time performing either sequential computation or I/O, during which only a few cores were utilized. And with so many cores available on the

[4]Note that the IT4Innovations clusters originally used PBS as their allocation manager; they have switched to Slurm in 2023.

node, this resulted in lower hardware utilization, as many cores remained idle. A simple method of improving utilization is to split the computation into smaller parts that each use a smaller number of cores. However, on the Karolina CPU partition, it is not possible to ask for less than a whole node directly using Slurm. This is where HyperQueue came in; once ARC-CE switched to it, the computations were configured for 32 cores and HyperQueue then took care of packing four such computations to each allocated Karolina node. This change resulted in an improvement of the average CPU utilization of the submitted computations on Karolina from 70% to 90%, which saves tens of thousands of computational node hours each year [176].

This use-case shows that HyperQueue can also be used programmatically by automated tools, not only manually by cluster users. It provides a machine-readable JSON output mode for its CLI commands that facilitates this kind of usage.

7.5 Evaluation

This section presents several experiments that evaluate the performance and ergonomic aspects of HyperQueue. They focus on its overhead, scalability, hardware utilization improvements achievable through its heterogeneous resource management techniques, CPU time consumption on login nodes, and also its automatic allocation and output streaming features. We also examine the HyperQueue virtual screening workflow and compare the scalability and overhead of HyperQueue and Dask

All experiments were performed on the CPU partition of the Karolina supercomputer [161] located at the IT4Innovations supercomputing center [135]. Each non-accelerated computational node of Karolina has two AMD EPYC™ 7H12 2.6 GHz 64 core CPUs, for a total of 128 cores per node (hyper-threading is disabled), and 256 GiB of DDR4 RAM. It runs on the RockyOS 8 operating system with Linux kernel 4.18.0 and `glibc` (GNU `C` standard library implementation) 2.28. All experiments were performed with HyperQueue version 0.18, compiled with Rust 1.79.0. The source code of the presented experiments is available in a GitHub repository[5].

In selected experiments, HyperQueue was benchmarked in a so-called *zero worker* mode, which is very similar to the zero worker mode that has been used to benchmark the overhead of RSDS, described in Section 6.2.3. In this mode, workers do not actually execute any tasks; when a task is about to be executed on a worker, it instead finishes immediately. This mode can be used to benchmark the inner overhead of HyperQueue, without it being affected by external factors, such as the actual execution of tasks. Experiments performed with this mode will be marked explicitly.

Unless otherwise noted, each benchmark was executed and measured three times. In all experiments, the server and the workers of HyperQueue (and Dask) were executed on separate nodes. Many experiments use the term *makespan*, which describes the duration from submitting a task graph (a HyperQueue *job*) until all tasks of that task graph are finished computing (as defined in Definition 6). We will also use the term *simple workflow*, which denotes a task graph without any dependencies. Unless stated otherwise, each task of such a workflow has a resource requirement of a single CPU core and runs for a fixed duration by executing the `sleep` UNIX program.

[5] `https://github.com/it4innovations/hyperqueue/tree/benchmarks/benchmarks`

7.5.1 Total overhead

In this experiment we evaluate the total *overhead* of HyperQueue. Under an ideal scenario, there exists a minimal makespan in which a given task graph can be executed on a fixed set of computational resources, assuming an infinitely fast communication network and no delays between executing individual tasks. We define overhead as any additional time on top of this ideal makespan, which is induced by the task runtime (in this case HyperQueue). This overhead is caused by network communication, scheduling and various forms of bookkeeping performed by a task runtime.

The overhead will of course depend on the task graph that is executed and the computational resources (workers) used for its execution. Nevertheless, it is interesting to examine the minimum overhead introduced by the task runtime on a trivial task graph, in particular to gain insight into the smallest duration of tasks that can be used without the overhead becoming too limiting.

We have benchmarked the *simple workflow* in two sizes (10 and 50 thousand tasks) with three different worker counts (1, 2 and 4 workers) and four different task durations (1 ms, 10 ms, 100 ms and 250 ms). The *ideal duration* for each task graph was estimated by taking the total time to execute all the tasks sequentially (task count × task duration) and dividing it by the total number of cores available (worker count × 128). This estimates a perfect scenario without any overhead, where all tasks are perfectly parallelized across all available CPU cores.

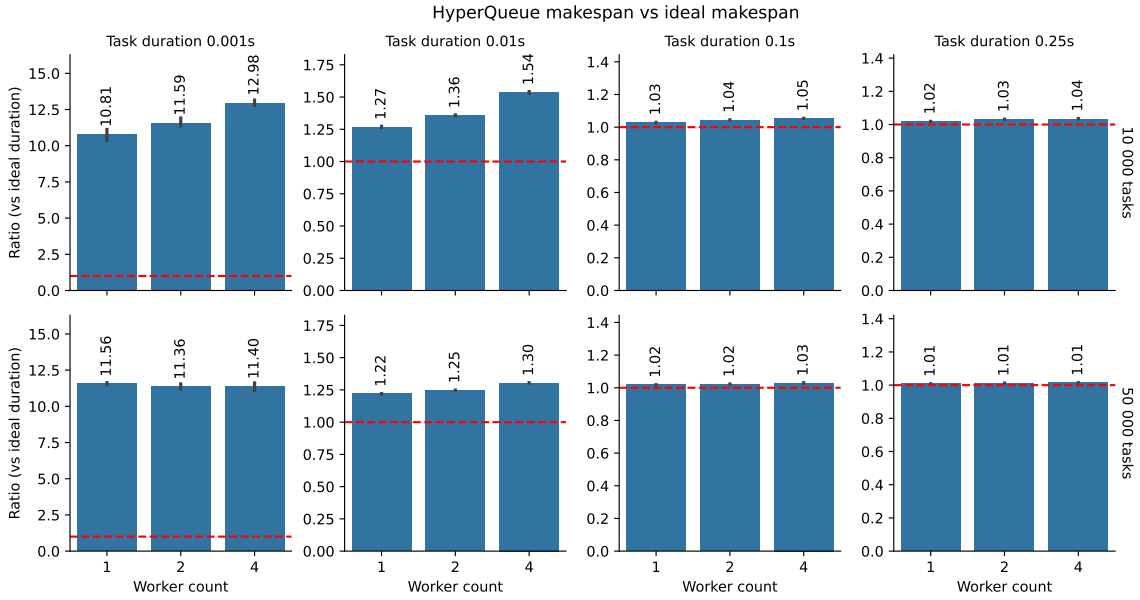


Figure 7.4: Total overhead of HyperQueue (ratio vs theoretically ideal makespan)

The results of this experiment can be observed in Figure 7.4. Each chart column shows a different task duration; the rows separate different task counts. Within each chart, the horizontal axis represents the number of used workers, while the vertical axis shows the duration needed to execute the task graph in HyperQueue, normalized to the theoretical ideal duration. The ideal

duration is marked with a red horizontal dotted line. For example, the value 1.05 specifies that the HyperQueue execution was 5% slower than the ideal duration.

We can see that for tasks that last for 100 ms or more, the overhead of HyperQueue is within 5% of the ideal makespan; with the larger task graph that contains 50 thousand tasks, the overhead stays within 3%. The overhead increases slightly with an added number of workers, which is expected; the scheduler needs to perform more work and the server also communicates with more workers over the network.

The situation becomes more interesting in the case where the task duration is only 1 ms. Here the overhead of HyperQueue seemingly becomes very large. We have examined this situation in detail and found out that the issue is caused mostly by slow command execution performance on Karolina. Our calculation of the theoretically ideal makespan assumes that the executed command will last for exactly 1 ms. However, through several benchmarks, we found out that running even the shortest possible process (that immediately exits) on Karolina takes *at least* 500 µs, and executing a program that is supposed to sleep for 1 ms can take several milliseconds. Because the overhead of actually executing the command is so large, it skews the total overhead of HyperQueue, as it is not able to reach the theoretically ideal makespan, because the execution of commands takes more time than expected. Note that this would be an issue for any task runtime that executes a command in each task, and there is not that much that HyperQueue could do to avoid this overhead. We eventually found out that the slow process spawning performance is caused primarily by an old version of the `glibc C` standard library implementation used on Karolina; there is not much that we could do about that.

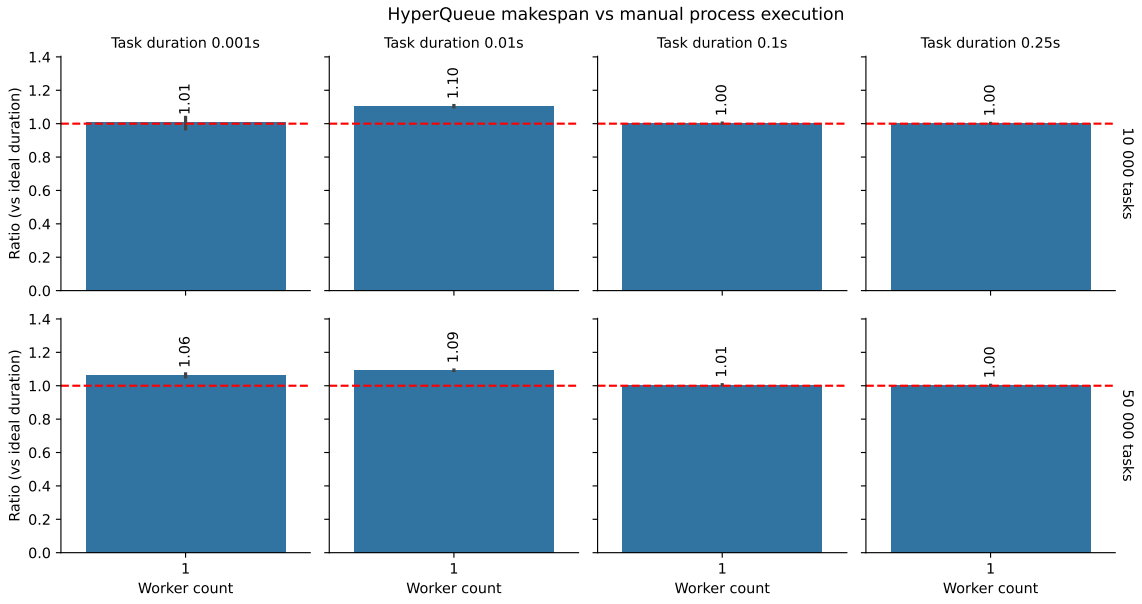


Figure 7.5: Total overhead of HyperQueue (ratio vs manual process execution)

To provide a fairer evaluation of the overhead of HyperQueue for short tasks, we have also evaluated it against a different baseline, which is not based on a theoretical calculation, but on the

actual performance of command execution on Karolina. We have implemented a simple program that executes the same tasks (`sleep` processes) as HyperQueue, in parallel on all 128 available cores. This provides a more reasonable baseline that takes into account the overhead of command execution on Karolina. The results of this experiments can be seen in Figure 7.5. The baseline is marked with a horizontal red line; it represents the fastest measured time to execute the given number of processes. We have performed this experiment only with a single worker, because our baseline program did not implement multi-node distribution. Based on these results, we can see that for the *simple workflow*, the actual overhead of HyperQueue (when compared to manually running the commands without a task runtime) stays within 10% even for very short tasks.

7.5.2 Task throughput

The previous experiment has evaluated the total overhead of HyperQueue while taking into account the time to execute tasks. To further examine the inner overhead of scheduling and network communication, in this experiment we evaluate the possible task throughput (number of tasks processed per second) using the *zero worker* mode. In this mode, HyperQueue performs all operations that it does normally (managing tasks and workers, scheduling, sending network packets), but workers do not actually execute tasks; the corresponding worker immediately marks each task as completed instead. This allows us to examine the overhead of HyperQueue without it being affected by process spawning, which can have severe overhead, as was demonstrated in the previous experiment. We have benchmarked the *simple workflow* containing from 10 thousand to 1 million tasks with varying worker counts (up to 16 workers and thus 2048 CPU cores).

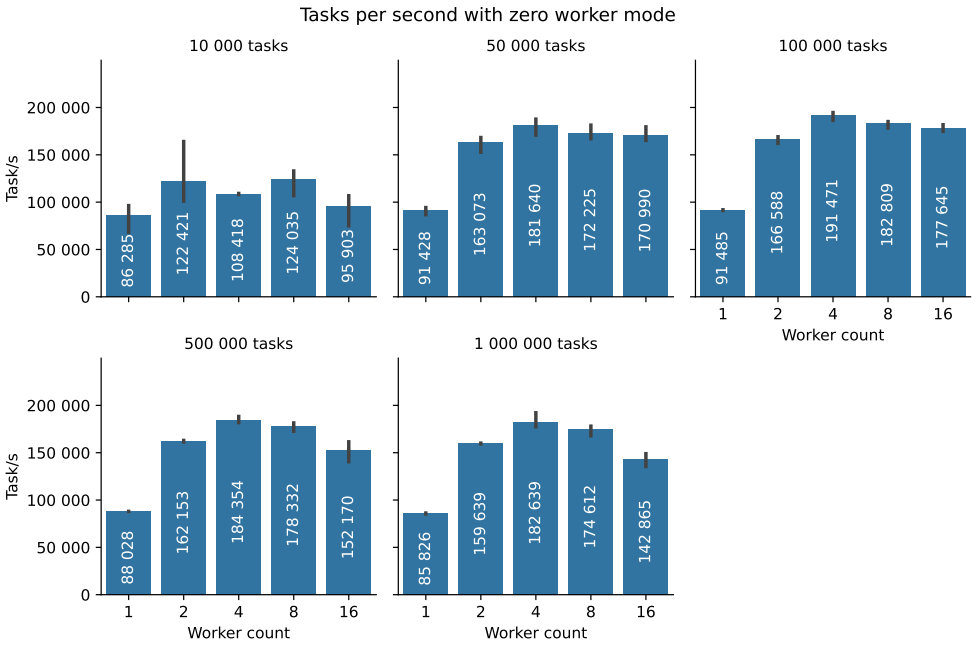


Figure 7.6: Tasks processed per second with *zero worker* mode

The results can be observed in Figure 7.6. The horizontal axis displays the number of workers in the task graph, while the vertical axis displays the achieved throughput measured in tasks processed per second. In theory, the throughput should be increasing with more added workers. For the smallest task graph with 10 thousand tasks, the throughput increases up to four workers (512 cores), then it decreases; the task graph is too small and the constant overhead factors start dominating over the available parallelism. For task graphs with up to 100 thousand tasks, the throughput also increases up to four workers, where it saturates. For even larger task graphs, the throughput once again starts decreasing with more than four workers; the overhead of network communication and scheduling starts to dominate.

The absolute numbers of the achieved throughput demonstrate that HyperQueue has very little baseline overhead, and can in theory execute hundreds of thousands of tasks per second. The tasks would have to be shorter than approximately 5 µs to fully saturate the throughput achievable by HyperQueue. Such a task duration is uncommon for scientific workflows (indeed, on Karolina even starting a new process takes a hundred times longer); it is more common for task-parallel programming models that have been described in Section 2.2.3, which are operating on a very different level of granularity than what HyperQueue was designed for. This task throughput is orders of magnitude higher than throughput measured for the Parsl, FireWorks and Dask task runtimes, as reported in [75].

As you may recall from Section 6.2.3, the overhead per task of Dask for a task graph with 50 thousand tasks and a single node (using the *zero worker* mode) was approximately 0.3 ms. The per-task overhead of HyperQueue (calculated as an inverted value of the number of tasks executed per second) for a similar configuration evaluated in this experiment is approximately 0.01 ms, an order of magnitude lower.

7.5.3 Strong scaling

This experiment evaluates the ability of HyperQueue to scale to many computational resources, by executing the same task graph with increasing worker counts (up to 64 workers and thus 8192 cores in total). Note that each benchmark configuration in this experiment was measured only once because of the large amount of computational resources required to allocate tens of worker nodes.

We have designed two separate scenarios for this experiment. In the first one, we have selected a fixed target makespan, so that the *simple workflow* would be executed in approximately 5 minutes (300 seconds) on a single worker, and benchmarked three task graphs with increasing task counts. This allowed us to examine how the scalability of HyperQueue changes based on the number of tasks in the task graph when the total computational workload stays the same. The task duration of each task was scaled accordingly for each graph, so that the total makespan on a single worker would be 5 minutes. The benchmarked worker counts were 1, 2, 4, 8, 16, 32 and 64.

The results of the benchmark with a fixed makespan can be observed in Figure 7.7. The horizontal axis displays the number of workers used for executing the task graph. In the top row, the vertical axis shows the measured makespan; in the bottom row, the vertical axis shows speedup against the measured makespan on a single worker. We can see that for all three measured task durations, HyperQueue was able to scale reasonably well up to 64 workers, where it was able

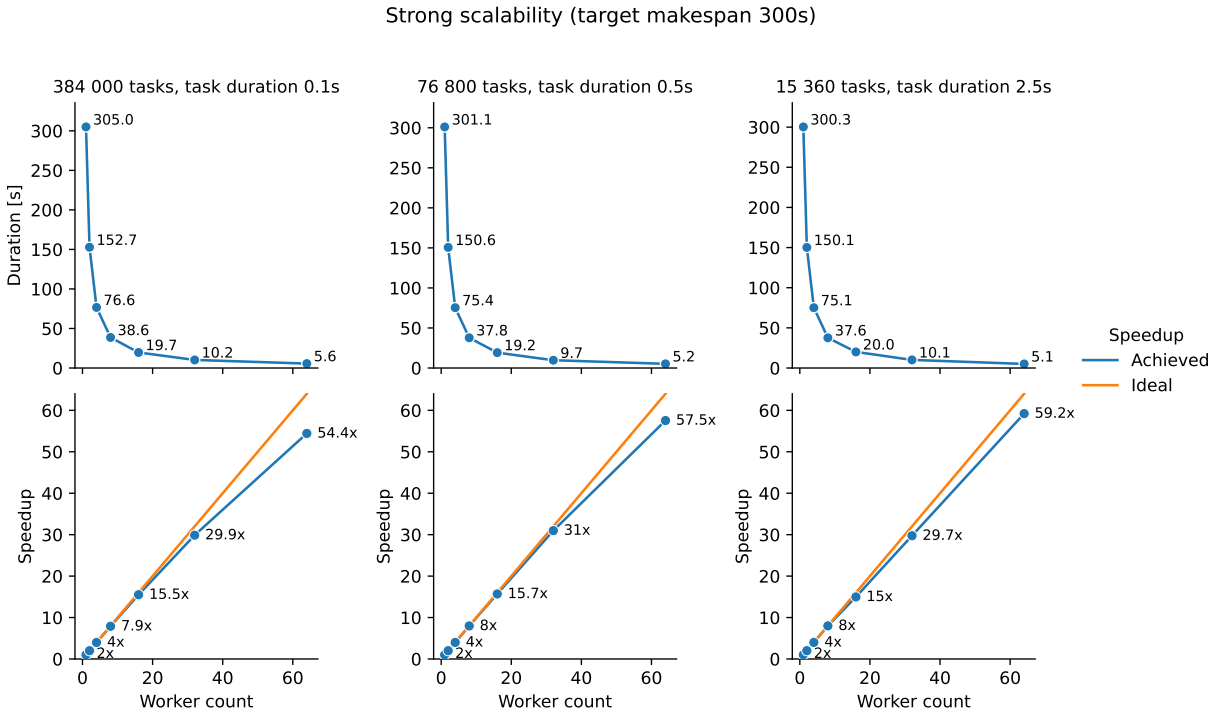


Figure 7.7: Strong scalability of HyperQueue with a fixed target makespan (300 s)

to compute the whole task graph in approximately 5 seconds. In the best case, with tasks that took 2.5 s to execute, it provided a 59.2x speedup over a single worker using 64 workers. In the worst case, with tasks that lasted 100 ms, it provided a 54.4x speedup over a single worker using 64 workers. With task duration 0.1 s and 64 workers, HyperQueue was able to dispatch almost 70 thousand tasks per second.

In the second scenario, we fixed the task duration of each task to 1 second and then varied the task count. This allowed us to examine how the strong scalability of HyperQueue changes with an increasing number of tasks.

We can observe the results in Figure 7.8. We can see that the scalability improves with the number of tasks; with 100 thousand tasks, HyperQueue is able to provide a 59x speedup with 64 workers. On the other hand, with only 10 thousand tasks, the speedup in this case reaches only 38.8x, because there are not enough tasks to saturate all workers. With 64 workers that each have 128 cores, the total number of cores managed by HyperQueue is 8192. Therefore, each worker barely gets a single task to compute in this small task graph.

In general, the results of our scalability and overhead experiments indicate that HyperQueue introduces very little overhead and is able to scale to a large amount of computational resources (tens of nodes and thousands of cores) and also to large amounts of tasks (hundreds of thousands) without an issue.

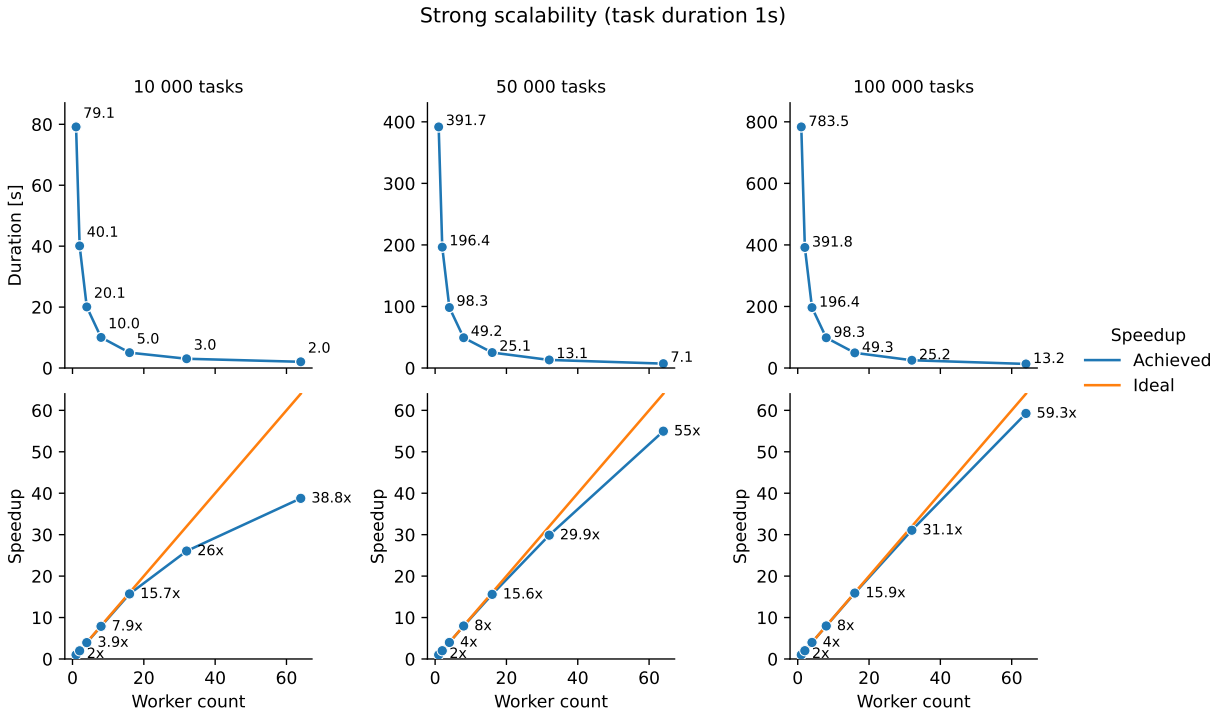


Figure 7.8: Strong scalability of HyperQueue with a fixed task duration (1 s)

7.5.4 Performance comparison of Dask and HyperQueue

This experiment evaluates basic performance differences between the Dask task runtime and HyperQueue. We have evaluated *dask/distributed* 2024.7.0 running under CPython 3.10.8 with a disabled monitoring dashboard for improved performance. All measurements in this experiment were performed only once, to reduce the required computational resources.

We have evaluated the scalability of both runtimes on the *simple workflow* with a target makespan set to 30 s on a single worker, with an increasing number of workers and a varying number of tasks. In this scenario, Dask was benchmarked in three separate worker configurations, with 1 process per node and 128 threads per process ($1p/128t$), with 8 processes per node and 16 threads per process ($8p/16t$) and with 128 processes per node and 1 thread per process ($128p/1t$). The reason for choosing different process/thread combinations has been extensively explained in Section 6.2.3; we chose two extremes (1 process and 128 processes) and then a compromise with 8 processes per node, according to recommendations in the Dask documentation [177].

The results of this experiment are displayed in Figure 7.9. The individual charts display separate task graphs with varying task counts (in each case, the task graph should have a makespan of 30 seconds on a single node). The horizontal axis shows the amount of used nodes and the vertical axis displays the makespan. Note that we are using the term *node count* instead of *worker count* here, because each Dask process spawned per node is technically a separate worker. We can see that the in the case of Dask, the performance depends a lot on the number of tasks in the task graph

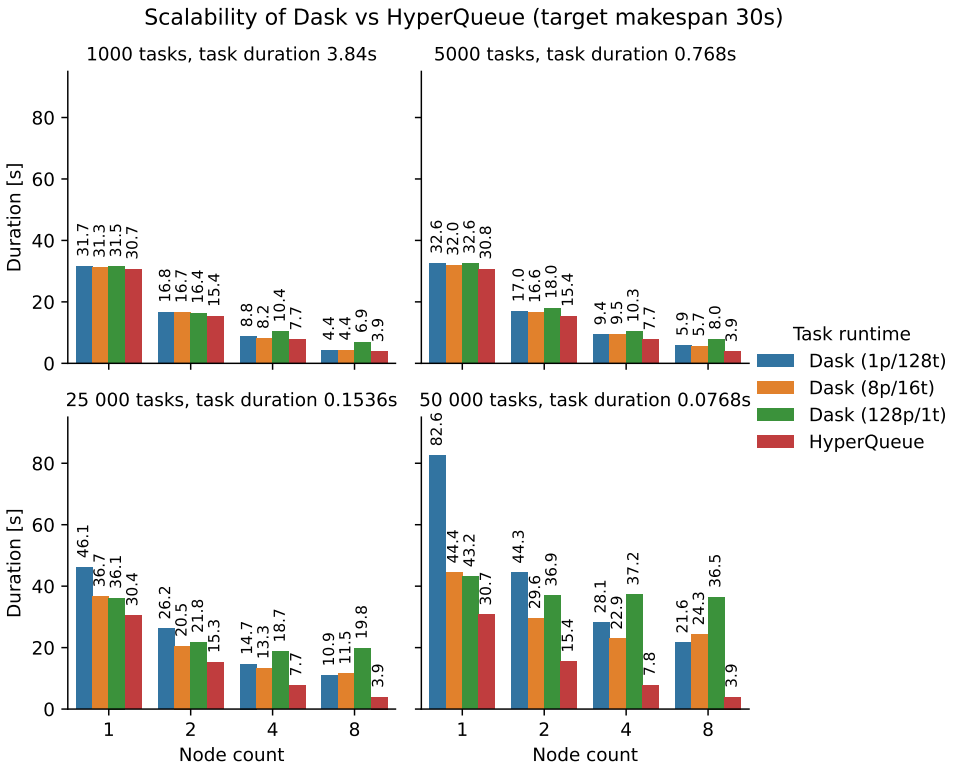


Figure 7.9: Scalability of HyperQueue vs Dask with a fixed target makespan (30 s)

and also on the task duration. With only 1000 tasks and task duration in the range of seconds, Dask is able to scale relatively well in all three configurations, although it is still slightly slower than HyperQueue. However, with a larger number of tasks (tens of thousands), and tasks that last only hundreds or tens of milliseconds, the performance of Dask decreases very quickly.

We can also see differences between the three Dask configurations, particularly in the benchmark with 50 000 tasks. With a single node, Dask is unable to keep up with the number of tasks, and it has a twice longer makespan than when multiple worker processes are used. On the other hand, once the number of nodes is increased, the situation reverses, and the configuration with a single worker per node becomes the fastest one, because Dask starts being overwhelmed by the number of connected workers. This suggests that some amount of manual tuning based on the number of used nodes and workers might be required to achieve optimal performance of Dask workflows. Conversely, HyperQueue was able to keep a consistent performance profile in this experiment without being heavily affected by the number of tasks or the task duration. And because each HyperQueue worker manages all resources of its node, it does not require any manual worker tuning.

Note that this benchmark actually presents a sort of a best-case scenario for Dask. It executes its tasks as Python functions; therefore, it does not need to start a new Linux process per task, unlike HyperQueue. Furthermore, since each task simply sleeps for a specified duration, it releases the GIL and thus does not block other Python threads of the worker, which can thus perform other

work concurrently.

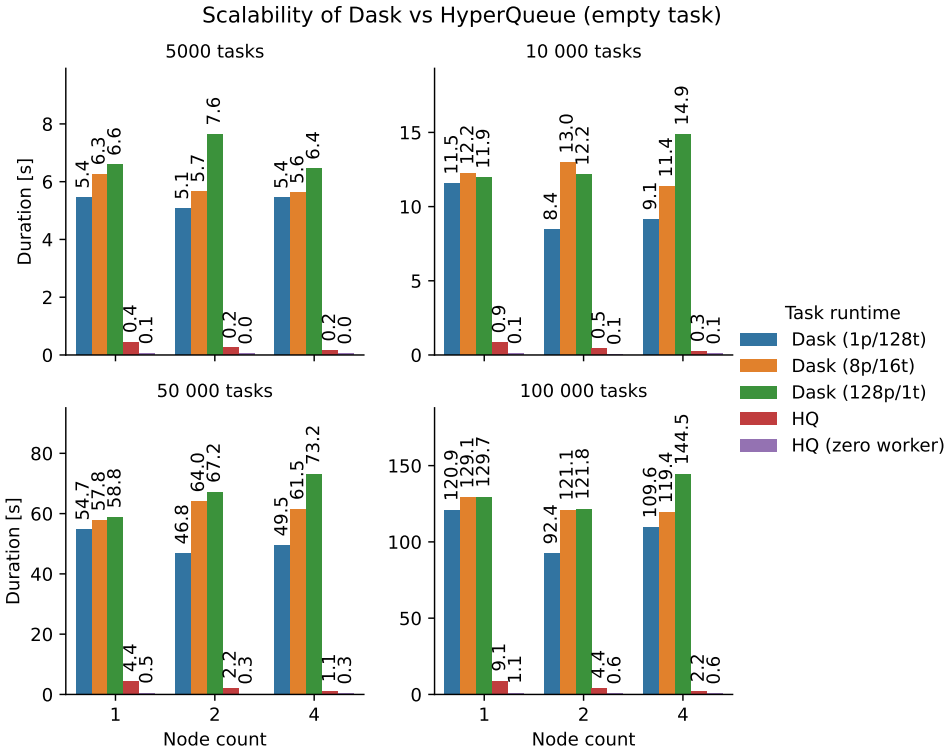


Figure 7.10: Scalability of HyperQueue vs Dask with an empty task

We have further examined the baseline overhead of Dask by executing an empty task (an empty Python function), which emulates the *zero worker* mode of HyperQueue, as it shows how the task runtime scales with the fastest possible task execution. We have compared this empty task execution with running the fastest possible task in HyperQueue (by spawning a process that immediately exits) and also with the *zero worker* mode of HyperQueue.

The results of this experiment can be seen in Figure 7.10. The horizontal axis shows the node count and the vertical axis displays the makespan. Here we can observe a large difference between the inner overhead of HyperQueue and Dask, which struggles to keep up with large amounts of very short tasks. This is of course an artificial scenario because the tasks are empty; however, it does demonstrate that Dask can have performance issues with large or very granular task graphs. It is also clear that the overhead of Dask increases quickly with multiple added workers, which is further exacerbated when multiple workers per node are used.

7.5.5 Group allocation strategies

This experiment examines the effect of resource group allocation strategies on the performance of programs that are sensitive to memory latency. Each Karolina CPU (non-accelerated) computational node has 128 cores and 256 GiB of memory, divided into 8 NUMA nodes. Accessing memory

across these NUMA nodes is slower than accessing memory within the same node, which can affect the performance of programs that are very sensitive to memory throughput or latency. We have created a simple program inspired by the STREAM benchmark [178] that exerts this behavior; it is very sensitive to NUMA memory placement. First, it allocates and initializes 4 GiB of memory stored in a single NUMA node. Then, a number of threads pinned to individual cores repeatedly read from this memory in turn. When these threads are running on cores from a different NUMA node than where the memory was allocated, the program takes longer to execute due to memory transfers being slower.

We have benchmarked the *simple workflow* with 100 tasks on a single Karolina node, where each task executes the described program with either 4 or 8 threads (cores). Each task is configured so the spawned threads are pinned to the specific set of cores assigned to that task by the scheduler. Each available group allocation strategy was examined; *Compact*, which tries to allocate cores using the smallest possible number of groups, *Strict compact*, which always draws cores from the smallest possible number of groups and *Scatter*, which attempts to allocate cores across different groups. As you may recall, HyperQueue workers by default separate available cores into groups according to their NUMA node. We also evaluate another option, *No strategy*, which emulates the behavior of task runtimes that do not provide support for managing groups or NUMA nodes. In this strategy, workers are configured with a flat set of CPU cores without any groups. The order of the available cores is also randomized so that the scheduler cannot use any knowledge of which tasks belong to the same NUMA node.

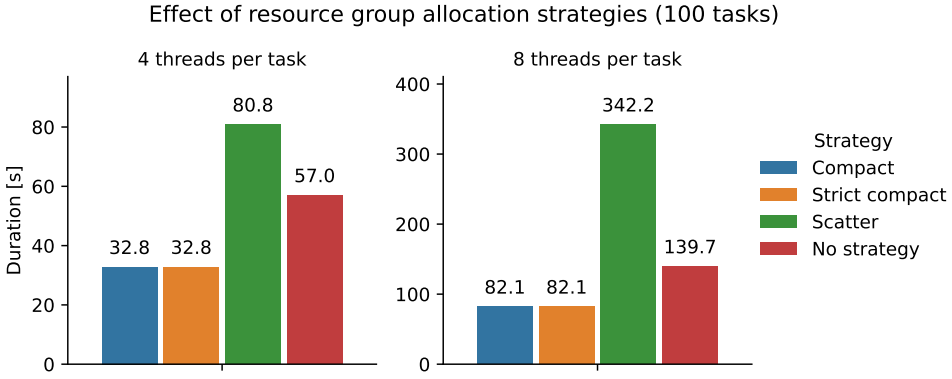


Figure 7.11: Effect of different group allocation strategies

We can observe the results of this experiment in Figure 7.11. The vertical axis displays the makespan of the executed task graph and the four bars represent results of individual strategies. For both evaluated core counts per task, the best performance was achieved using the *Compact* strategy, because the threads were able to achieve the highest possible memory throughput. The *Strict compact* strategy achieved essentially the same result; because the task durations were quite uniform, these two strategies resulted in very similar placements of cores to tasks. On the other hand, the *Scatter* strategy resulted in the worst possible performance, as expected. The results of the *No strategy* configuration are perhaps the most interesting. When cores were assigned to tasks

essentially at random, without taking NUMA effects into account, the performance was better than with the *Scatter* strategy, because in some cases the tasks received multiple cores belonging to the same NUMA node. However, it was still almost twice as slow than with the *Compact* strategy. This suggests that for memory-sensitive programs, using a NUMA-aware task allocation strategy can significantly improve the achieved performance.

7.5.6 Fractional resource requirements

In this experiment we evaluate how can fractional resource requirements improve hardware utilization, and also demonstrate their interplay with non-fungible resources. We perform a simple hyperparameter search that executes 20 independent tasks. Each task executes a Python script that trains a neural network model using the TensorFlow [34] deep-learning framework; the model is designed to perform image classification based on the Fashion MNIST dataset [179]. The experiment was performed on the Karolina GPU partition [180], which contains very powerful NVIDIA A100 GPU accelerators that can achieve over 300 teraFLOPS (trillion floating-point operations per second) and have 40 GiB of memory.

The training script leverages a GPU to accelerate the training; however, it is unable to fully utilize the NVIDIA A100 accelerator. Therefore, it is beneficial to execute multiple training tasks per a single GPU in order to improve its utilization, which can be achieved using fractional resource requirements. We examine four configurations of a fractional GPU resource requirement set by each task, 1, 0.5, 0.25 and 0.1. A fractional requirement of n states that $\frac{1}{n}$ tasks can be executed in parallel on a single GPU; for example, with a requirement of 0.1 GPUs, 10 tasks can run on the same accelerator at once. The requirement of 1 GPU per task emulates the behavior of a task runtime without support for fractional resource requirements, which would be unable to assign multiple tasks to the same accelerator at the same time.

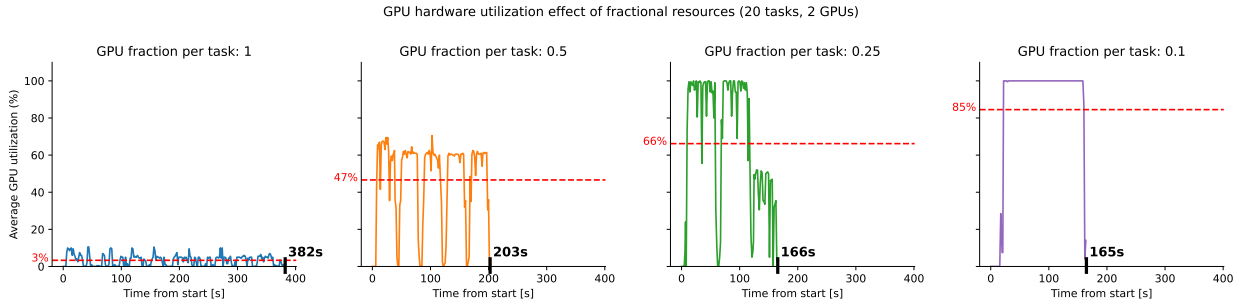


Figure 7.12: GPU hardware utilization improvements with fractional resource requirements

We have executed the described workflow on a Karolina node with two GPUs. The results are displayed in Figure 7.12. The vertical axis shows the average utilization of both GPUs, which was measured using the `nvidia-smi` utility; the horizontal axis shows the progress of the computation in seconds. The four columns denote different configurations for the fraction of an accelerator required by each task. The red horizontal dotted line displays the average GPU utilization over

the course of the whole computation; the black ticks on the horizontal axis denote the makespan of each benchmarked scenario.

We can see that when only a single task is executed on a GPU at the same time, the achieved hardware utilization reaches only 3% and the makespan is over 380 s. When each task required only a half of an accelerator, the makespan was reduced almost by a half, to approximately 200 s and the average utilization has also increased to 47%. With a quarter of a GPU required per task, the makespan was further reduced to 164 s. In the last evaluated scenario, where up to ten tasks were being executed on a single GPU in parallel, we observe diminishing returns. Even though the reported utilization still keeps increasing, the makespan decreases only very slightly. This suggests that the overhead associated with running too many separate processes on the same GPU device start to overcome the benefits achieved by better parallelization.

This experiment demonstrates that using fractional resource requirements can significantly improve hardware utilization and primarily also improve the resulting makespan in situations where individual tasks are unable to make full use of very powerful resources.

This experiment also demonstrates how are non-fungible resources useful in combination with fractional resource requirements. Thanks to the fact that GPUs managed HyperQueue workers are non-fungible and have a separate identity, each task is assigned a specific GPU device that it should leverage by the scheduler, through the `CUDA_VISIBLE_DEVICES` environment variable. This environment variable is then automatically detected by TensorFlow to select the correct GPU that should be used for accelerating the training process. This mechanism works even if multiple tasks share the same device at the same time.

7.5.7 Resource variants

This experiment evaluates the load-balancing benefits of *resource variants*, which were described in Section 7.2.4. For evaluation, we use the simple workflow with 300 tasks executed on a single node with 8 virtual GPUs and the same number of CPUs. Tasks in this experiment simulate a computation that can run either on a GPU or on a CPU. To emulate the performance difference between the GPU and the CPU, tasks executed on the accelerator will sleep for 1 s and tasks executed on the CPU will sleep for a multiple of this duration, according to a given ratio (which is a benchmark parameter). Note that we could also use a larger number of CPU cores and reduce the benchmarked values of this ratio; the result should be approximately the same.

Our goal was to determine how will the resulting makespan differ for the following three situations:

- Tasks are executed only on the GPU.
- A fixed fraction of tasks is executed on the GPU, the rest runs on the CPU.
- The fraction of tasks that is executed on the GPU is determined dynamically by HyperQueue using resource variants.

The results of the experiment can be observed in Figure 7.13. The vertical axis displays the makespan; the five bars represent different configurations of resources. In the *GPU only* mode, all tasks were configured to run on a single GPU. In the *GPU or CPU* mode, tasks were configured to

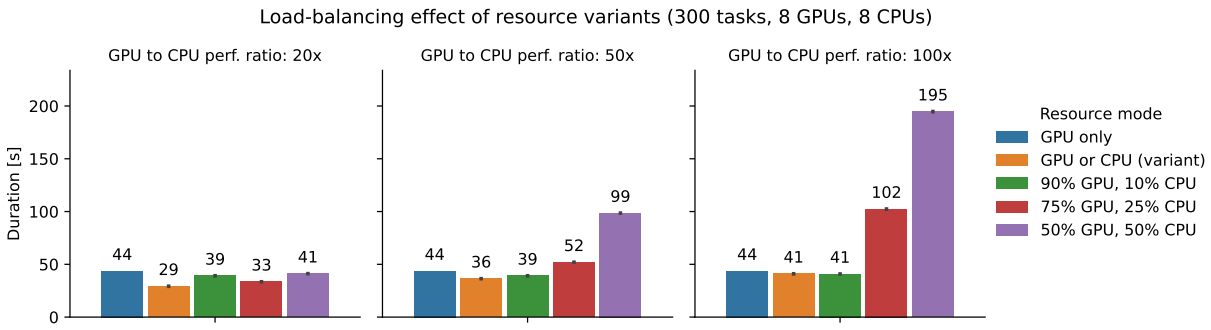


Figure 7.13: Load-balancing effect of resource variants

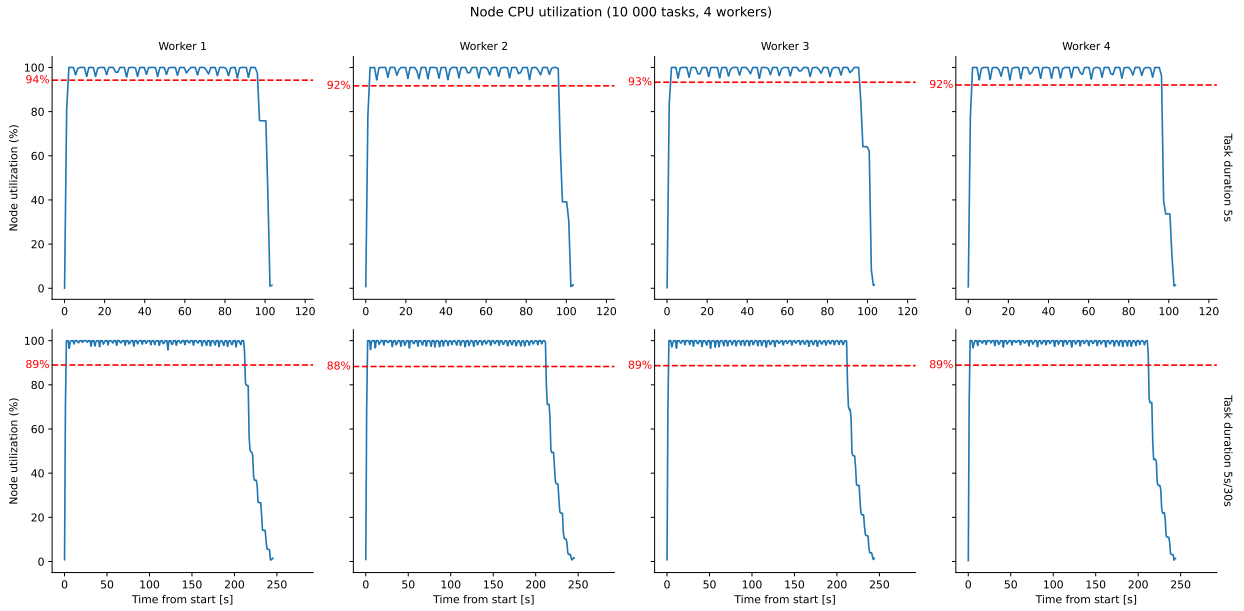
run either on the GPU or on the CPU using resource variants. The remaining three modes assign a specific fraction of tasks to run on the GPU; the rest was executed on the CPU. The three charts show results for different values of the GPU vs CPU performance ratio, i.e. how much faster was the task running on a GPU than on the CPU.

We can see that unless the GPU ratio is too high or we assign too many tasks to the CPU, it is beneficial to also leverage the CPU resources rather than using only GPUs. With CPU fractions determined manually in advance, the result depends heavily on the *GPU ratio*, and in certain configurations it can make the makespan significantly longer. This would thus be both less efficient and more laborious to configure for the user. On the other hand, when the resources for each task are configured using two resource variants (either GPU or CPU), HyperQueue is able to dynamically adapt to all three GPU ratios. In all three scenarios, the makespan with resource variants is strictly smaller than with GPUs only, and it either surpasses or matches the best result achieved with a predetermined fraction of tasks running on the CPU. This shows that resource variants can help achieve better hardware utilization without requiring the user to fine-tune resource configurations of individual tasks in the task graph.

7.5.8 Worker CPU utilization

One of the primary metrics that task graph users care about is hardware utilization – how well can a task graph (executed with a given task runtime) make use of available resources? This experiment examines the average CPU utilization of worker nodes while executing a task graph with HyperQueue, both for tasks with a uniform duration and also for tasks with a skewed distribution of durations. We have benchmarked the *simple workflow* with 10 thousand tasks, where each task fully utilizes a full CPU core by executing the `stress` UNIX utility (instead of just sleeping). This task graph was executed on four workers whose CPU utilization was periodically measured over the course of the computation.

The results are shown in Figure 7.14. The horizontal axis shows the progress of the computation in seconds, the vertical axis displays the average CPU utilization over all 128 cores for each separate worker. The red horizontal dotted line marks the total average utilization of each worker over the course of the whole computation. The top row displays a situation where all tasks in the task



In the bottom row, 75% of tasks last 5 s, 25% of tasks last 30 s

Figure 7.14: CPU utilization of HyperQueue worker nodes

graph have a uniform duration of 5 seconds. In this case, HyperQueue workers kept their average utilization at or above 92%. Note that it takes some amount of time for each `stress` invocation to fully reach full core utilization, and since all tasks have the same duration, the tasks perform this ramp-up process in a mostly synchronized fashion. This causes the "teeth" in the utilization chart. Therefore, it is not realistic to reach full 100% node utilization with this task graph.

The second row displays a situation where the workload is skewed. Most (75%) of the tasks still have a duration of 5 s; the remainder simulates a "long tail" of slower tasks that take 30 s to execute. This is a more difficult situation for achieving high utilization; because some tasks are longer, they might get stuck executing at a time when other resources can no longer be utilized. We can observe this towards the end of the workflow execution, where approximately 30 seconds before the end, the utilization starts to drop. However, even for this skewed task graph, the average CPU utilization of all four workers was kept at almost 90%.

7.5.9 LiGen virtual screening workflow

This experiment evaluates the achieved hardware utilization and scalability of the LiGen virtual screening workflow implemented using HyperQueue. As explained previously in Section 7.4.1, the workflow uses a SMILES file as its main input, which contains a description of a single ligand on each line. To allow HyperQueue to balance the load of the computation across available resources and nodes, the workflow first splits the single input file into several smaller subfiles, each containing a subset of the input lines. Each such file is then processed in two tasks; the first expands the ligands into a three-dimensional structure stored in a MOL2 file and the second performs scoring

on this expanded representation, which generates a CSV file. All the intermediate CSV files are then combined with a single final task at the very end of the workflow.

The number of ligands per file and the number of threads that are used for performing scoring on each file are configurable parameters. There are various trade-offs associated with setting these parameters. The expansion part is sequential; therefore, using as many files (tasks) as possible for this step is beneficial up to the number of available cores. On the other hand, each scoring invocation of LiGen for a file has a non-trivial cost, as it needs to perform a preprocessing step and a part of the computation is also performed sequentially. Therefore, for the scoring part, using fewer files is generally a better choice. Yet, with fewer files, there will also be less opportunities for load balancing performed by HyperQueue and there might not be enough tasks to saturate the available computational resources.

We examine the effect of these two parameters on makespan and the achieved hardware utilization on an input containing 200 thousand ligands. The computation duration varies significantly based on the length of each ligand; therefore, we have tested two input files. The first one ($uniform$) contains 200 thousand copies of the same ligand, which simulates a balanced workload. The second one ($skewed$) contains a variation of ligands with different lengths.

The results can be observed in Figure 7.15. The vertical axis displays the achieved hardware utilization on a single worker; the horizontal axis shows the progress of the computation in seconds. The rows denote the two individual input files, while the columns show results for different combinations of the two input parameters. The red horizontal dotted line displays the total average utilization of a given worker over the course of the whole computation; the black vertical dotted line separates the expansion and scoring sections of the computation. The average node utilization of each section is displayed below the section name. The black ticks on the horizontal axis denote the makespan of each benchmarked scenario.

We can see that the parameter affecting the number of ligands per file has a significant effect on the resulting makespan; using too few or too many ligands results in a longer execution. As expected, the expansion section needs enough tasks to saturate the available cores, because it is single-threaded. For 5000 ligands per file (the rightmost column), the total number of expansion tasks is only 40; therefore, the achieved utilization (for both input files) is only approximately 32%, which corresponds to using just 40 out of the 128 available cores. With more than 128 files available, the utilization approaches 100% for the uniform input. For the skewed input file, the utilization for the expansion section is in general worse than with the uniform input, as expected. With 100 ligands per file, it is reduced significantly; unlike with the uniform input, some of the files were expanded much quicker than others, which led to more load imbalance.

The scoring part is internally parallel and uses all 8 assigned cores; therefore, it does not have to be divided into so many tasks. We can see that even with just 40 files it achieves approximately 80% utilization for both input workloads. In fact, with too many files, we can see the overhead associated with the sequential part of scoring (which needs to be performed for each file separately) starts to dominate; even with the uniform workload, the CPU utilization of the scoring section with 2000 files reaches only approximately 63% with the uniform input and just 44% with the skewed input.

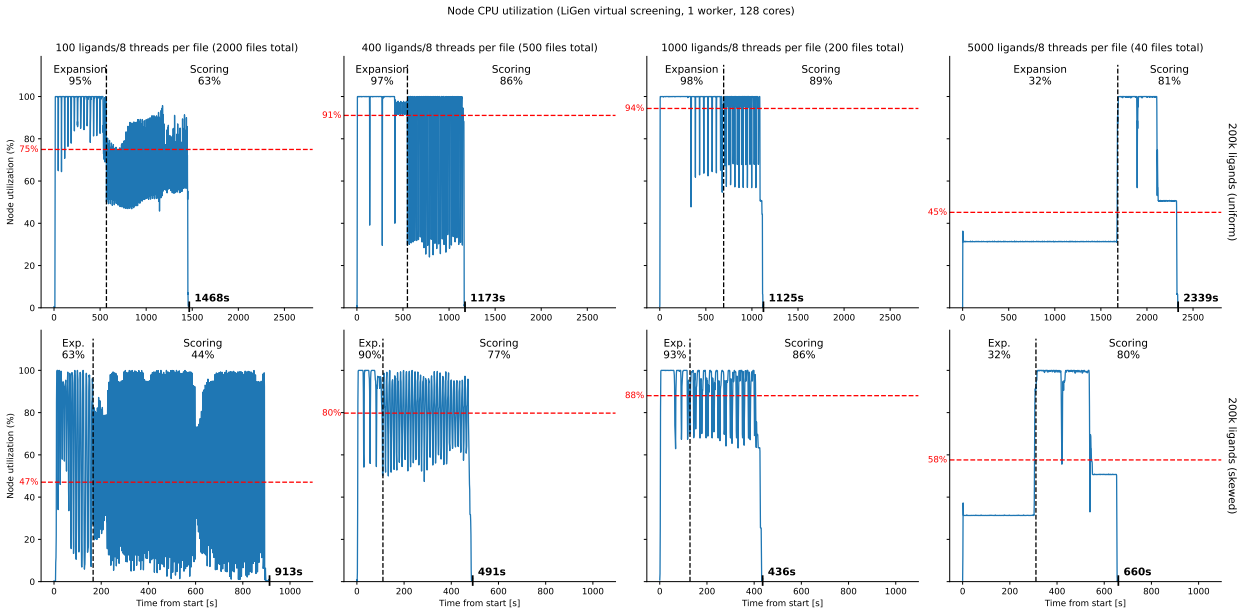


Figure 7.15: Worker hardware utilization with the LiGen virtual screening workflow

The two configurations in the middle present a sort of a sweet spot; especially with 1000 ligands per file, the total average utilization for both workloads is at or above 88%. However, it should be noted that the best configuration will change based on the number of used workers. This can be observed in Figure 7.16, which shows how does the workflow scale up to 4 workers (512 cores). The horizontal axis shows the number of workers and the vertical axis shows the duration of the executed task graph. We can see that with four workers, the configuration with 400 ligands per task starts to beat the configuration with 1000 ligands per task, simply because the latter configuration is not able to provide enough parallelism (tasks) to make efficient use of all the available hardware resources. In general, HyperQueue is able to reduce the makespan with additional workers being added for all four tested configurations.

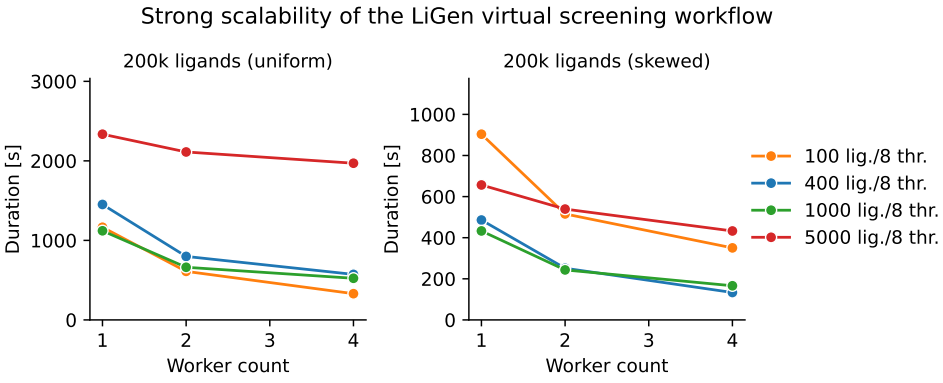


Figure 7.16: Scalability of the LiGen virtual screening workflow

The results of this experiment have showed that HyperQueue is able to reach high node utilization even for imbalanced workloads. However, we can also see that even when the overhead of the task runtime does not get in the way, for some workflows it might be necessary to tune the number of tasks in order to achieve optimal performance.

7.5.10 Automatic allocation

This experiment evaluates the ability of the automatic allocator to scale computational resources both up and down in reaction to computational load. The LiGen virtual screening workflow was executed using 16 ligands per file and 8 threads per file on an input file containing 24 thousand ligands with a skewed distribution of ligands. No workers were explicitly allocated for this experiment; instead, the automatic allocator was configured to submit allocations that would start a single worker per allocated node. The maximum number of workers was set to four and the wall-time (the maximum duration) of the allocations was set to one minute. This very short wall-time was chosen simply to make the benchmark run shorter and to simulate the situation where an allocation ends in the midst of a task graph being computed. The experiment was performed with two configurations for the *idle timeout* duration, which decides after what time a worker (and its allocation) shuts down when it is fully idle. Each configuration was benchmarked a single time.

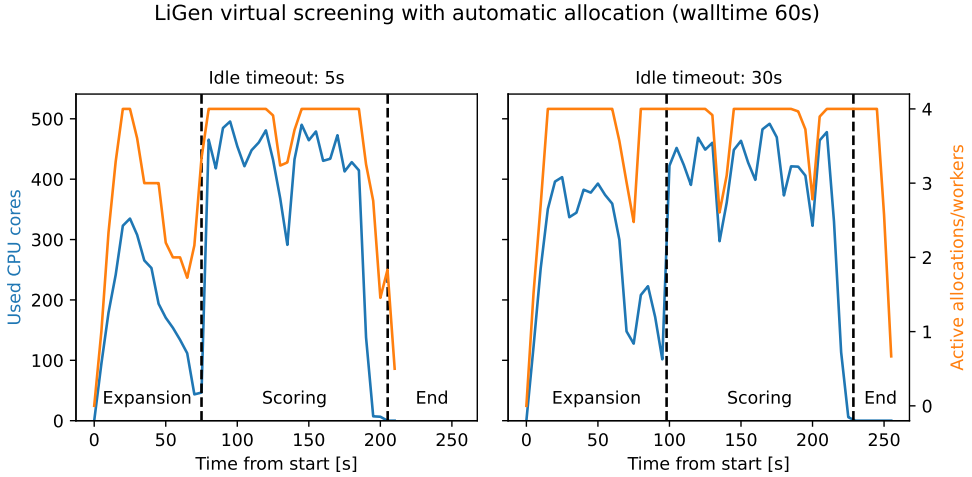


Figure 7.17: Scaling of allocations using the automatic allocator

We can observe the result of the experiment in Figure 7.17. The horizontal axis shows the progress of the computation in seconds. The left vertical axis and the blue line shows the number of cores used for executing tasks (number of running tasks × cores required per task). The right vertical axis and the orange line shows the number of active allocations, which also corresponds to the number of workers, because each allocation spawned exactly one worker. The dashed vertical lines show individual sections of the computation; expansion of ligands, scoring of expanded ligands and an *End* section, where the task graph was already computed and the idle allocations were waiting for the idle timeout to be applied. Note that when we benchmark automatic allocation,

the allocation manager and the global state of the cluster have a significant influence on the time allocations spend waiting to be executed in the queue, and thus also on the total makespan of the executed task graph; therefore, the makespans of these two executions cannot be directly compared.

We can see that allocations were dynamically submitted in response to tasks that needed workers to execute and that new allocations were quickly created in response to previous allocations ending. As you may recall, the wall-time was set to only one minute, after which each allocation ended, which causes the periodic drops in active allocations in both charts. In the left chart, we can see the results for idle timeout being configured to 5 s. We can observe that in this case the allocations were scaled down during the expansion part of the workflow, as not that many resources were required towards the end of expansion. Furthermore, towards the end of the computation, allocations started being scaled down even before the task graph has finished computing. In the right chart, where the idle timeout was much longer (30 s), the allocations were active for the whole duration of the task graph execution, except for short periods of time where new allocations had to be spawned due to wall-time being exhausted. All four allocations were also active up until the very end of the task graph computation, which resulted in more allocation time being wasted.

With idle timeout 5 s, the workflow has consumed 682 node-seconds of allocation time, while with idle timeout 30 s, it has consumed 916 node-seconds. This shows that it is crucial to configure the idle timeout properly to avoid wasting allocation time. By default, HyperQueue uses a wall-time of 1 hour and an idle timeout of 5 minutes, which corresponds to the same wall-time/idle-timeout ratio as the benchmarked configuration with an idle timeout of 5 s.

7.5.11 Server CPU consumption

When deployed on supercomputers, the HyperQueue server is primarily designed to be executed on their login nodes. This could pose a problem if it consumed too many system resources, because login nodes are shared by multiple users and they are not designed for computationally intensive tasks. Some clusters even forcefully limit the total amount of CPU time that can be consumed by processes running on login nodes [181] and terminate the process if it exceeds the maximum allowed time.

To evaluate how many resources the server consumes, we have performed an experiment where the server had to schedule a large number of tasks in a short time period (which acts as a sort of a stress test); we have measured its total CPU time consumption across all cores over this period. The server was running on a login node and it was managing up to 12 workers (1536 cores) and up to 200 thousand tasks. Each worker was deployed on a computational node, so that the server had to communicate with the workers over the network. The benchmark was executed with a fixed makespan; the duration of each task was scaled so that the whole task graph would finish computing in one minute. We measured the total amount of CPU time consumed by the server (both in user-space and in the kernel), using standard Linux process monitoring tools.

Figure 7.18 shows how the CPU utilization of the server changes with a fixed number of workers (12) and an increasing number of tasks. In this case, the server had to schedule the *simple workflow* with the number of tasks ranging from 10 to 200 thousand. The horizontal axis shows the number of tasks that were used for the given benchmark run and the vertical axis shows the CPU time

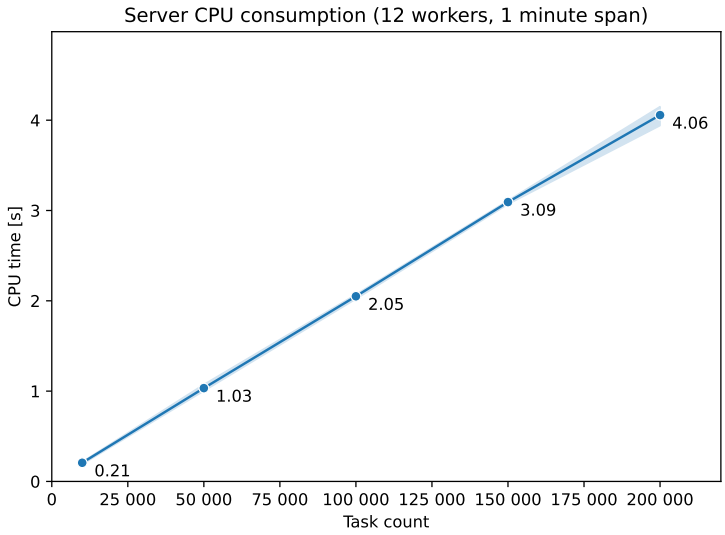


Figure 7.18: HyperQueue server CPU time consumption with an increasing number of tasks

consumed by the server over the measured one-minute period. We can see that the amount of CPU resources scales approximately linearly with the number of tasks scheduled by the server. In the worst case, the server has consumed less than five seconds of CPU time over one minute of real time, even though it had to schedule 200 thousand tasks within this short period, which is an extreme stress test that far exceeds the rate of scheduled tasks in most scientific workflows. Over all the benchmarked task counts, the average CPU consumption of the server per second and per 1000 tasks was approximately 0.0003 s. In other words, for every thousand tasks in the task graph, the server consumed approximately 0.3 ms CPU time every second.

Figure 7.19 displays a situation where the number of tasks is fixed (50 000), but the number of workers increases from 2 to 12. The horizontal axis shows the number of workers that were used for the given benchmark run and the vertical axis shows the CPU time consumed by the server over the one-minute period. In this case, the consumption of the server does not increase by a large amount with more added workers.

In terms of memory consumption, in the largest evaluated case with a task graph containing 200 thousand tasks, the memory consumption of the server, measured using the Linux RSS (Resident Set Size) metric, was approximately 120 MiB, which shows that the server also does not consume large amounts of memory.

The amount of used resources will of course vary based on the executed workflow. However, this experiment shows that the server consumes very few resources in general and that it still has a lot of leeway available even if some task graphs proved to be more computationally demanding than the benchmarked stress test.

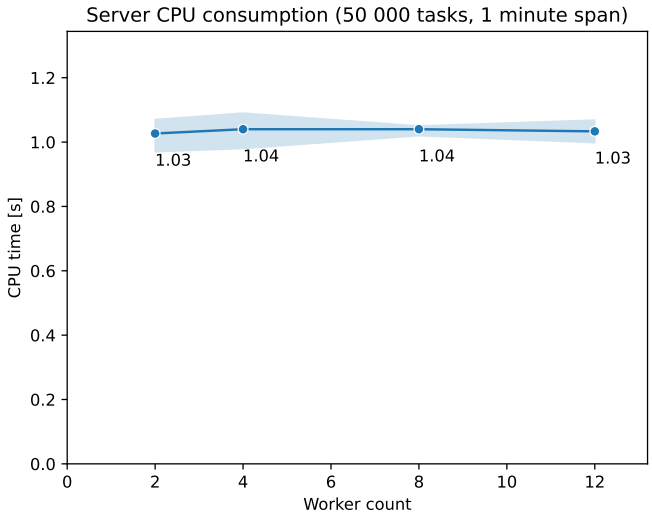


Figure 7.19: HyperQueue server CPU time consumption with an increasing number of workers

7.5.12 Encryption overhead

As has already been noted in Section 7.3.1, HyperQueue encrypts all network communication between the server, the workers, and the client. This encryption is performed because the server is typically deployed on a login node, and thus the communication it performs can be in theory observed by other users of the cluster. In this experiment, we have evaluated the overhead of the encryption, by benchmarking HyperQueue both with encryption enabled and disabled.

We have benchmarked three instances of the *simple workflow* containing 10, 50 and 100 thousand tasks executed with four HyperQueue workers in two configurations; with and without the *zero worker* mode. Because the *zero worker* mode does not actually execute any tasks, it emphasizes the inner overhead of HyperQueue and should thus also amplify potential differences in communication overhead caused by the encryption. Without the *zero worker* mode, each task simply executed the shortest possible sleep command, i.e. `sleep 0`.

We can observe the results of the experiment in Figure 7.20. The horizontal axis shows the number of tasks in the benchmarked task graph and the vertical axis shows the *makespan*. In the *zero worker* mode (displayed on the left), it is clear that encrypting the communication in fact produces a measurable overhead. In the largest measured case with 100 thousand tasks, the makespan is approximately twice longer with encryption enabled. However, as soon as the tasks do any actual work (albeit it being the simplest work possible), the encryption overhead is no longer noticeable and it does not affect the total makespan of the task graph.

In more realistic workflows, the executed tasks will be almost certainly longer than just executing the trivial `sleep` command. In that case, the overhead of encryption will be even less noticeable; the larger the duration of the executed tasks, the less important is the overhead introduced by HyperQueue. If some users would still not want to use encryption for some reason, HyperQueue could be easily modified to add the option to disable encryption at runtime.

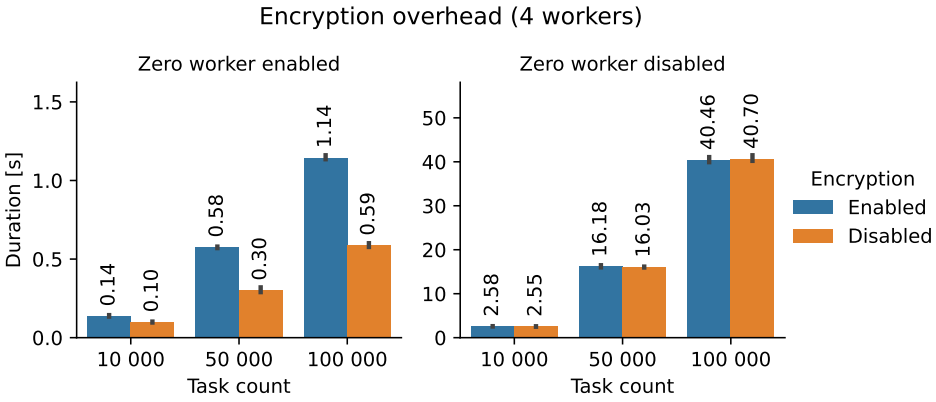


Figure 7.20: Overhead of encryption in HyperQueue communication

7.5.13 Output streaming

In this experiment we evaluate the effect of the *output streaming* mode, which can be used to stream the outputs of *stdout* and *stderr* streams from individual tasks to the HyperQueue server, to avoid creating a large number of files on the filesystem. We have benchmarked two task graphs without dependencies, with 10 and 50 thousand tasks, where each task generates a fixed amount of data (10 or 100 KiB per task). Each task simply outputs the corresponding amount of data to its standard output (*stdout*). The standard error output is disabled and no output is printed to it. In the largest configuration, the task graph produces approximately 5 GiB of data in total.

With output streaming enabled, each task output is streamed to the server, which stores all data sequentially into a single log file. Without output streaming, each task simply creates a single file on disk and writes its *stdout* output to it. The experiment was performed on two different filesystem partitions of the Karolina cluster. The first partition, called SCRATCH, uses the Lustre networked filesystem, which is designed for parallel high intensity I/O workloads [182]. It is a representative of a high-performance filesystem; it can in theory reach up to 700 GiB/s write throughput. The second partition, called PROJECT, uses GPFS (General Parallel File System), which is a much slower filesystem implementation that focuses on providing high availability and redundancy [183].

We can see the results of the experiment in Figure 7.21. The horizontal axis displays the amount of output produced by each task and the vertical axis shows the makespan of the executed task graph. The top row contains data for the SCRATCH filesystem; the bottom row contains data for the PROJECT filesystem. Three separate configurations are displayed in the chart. In the *Stdout* mode, the task graph was executed without output streaming, while in the *Streaming* mode, the task graph was executed with output streaming. The third mode will be described later below.

The results differ significantly based on the amount of output produced by each task and also based on the used filesystem. On the SCRATCH Filesystem, output streaming was able to reduce the makespan approximately by a half in the case where each task outputted 10 KiB of data. However, when the task output was larger (100 KiB), the situation reversed, and output streaming became slower. This is most likely caused by the fact that the sequential writing into the log file

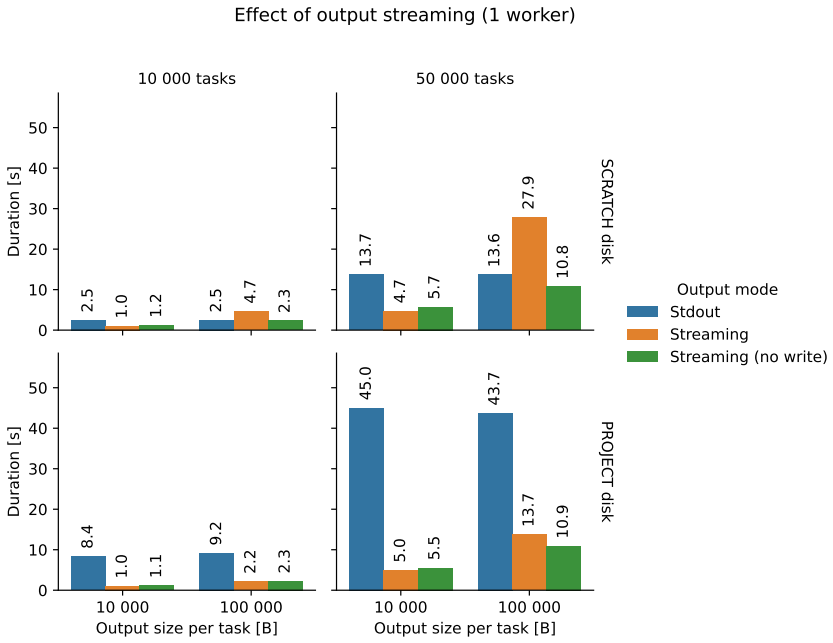


Figure 7.21: Effect of output streaming on the makespan of tasks

by the server becomes a bottleneck, as it cannot be easily parallelized by the filesystem (unlike e.g. writing to many files in parallel). We have confirmed this by benchmarking a third mode, which we call *Streaming (no write)*. In this configuration, output streaming was enabled; therefore, the output of tasks was streamed over the network from workers to the server, but then the server simply discarded the data and did not write it to the sequential log file. We can see that without actually sequentially writing the data to the log file, the makespan is still shorter than without output streaming. It is thus clear that the primary bottleneck in this case is in fact the final write to the log file.

It should be noted that even though output streaming is slower in this case, it still provides the primary benefit of generating only a single file on the filesystem. It can thus help alleviate disk quota limits without any user effort.

With the PROJECT filesystem, output streaming is 4–8 times faster than writing to files directly. The performance benefit of output streaming thus depends heavily on the used filesystem; it seems to be particularly helpful on slower filesystems. One interesting result is that output streaming is generally faster on the PROJECT filesystem than on the SCRATCH filesystem (even though the PROJECT filesystem itself should be much slower). Our conjecture is that this might be caused by the network topology of the cluster. The SCRATCH filesystem uses the InfiniBand [112] network connection, which is also used for network communication between the server and the workers; therefore, performing file I/O on the SCRATCH filesystem while also sending many network packets might cause network contention.

Using output streaming has one additional advantage; it can reduce the total amount of consumed disk space, because it creates only a single file on disk per job. On most filesystems, every

Output per task [B]	Task count	Size (stream) [MiB]	Size (stdout) [MiB]	Ratio
10 000	10 000	96	118	1.23x
10 000	50 000	479	589	1.23x
100 000	10 000	955	978	1.02x
100 000	50 000	4775	4916	1.03x

Table 7.1: Size of task output on disk with and without I/O streaming

file occupies at least a single disk block (which typically consumes several KiB of space); therefore, creating a large number of files also increases disk usage in general. Table 7.1 shows the total size of the output of all tasks of the benchmarked task graphs on disk. The *Size (stream)* column shows the final size of the sequential log file (in this case, output streaming was used), the *Size (stdout)* column displays the total size of a directory containing all files with outputs of individual tasks (in this case, output streaming was not used) and the *Ratio* column shows how much larger was the output in the case where streaming was not used. In the case where each task has outputted 10 KiB of data, writing the output to individual files on disk resulted in approximately 23% higher disk usage. In the second configuration, where each task has outputted 100 KiB of data, this difference was reduced to 3%, because the disk space overhead of file metadata became dwarfed by the size of their content.

Output streaming can help users avoid disk quota limits, reduce consumed disk space and in certain situations also help with I/O performance. However, it is also clear that streaming so much data to a single location (the server) can be slower than writing to many files in parallel if the given filesystem is optimized for such a use-case. As a future improvement, streaming could be made more scalable by streaming task outputs to a single file (or a small amount of files) *per worker*, in order to distribute the I/O load; outputs from individual worker log files could then be merged by the server on demand.

7.6 Comparison with other task runtimes

Several existing task runtimes and their ability to cope with challenges that affect task graph execution on HPC clusters were described in Chapter 4. Even within the particular area of task-based programming that was specified in Chapter 2, there are dozens of tools designed for executing some kind of task graph, whose features and target use-cases partially overlap with HyperQueue. We will focus on a smaller subset of task runtimes that enable performing some form of meta-scheduling on top of allocation managers and can thus be more directly compared with HyperQueue. Several representatives of these tools will be mentioned in this section.

It should be noted that when comparing task runtimes, we compare not only the inherent properties of their design and programming model (such as how do they specify dependencies or manage resource requirements), but also their implementation details (such as what runtime dependencies are necessary to deploy the task runtime). As was shown in Section 5.1.4 and Section 6.2, implementation details are crucial; they significantly affect both the efficiency and scalability of the task runtime and they also dictate how easy it is to use and deploy it. Therefore, we consider it

important to compare actual implementations, rather than only theoretical designs without any implementation available.

Task runtime	Task interfaces: CLI	Task interfaces: Workflow file	Task interfaces: Python API	Task interfaces: Dynamic task graphs	Task dependencies: Explicit	Task dependencies: Inferred from files	Fault tolerance: Worker failure	Fault tolerance: Server failure	Meta-scheduling: Graph partitioning	Meta-scheduling: Automatic allocation	Meta-scheduling: Generic workers	Resources: Basic resources	Resources: Custom resources	Resources: Non-fungible resources	Resources: Related resources	Resources: Fractional resources	Resources: Resource variants	Resources: Multi-node tasks	Data transfers: Directly between tasks	Data transfers: Output streaming	Deployment: Runtime dependencies
GNU parallel [79]	👍	-	-	-	-	-	◐	-	-	-	-	-	-	-	-	-	-	-	-	👍	Perl
HyperShell [96]	👍	-	👍	-	-	-	●	👍	◐	👍	-	-	-	-	-	-	-	-	-	👍	Python
Dask [43]	-	-	👍	👍	👍	-	●	-	●†	👍†	👍	👍	👍	-	-	-	-	-	👍	👍	Python
Ray [76]	-	-	👍	👍	👍	-	●	👍*	●	-	👍	👍	👍	-	👍	👍	-	-	👍	👍	Python
Parsl [75]	-	-	👍	👍	👍	-	●	-	●	👍	-	👍	-	-	-	-	-	👍	👍	-	Python
PyCOMPSs [77]	👍	-	👍	👍	👍	-	●	👍*	◐	-	-	👍	-	-	👍	-	-	👍	👍	👍	Python, Java
Pegasus [18]	-	👍	👍	-	👍	👍	●	👍	◐	-	-	👍	-	-	-	-	-	-	-	👍	Python, Java, HTCondor [184]
Balsam [99]	-	-	👍	👍	👍	-	●	👍	●	👍	👍	👍	-	-	-	-	-	👍	👍	-	Python, PostgreSQL
AutoSubmit [81]	-	👍	-	-	👍	-	●	👍	◐	👍	-	👍	-	-	-	-	-	👍	-	-	Python
FireWorks [82]	-	👍	👍	👍	👍	-	◐	👍	●	👍	👍	-	-	-	-	-	-	-	-	-	Python, MongoDB
Merlin [80]	-	👍	-	-	👍	-	◐	👍	●	-	-	👍	-	-	-	-	-	👍	-	-	Python, RabbitMQ/Redis
SnakeMake [44]	-	👍	👍	-	-	👍	●	-	◐	👍†	-	👍	👍	-	-	-	-	👍	-	-	Python
HyperQueue [3]	👍	👍	👍	👍	👍	-	●	👍	●	👍	👍	👍	👍	👍	👍	👍	👍	👍	-	👍	

👍 supported; - not supported; ● automatic; ◐ manual;
†with an external plugin; *with additional runtime dependencies

Table 7.2: Comparison of meta-scheduling task runtimes

Table 7.2 provides a high-level comparison of HyperQueue and twelve other task runtimes. It should be noted that there are also many important properties that are not mentioned in this table; it primarily compares meta-scheduling and resource management aspects described in this chapter and other features related to challenges described in Chapter 4. Furthermore, there is no single universally best task runtime; each one has a different programming model and a different set of features and trade-offs based on which use-cases it was primarily designed for.

The first section of the table, *Task interfaces*, shows which interfaces can be used to submit tasks and if they support modifying task graphs dynamically. CLI specifies that tasks can be submitted from the command-line, *Workflow file* specifies that task graphs can be defined using e.g. YAML or TOML files and *Python API* marks runtimes that provide the option to specify tasks using Python. Note that the CLI column indicates that tasks and task graphs can be actually defined and submitted using the command-line without requiring additional input files, not just that the runtime provides some form of a CLI (which is offered by most runtimes). The last column, *Dynamic*

task graphs, states whether the task runtime supports adding new tasks to already submitted task graphs dynamically, which is useful e.g. for expressing iterative workflows.

Most evaluated task runtimes provide a Python API, as it enables the most general way of defining task graphs. However, it is relatively rare for other tools to offer a comprehensive CLI that could be used to submit simple task graphs without the need to implement a script using (an imperative) programming language. Merlin and AutoSubmit only allow defining task graphs using workflow files, which can also be quite verbose for simple use-cases. HyperQueue offers both a terminal interface for simple use-cases, a Python API for more complex task graphs and it also supports workflow files.

The second section, *Task dependencies*, marks the ability to express dependencies between tasks. *Explicit* dependencies are described by the user in the task graph definition, usually by mentioning a set of task identifiers on which a given task depends. GNU parallel and HyperShell do not allow specifying task dependencies, as they are designed mostly for scenarios where users simply need to execute a large number of tasks in parallel. Most other tools allow expressing dependencies between tasks explicitly. Pegasus and SnakeMake are also able to infer dependencies automatically based on input/output files consumed/produced by the individual tasks. For SnakeMake, it is the only way to express dependencies, which can limit performance for task graphs with many tasks, as every dependency necessitates the creation of a file on a filesystem.

The third section, *Fault tolerance*, describes what happens after a worker or a server failure. Task runtimes that can automatically restart tasks after a worker failure are marked with ●; those that require user action to recompute failed tasks are marked with ◐. All the evaluated task runtimes provide support for at least some kind of resilience against worker failures; it is a truly fundamental feature without which it would be infeasible to execute task graphs. To be resilient against server failures, the task runtime should be able to restore the state of already computed tasks and all already submitted task graphs when the main server or scheduler component crashes (e.g. when it is executed on a login node that abruptly shuts down). This property is useful for avoiding unnecessary re-execution of long-running task graphs. Resilience against server failures is not supported so ubiquitously, although is is still a relatively common feature. However, most task runtimes require an external service or a database instance to implement server persistence, which can complicate their deployment.

The fourth section, *Meta-scheduling*, specifies three properties related to executing tasks on top of allocations. The following list describes the individual columns:

Graph partitioning states whether the task runtime can assign tasks to allocations fully automatically (●) or if users have to partition tasks manually (◐). As already mentioned in Section 4.1, tools such as Pegasus, AutoSubmit or SnakeMake require users to manually specify how should tasks be grouped into allocations, which requires more work from the user and it can also limit the achieved hardware utilization. Other task runtimes, such as Dask or Balsam, are able to fully automatically assign tasks to allocations without any intervention from the user, same as HyperQueue does.

Automatic allocation highlights task runtimes that are able to automatically submit *HPC allocations*. There are various levels of support for this feature. SnakeMake or AutoSubmit simply

submit allocations that are designed for executing a specific task (or a group of tasks), while other task runtimes, such as Dask, Balsam or HyperQueue, submit allocations dynamically based on the current computational load and task state; these allocations then start a generic worker that is not tied to any specific task or a subset of tasks. Unlike HyperQueue, Dask and Balsam support configuring only a single allocation manager queue into which new allocations can be submitted.

Generic workers states whether the task runtime supports provisioning generic workers that can execute any tasks whose resource requirements are compatible with it, without requiring users to preconfigure workers statically for a specific subset of tasks (which makes load balancing less flexible). Merlin requires preconfiguring queues that are used for executing specific tasks (such as a task that requires exactly two nodes or sixteen CPU cores to execute). A similar approach is used by Parsl. On the other hand, Dask or HyperQueue make worker spawning very easy by using a single generic command, since their workers are not tied to any specific subset of tasks.

The following section, *Resources*, deals with resource management. It shows which task runtimes are only able to specify basic (such as CPU, GPU or memory) task resource requirements and which are able to define custom resource kinds. Support for custom resource requirements (and thus heterogeneous clusters) is not very common; apart from HyperQueue, only Dask, Ray and SnakeMake allow users to define their own resource kinds that are then used during task scheduling. The following four columns describe the support for the complex heterogeneous resource management concepts that were described in Section 7.2. As was already noted, support for these features is not prevalent in existing task runtimes. One notable exception is Ray, which supports both related resources and fractional resources.

The *Multi-node* column then shows which programming models enable describing multi-node tasks. Note that the table only considers this feature as being supported in task runtimes that provide the ability to assign the number of nodes *per task*, not just per task graph or per allocation. We can see that SnakeMake is the only task runtime apart from HyperQueue that supports both custom resource kinds and multi-node tasks.

The penultimate section, *Data transfers*, states whether it is possible to exchange data outputs between tasks directly over the network without going through the filesystem (*Directly between tasks*) and if the task runtime offers the ability to stream standard (error) output of tasks over the network to overcome distributed filesystem limitations (*Output streaming*). Direct data transfer of task outputs is supported by several Python-first task runtimes, such as Dask or Ray; however, it is not supported by HyperQueue.

The last section, *Deployment*, specifies which runtime dependencies and external services have to be available in order to execute the task runtime. We can see that almost all evaluated task runtimes require a Python interpreter, as it is a very popular technology for working with tasks. Some task runtimes, such as Merlin or FireWorks, additionally require an external service for managing persistence or task assignments; these can be challenging to deploy on HPC clusters. Note that this column specifies the *required* runtime dependencies, which are necessary for the task runtime to function. It does not mention optional dependencies; for example, HyperQueue's Python API of course also requires a Python interpreter, but HyperQueue itself can operate without it.

Even though several existing task runtimes also leverage meta-scheduling, they do not offer a holistic set of features that would help resolve the most pressing issues of task graph execution on heterogeneous supercomputers all at once. Some of them are not focused primarily on HPC problems, while others tie tasks to allocations too strongly, which reduces load-balancing opportunities and limits local prototyping. Most task runtimes also require runtime dependencies that might complicate their deployment.

HyperQueue treats HPC as a first-class citizen and provides a unified set of features that help task graph authors execute their task graphs in an easy way on HPC clusters. It is also very efficient and can scale to a large number of hardware resources while being trivial to deploy. Furthermore, HyperQueue provides unique resource scheduling features in the form of non-fungible resources, related resources, fractional resource requirements and resource variants, which can help improve the achieved hardware utilization on heterogeneous clusters. On the other hand, in the current version, its programming model does not yet integrate direct data transfers between tasks,which can be limiting for use-cases that require this functionality.

Summary

This chapter has presented a meta-scheduling and resource management design for executing task graphs on heterogeneous HPC clusters that was created in order to overcome the issues mentioned in Chapter 4. It has also introduced HyperQueue, a distributed task runtime that implements this design through an HPC focused programming model, which enables ergonomic execution of workflows on supercomputers. It is able to meta-schedule tasks to provide load balancing among separate allocations while respecting complex resource requirements that can be arbitrarily configured separately for each task. Its automatic allocator can submit allocations fully autonomously on behalf of the user, which further simplifies the execution of HyperQueue task graphs. It also offers both a straightforward CLI and a Python API for more complex use-cases and it is also trivial to deploy on supercomputers.

The overhead and scalability of HyperQueue was evaluated in various situations. The results indicate that it consumes little resources, introduces a reasonable amount of overhead and can scale to tens of nodes and thousands of cores. The experiments have also shown that it is more than competitive against Dask, a very popular task runtime that is being commonly used for running task graphs on distributed clusters. We have also demonstrated that the heterogeneous resource management concepts introduced in Section 7.2 are able to both improve the achieved hardware utilization of heterogeneous resources and also decrease the makespan of executed task graphs.

Several ideas for possible future improvements of HyperQueue will be summarized in the final chapter.

Chapter 8

Conclusion

The main goal of this thesis was to design and implement approaches that would enable efficient and ergonomic execution of task graphs on heterogeneous supercomputers. This goal was divided into three objectives:

1. Identify and analyze existing challenges and bottlenecks in this area.
2. Design a set of general approaches for overcoming these challenges.
3. Implement and evaluate an HPC-optimized task runtime that leverages the proposed approaches.

Below you can find a description of how were these objectives fulfilled.

The main challenges affecting HPC task graph execution were described in Chapter 4, which has identified several areas that can cause problems when executing task graphs on modern heterogeneous HPC clusters, described how are existing tools able to deal with them and outlined motivation for the work presented in the rest of the thesis.

The following two chapters focused on the performance and efficiency aspects of executing task graphs. Chapter 5 has introduced Estee, a simulation environment designed for prototyping task schedulers and benchmarking various aspects of task graph execution. We have used Estee to perform a comprehensive study of several task scheduling algorithms and examine several hypotheses of the effect of various factors on the performance of the scheduler. Our experiments gave us insight into the quality of the evaluated schedulers.

Chapter 6 focused on the performance bottlenecks of a real-world task runtime Dask in a non-simulated environment. Our analysis has shown that Dask is severely limited by its implementation characteristics, moreso than its used scheduling algorithm. It is primarily limited by its choice of Python as an implementation language, which introduces massive overhead particularly for HPC use-cases. We have proposed and implemented RSDS, a backwards-compatible implementation of the Dask server written in Rust, which was designed to minimize runtime overhead. Our experiments demonstrated that such an optimized server implementation can improve the scalability and end-to-end performance of Dask workflows by several times, even though it uses a simpler scheduling algorithm.

Finally, Chapter 7 introduced a general meta-scheduling approach designed to avoid complications caused by interacting with HPC allocation managers and to efficiently manage complex resources of modern heterogeneous clusters. The described method completely separates task submission from computational resource provisioning, which enables fully dynamic and automatic load balancing even across different allocations. It also introduces resource management concepts designed to improve hardware utilization of heterogeneous clusters.

The described meta-scheduling and resource management approach, along with insights gained from the performed scheduler evaluation and task runtime optimization, was leveraged in the implementation of HyperQueue, an HPC-optimized task runtime that facilitates efficient execution of task graphs in the presence of allocation managers. The most important features of HyperQueue, such as comprehensive support for heterogeneous resource management and multi-node tasks, fault-tolerant task execution and automatic submission of allocations, have been described in Chapter 7. Its overhead and scalability were also evaluated on several benchmarks designed to examine its resource management capabilities and push it to its performance limits. The results of these experiments indicate that it does not introduce significant overhead, it can be used to scale task graphs to a large amount of computational resources and it can efficiently utilize heterogeneous resources.

The following list describes how HyperQueue deals with the challenges described in Chapter 4 and also which improvements could be made to it as future work, in order to further improve its ability to provide ergonomic and efficient task graph execution on heterogeneous HPC clusters.

Allocation manager The used meta-scheduling approach removes the need for the workflow author to think about mapping tasks to allocations or dealing with various allocation limits. And thanks to the automatic allocator, users do not even need to submit any allocations by themselves. The automatic allocator could be extended with task duration or allocation start time predictions in the future, to improve its decisions on when to actually submit allocations.

Cluster heterogeneity HyperQueue provides comprehensive support for heterogeneous clusters by enabling tasks to specify arbitrary resource requirements and by matching these requirements with resources provided by workers. In addition to supporting arbitrary resource kinds, CPU core pinning and time requests, it also supports complex resource requirements in the form of non-fungible resources, related resources, fractional resource requirements and resource variants, which can deal with the most complex resource management scenarios. To our knowledge, there is no other state-of-the-art task runtime that implements all these concepts.

Workers also provide automatic detection of available resources, which further improves the ergonomics of deploying computational providers on heterogeneous clusters.

Performance and scalability Experiments presented in Section 7.5 demonstrate that HyperQueue is able to scale to HPC-sized workflows and that it does not introduce significant overhead over executing tasks manually.

For use-cases that are limited by I/O bandwidth due to creating too many output files on distributed filesystems, HyperQueue offers *output streaming*, which is able to avoid filesystem limitations by streaming task outputs to the server and storing it in a single file.

The performance of HyperQueue could be further improved by integrating HPC-specific technologies, such as InfiniBand [112] or MPI, to speed up network communication and filesystem I/O, or by adding support for stateful task environments. These could help avoid the need to create a separate Linux process for each executed task, which can have a non-trivial overhead on certain HPC clusters, as was demonstrated by our experiments.

Fault tolerance HyperQueue is fault-tolerant by default; tasks that do not finish computing successfully due to reasons outside their control are automatically rescheduled to a different worker, without requiring any manual user intervention. Workers are designed to be transient; because they are usually executed in relatively short-running allocations, their failures are handled gracefully. The server itself is also fault-tolerant and can reload its task database after being restarted, which enables continuous execution of task graphs even in the case of e.g. login node failures.

Multi-node tasks HyperQueue provides built-in support for multi-node tasks and can even combine them with standard single-node tasks within the same task graph. It also provides basic integration with popular HPC technologies like MPI to make their usage in multi-node tasks easier. Multi-node tasks could be further extended to be more granular, so that a multi-node task would not necessarily have to use its whole node for execution. For some use-cases it would also be useful to have an option to combine multi-node tasks with data transfers.

Deployment In terms of ease-of-deployment, HyperQueue is essentially optimal; it is distributed as a single binary that runs fully in user-space and that does not have any dependencies. Its users thus do not have to install any dependencies or deploy complex services on the target supercomputer (which can in some cases be very difficult or even impossible) in order to use it. It is also simple to make it completely self-contained by statically linking a `C` standard library into it, which would remove its only runtime dependency on the `glibc C` standard library implementation. Its Python API, which is an optional component, is distributed as a standard Python package that can be easily installed using standard Python package management tools on a variety of Python versions.

Programming model HyperQueue allows defining task graphs through three interfaces; a CLI, a workflow file, or a Python API, so that it can support a wide range of use-cases. Its task graphs can also be modified even after being submitted, which allows expressing dynamic use-cases, such as iterative computation.

HyperQueue does not currently support direct data transfers between tasks, i.e. enabling tasks to pass their outputs as inputs to dependent tasks through network communication. This limits its applicability (or at least its ergonomics) in scenarios that need to frequently exchange many outputs between tasks. Adding support for data transfers would expand the set of workflows that can be naturally expressed with its Python API and in certain cases it could also improve performance by avoiding the need to exchange task outputs through the filesystem.

8.1 Impact

This final section summarizes the impact that RSDS and HyperQueue had on real-world projects.

RSDS

After we had an indication that RSDS could be leveraged to improve the efficiency of existing Dask workflows, we presented our RSDS research to maintainers of Dask. Although replacing their server implementation or switching from Python to a different implementation language was not a feasible approach for them, some of the ideas implemented in RSDS have since been adapted in the Dask project. This helped to alleviate some of the bottlenecks that we have discovered with our experiments and improved the performance of Dask in general[1,2,3].

HyperQueue

HyperQueue has already been adopted in several projects and it is also actively being used by various researchers and teams across several European HPC centers. It has been proposed as one of the designated ways for executing HPC computations in several supercomputing centers, such as LUMI [185], CSC-FI [186, 187], IT4Innovations [188] or CINECA [189]. It is also available in a precompiled form on several clusters managed by these centers.

HyperQueue can also be integrated as a general task execution system into other tools, thanks to its sophisticated resource management and task scheduling capabilities. This has been leveraged by several workflow management systems that have integrated HyperQueue as one of their task execution backends, such as Aiida [190], NextFlow [191], UM-Bridge [192], StreamFlow [193], ERT [194] or HEAppE [105].

HyperQueue is also used in various research projects. Scientists from the Czech Academy of Sciences use it to execute simulations that analyze data from the ATLAS [173] experiment performed at CERN. Thanks to HyperQueue, they were able to improve the achieved hardware utilization on the IT4Innovations Karolina [161] supercomputer by 30%, which saves them tens of hundreds of node hours per year [176]. HyperQueue has also been used to execute workflows in several projects funded by the European Union, such as EVEREST [195], ACROSS [196], EXA4MIND [172] and MaX [197]. It was especially useful for the LIGATE [4] project, where it was used to implement several MD workflows that were executed using hundreds of thousands of CPU and GPU hours on the most powerful European supercomputers.

Given the use-cases mentioned above, I believe that the practical applicability of the proposed task graph execution design has been demonstrated and that this thesis has thus achieved the goals that it originally set out to. I am confident that HyperQueue provides a tangible benefit in terms of ergonomic and efficient execution of task graphs on supercomputers and that it resolves most of the challenges that have been described extensively in this thesis through its HPC-driven design. I hope that HyperQueue will eventually see even more widespread usage in the HPC community.

[1] `https://github.com/dask/distributed/issues/3139`
[2] `https://github.com/dask/distributed/issues/3783`
[3] `https://github.com/dask/distributed/issues/3872`

List of own publication activities

All citation data presented below is actual as of August 2024, unless otherwise specified. Citation data was taken from Scopus[4]. Self-citation is defined as a citation with a non-empty intersection between the authors of the citing and the cited paper. SJR (Scientific Journal Rankings) ranking was taken from Scimago Journal[5], IF (Impact Factor) ranking was taken from Oxford Academic[6]. The h-index of the author of this thesis according to the Scopus database is 5, with 83 total citations (both excluding self-citations). All publications listed below have already been published.

Note that Ada Böhm was named Stanislav Böhm in older publications.

Publications Related to Thesis

- **J. Beránek**, A. Böhm, G. Palermo, J. Martinovič, and B. Jansík. "HyperQueue: Efficient and ergonomic task graphs on HPC clusters". In: *SoftwareX* 27 (2024), p. 101814. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2024.101814

  Total citations: 0, venue: journal (SJR 0.544)

- S. Böhm and **J. Beránek**. "Runtime vs Scheduler: Analyzing Dask's Overheads". In: *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)* (2020-11), pp. 1–8. DOI: 10.1109/WORKS51914.2020.00006

  Total citations: 8 (2 self-citations), venue: conference proceedings

- **J. Beránek**, S. Böhm, and V. Cima. "Analysis of workflow schedulers in simulated distributed environments". In: *The Journal of Supercomputing* 78.13 (2022-09), pp. 15154–15180. ISSN: 1573-0484. DOI: 10.1007/s11227-022-04438-y

  Total citations: 1 (no self-citations), venue: journal (SJR 0.684)

- G. Palermo, G. Accordi, D. Gadioli, E. Vitali, C. Silvano, B. Guindani, D. Ardagna, A. R. Beccari, D. Bonanni, C. Talarico, F. Lughini, J. Martinovič, P. Silva, A. Böhm, **J. Beránek**, J. Křenek, B. Jansík, B. Cosenza, L. Crisci, P. Thoman, P. Salzmann, T. Fahringer, L. T. Alexander, G. Tauriello, T. Schwede, J. Durairaj, A. Emerson, F. Ficarelli, S. Wingbermühle, E. Lindahl, D. Gregori, E. Sana, S. Coletti, and P. Gschwandtner. "Tunable and Portable Extreme-Scale Drug Discovery Platform at Exascale: The LIGATE Approach". In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. CF '23. Bologna,

[4] https://www.scopus.com
[5] https://www.scimagojr.com
[6] https://academic.oup.com/bioinformatics

Italy: Association for Computing Machinery, 2023, pp. 272–278. ISBN: 9798400701405. DOI: 10.1145/3587135.3592172

Total citations: 4 (4 self-citations), venue: conference proceedings

Posters

- S. Böhm, **J. Beránek**, V. Cima, R. Macháček, V. Jha, A. Kočí, B. Jansík, and J. Martinovič. "HyperQueue: Overcoming Limitations of HPC Job Managers". In: SuperComputing'21 (2021). URL: https://sc21.supercomputing.org/proceedings/tech_poster/tech_poster_pages/rpost104.html

  Presented at the SuperComputing'21 conference.

- S. Böhm, **J. Beránek**, V. Cima, R. Macháček, V. Jha, A. Kočí, B. Jansík, and J. Martinovič. "LIGATE: HyperQueue Scheduler". In: HPCSE 2022 (2022)

  Presented at the HPCSE (High Performance Computing in Science and Engineering) 2022 conference.

- V. Cima, **J. Beránek**, and S. Böhm. "ESTEE: A Simulation Toolkit for Distributed Workflow Execution". In: SuperComputing'19 (2019). URL: https://sc19.supercomputing.org/proceedings/tech_poster/tech_poster_pages/rpost131.html

  Presented at the SuperComputing'19 conference.

Publications Not Related to Thesis

- M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, **J. Beránek**, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G. Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. Di Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. 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*. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 282–297. ISBN: 9781450385572. DOI: 10.1145/3466752.3480133

  Total citations: 45 (20 self-citations), venue: conference proceedings

- M. Besta, Z. Vonarburg-Shmaria, Y. Schaffner, L. Schwarz, G. Kwasniewski, L. Gianinazzi, **J. Beránek**, K. Janda, T. Holenstein, S. Leisinger, P. Tatkowski, E. Ozdemir, A. Balla, M. Copik, P. Lindenberger, M. Konieczny, O. Mutlu, and T. Hoefler. "GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra". In: *Proc. VLDB Endow.* 14.11 (2021-07), pp. 1922–1935. ISSN: 2150-8097. DOI: 10.14778/3476249.3476252

  Total citations: 10 (7 self-citations), venue: journal (SJR 2.376)

- S. Di Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, **J. Beránek**, M. Besta, L. Benini, D. Roweth, and T. Hoefler. "Network-Accelerated Non-Contiguous Memory Transfers". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019-11), pp. 1–14. DOI: 10.1145/3295500.3356189

  Total citations: 10 (3 self-citations), venue: conference proceedings

- S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, **J. Beránek**, L. Benini, and T. Hoefler. "A RISC-V in-network accelerator for flexible high-performance low-power packet processing". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 2575-713X. 2021-06, pp. 958–971. DOI: 10.1109/ISCA52012.2021.00079

  Total citations: 21 (5 self-citations), venue: conference proceedings

- T. De Matteis, J. de Fine Licht, **J. Beránek**, and T. 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. DOI: 10.1145/3295500.3356201

  Total citations: 27 (2 self-citations), venue: conference proceedings

- S. Böhm, **J. Beránek**, and M. Šurkovský. "Haydi: Rapid Prototyping and Combinatorial Objects". In: *Foundations of Information and Knowledge Systems*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 133–149. ISBN: 978-3-319-90050-6. DOI: 10.1007/978-3-319-90050-6_8

  Total citations: 0, venue: conference proceedings

- O. Vávra, **J. Beránek**, J. Šťourač, M. Šurkovský, J. Filipovič, J. Damborský, J. Martinovič, and D. Bednář. "pyCaverDock: Python implementation of the popular tool for analysis of ligand transport with advanced caching and batch calculation support". In: *Bioinformatics* 39.8 (2023-07), btad443. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btad443

  Total citations: 0, venue: journal (IF 5.8)

- M. Golasowski, **J. Beránek**, M. Šurkovský, L. Rapant, D. Szturcová, J. Martinovič, and K. Slaninová. "Alternative Paths Reordering Using Probabilistic Time-Dependent Routing". In: *Advances in Networked-based Information Systems*. 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

  Total citations: 4 (4 self-citations), venue: conference proceedings

- J. Martinovič, M. Golasowski, K. Slaninová, **J. Beránek**, M. Šurkovský, L. Rapant, D. Szturcová, and R. Cmar. "A Distributed Environment for Traffic Navigation Systems". In: *Complex, Intelligent, and Software Intensive Systems*. 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

  Total citations: 1 (1 self-citation), venue: conference proceedings

Bibliography

[1] J. Beránek, S. Böhm, and V. Cima. "Analysis of workflow schedulers in simulated distributed environments". In: *The Journal of Supercomputing* 78.13 (2022-09), pp. 15154–15180. ISSN: 1573-0484. DOI: 10.1007/s11227-022-04438-y.

[2] S. Böhm and J. Beránek. "Runtime vs Scheduler: Analyzing Dask's Overheads". In: *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)* (2020-11), pp. 1–8. DOI: 10.1109/WORKS51914.2020.00006.

[3] J. Beránek et al. "HyperQueue: Efficient and ergonomic task graphs on HPC clusters". In: *SoftwareX* 27 (2024), p. 101814. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2024.101814.

[4] G. Palermo et al. "Tunable and Portable Extreme-Scale Drug Discovery Platform at Exascale: The LIGATE Approach". In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. CF '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 272–278. ISBN: 9798400701405. DOI: 10.1145/3587135.3592172.

[5] J. Michalakes et al. "The Weather Reseach and Forecast Model: Software Architecture and Performance". In: 2004-01.

[6] J. P. Slotnick et al. "CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences". In: 2014.

[7] H. Peréz-Sánchez et al. "Applications of High Performance Computing in Bioinformatics, Computational Biology and Computational Chemistry". In: *Bioinformatics and Biomedical Engineering*. Cham: Springer International Publishing, 2015, pp. 527–541. ISBN: 978-3-319-16480-9.

[8] T. Ben-Nun and T. Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis". In: *ACM Comput. Surv.* 52.4 (2019-08). ISSN: 0360-0300. DOI: 10.1145/3320060.

[9] R. Schaller. "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[10] S. A. McKee and R. W. Wisniewski. "Memory Wall". In: *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, pp. 1110–1116. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_234.

[11] P. Bose. "Power Wall". In: *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, pp. 1593–1608. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_499.

[12] A. Sodani et al. "Knights Landing: Second-Generation Intel Xeon Phi Product". In: *IEEE Micro* 36 (2016), pp. 34–46.

[13] M. Feldman. URL: https://www.top500.org/news/new-gpu-accelerated-supercomputers-change-the-balance-of-power-on-the-top500/ (visited on 04/21/2023).

[14] R. Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[15] J. Nickolls et al. "Scalable Parallel Programming with CUDA." In: *ACM Queue* 6.2 (2008), pp. 40–53. DOI: 10.1145/1365490.1365500.

[16] I. Laguna et al. “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.

[17] T. Ben-Nun et al. “Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures”. 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: 10.1145/3295500.3356173.

[18] E. Deelman et al. “The Evolution of the Pegasus Workflow Management Software”. In: *Computing in Science & Engineering* 21.4 (2019), pp. 22–36. DOI: 10.1109/MCSE.2019.2919690.

[19] H. Bhatia et al. “Generalizable Coordination of Large Multiscale Workflows: Challenges and Learnings at Scale”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476210.

[20] S. Lampa, J. Alvarsson, and O. Spjuth. “Towards agile large-scale predictive modelling in drug discovery with flow-based programming design principles”. In: *Journal of Cheminformatics* 8.1 (2016-11), p. 67. ISSN: 1758-2946. DOI: 10.1186/s13321-016-0179-6.

[21] S. Chunduri et al. “Characterization of MPI Usage on a Production Supercomputer”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 386–400. DOI: 10.1109/SC.2018.00033.

[22] D. E. Bernholdt et al. “A survey of MPI usage in the US exascale computing project”. In: *Concurrency and Computation: Practice and Experience* 32.3 (2020). e4851 cpe.4851, e4851. DOI: https://doi.org/10.1002/cpe.4851.

[23] I. Laguna et al. “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.

[24] F. Darema et al. “A single-program-multiple-data computational model for EPEX/FORTRAN”. In: *Parallel Computing* 7.1 (1988), pp. 11–24. ISSN: 0167-8191. DOI: https://doi.org/10.1016/0167-8191(88)90094-4.

[25] M. P. Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. USA, 1994.

[26] G. Almasi. “PGAS (Partitioned Global Address Space) Languages”. In: *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, pp. 1539–1545. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_210.

[27] R. Rabenseifner, G. Hager, and G. Jost. “Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes”. In: *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43.

[28] *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 3B*. Intel Corporation. 2007-08.

[29] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Third. 2011-09. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[30] N. C. Strategy. *Back to the Building Blocks: a Path Towards Secure and Measurable Software*. National Cybersecurity Strategy, 2024. URL: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf (visited on 03/24/2024).

[31] W. Gropp and E. Lusk. “Fault tolerance in message passing interface programs”. In: *The International Journal of High Performance Computing Applications* 18.3 (2004), pp. 363–372.

[32] E. Lindahl, B. Hess, and D. van der Spoel. "GROMACS 3.0: a package for molecular simulation and trajectory analysis". In: *Molecular modeling annual* 7.8 (2001-08), pp. 306–317. ISSN: 0948-5023. DOI: 10.1007/s008940100045.

[33] M. J. Abraham et al. "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers". In: *SoftwareX* 1-2 (2015), pp. 19–25. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2015.06.001.

[34] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[35] A. Sergeev and M. D. Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *ArXiv* abs/1802.05799 (2018). URL: https://api.semanticscholar.org/CorpusID:3398835.

[36] N. Maruyama et al. "Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers". In: *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12. DOI: 10.1145/2063384.2063398.

[37] K. Hammond. "Glasgow Parallel Haskell (GpH)". In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Boston, MA: Springer US, 2011, pp. 768–779. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_46.

[38] T. M. A. Do et al. "Accelerating Scientific Workflows on HPC Platforms with In Situ Processing". In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2022, pp. 1–10. DOI: 10.1109/CCGrid54584.2022.00009.

[39] P. Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Engineering Bulletin* 38 (2015-01).

[40] R. Shree et al. "KAFKA: The modern platform for data management and analysis in big data domain". In: *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*. 2017, pp. 1–5. DOI: 10.1109/TEL-NET.2017.8343593.

[41] T. Akidau et al. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing". In: *Proceedings of the VLDB Endowment* 8 (2015), pp. 1792–1803.

[42] D. G. Murray et al. "Naiad: a timely dataflow system". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 439–455. ISBN: 9781450323888. DOI: 10.1145/2517349.2522738.

[43] M. Rocklin. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: 2015-01, pp. 126–132. DOI: 10.25080/Majora-7b98e3ed-013.

[44] J. Köster and S. Rahmann. "Snakemake—a scalable bioinformatics workflow engine". In: *Bioinformatics* 28.19 (2012-08), pp. 2520–2522. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts480.

[45] S. Lampa, J. Alvarsson, and O. Spjuth. "Towards agile large-scale predictive modelling in drug discovery with flow-based programming design principles". In: *Journal of Cheminformatics* 8 (2016-11). DOI: 10.1186/s13321-016-0179-6.

[46] T. Cheatham et al. "Bulk synchronous parallel computing-a paradigm for transportable software". In: *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*. Vol. 2. 1995, 268–275 vol.2. DOI: 10.1109/HICSS.1995.375451.

[47] A. V. Gerbessiotis and L. G. Valiant. "Direct bulk-synchronous parallel algorithms". In: *Algorithm Theory — SWAT '92*. Ed. by O. Nurmi and E. Ukkonen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 1–18. ISBN: 978-3-540-47275-9.

[48] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (2008-01), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[49] W. Hadley. "The Split-Apply-Combine Strategy for Data Analysis". In: *Journal of Statistical Software* 40 (2011-04). DOI: 10.18637/jss.v040.i01.

[50] M. Zaharia et al. "Apache Spark: a unified engine for big data processing". In: *Commun. ACM* 59.11 (2016-10), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664.

[51] D. Petersohn. "Dataframe Systems: Theory, Architecture, and Implementation". PhD thesis. EECS Department, University of California, Berkeley, 2021-08. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-193.html.

[52] M. Breddels and J. Veljanoski. "Vaex: Big Data exploration in the era of Gaia". In: *Astronomy & Astrophysics* 618 (2018-01). DOI: 10.1051/0004-6361/201732493.

[53] R. D. Team. *RAPIDS: Libraries for End to End GPU Data Science*. 2023. URL: https://rapids.ai.

[54] C. Augonnet et al. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 2011-02), pp. 187–198. DOI: 10.1002/cpe.1631.

[55] R. D. Blumofe et al. "Cilk: An Efficient Multithreaded Runtime System". In: *Journal of Parallel and Distributed Computing* 37.1 (1996), pp. 55–69. ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1996.0107.

[56] H. Kaiser et al. "HPX: A Task Based Programming Model in a Global Address Space". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. Eugene, OR, USA: Association for Computing Machinery, 2014. ISBN: 9781450332477. DOI: 10.1145/2676870.2676883.

[57] G. Bosilca et al. "PaRSEC: Exploiting Heterogeneity to Enhance Scalability". In: *Computing in Science & Engineering* 15.6 (2013), pp. 36–45. DOI: 10.1109/MCSE.2013.98.

[58] M. Bauer et al. "Legion: Expressing locality and independence with logical regions". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.71.

[59] C. Pheatt. "Intel® threading building blocks". In: *J. Comput. Sci. Coll.* 23.4 (2008-04), p. 298. ISSN: 1937-4771.

[60] P. Thoman et al. "A taxonomy of task-based parallel programming technologies for high-performance computing". In: *The Journal of Supercomputing* 74.4 (2018-04), pp. 1422–1434. ISSN: 1573-0484. DOI: 10.1007/s11227-018-2238-4.

[61] M. Miletto and L. Schnorr. "OpenMP and StarPU Abreast: the Impact of Runtime in Task-Based Block QR Factorization Performance". In: 2019-11, pp. 25–36. DOI: 10.5753/wscad.2019.8654.

[62] M. Kotliar, A. V. Kartashov, and A. Barski. "CWL-Airflow: a lightweight pipeline manager supporting Common Workflow Language". In: *GigaScience* 8.7 (2019-07), giz084. ISSN: 2047-217X. DOI: 10.1093/gigascience/giz084.

[63] *Dagster*. URL: https://dagster.io/ (visited on 06/12/2024).

[64] *Prefect*. URL: https://www.prefect.io/.

[65] *Argo Workflows*. URL: https://argoproj.github.io/workflows/ (visited on 06/12/2024).

[66] Y. Robert et al. "Task Graph Scheduling". In: 2011-01, pp. 2013–2025. ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4_42.

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

[68] H. Wang and O. Sinnen. "List-Scheduling vs. Cluster-Scheduling". In: *IEEE Transactions on Parallel and Distributed Systems* (2018).

[69] J. D. Ullman. "NP-complete Scheduling Problems". In: *J. Comput. Syst. Sci.* 10.3 (1975-06), pp. 384–393. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80008-0.

[70] T. L. Adam, K. M. Chandy, and J. R. Dickson. "A Comparison of List Schedules for Parallel Processing Systems". In: *Commun. ACM* 17.12 (1974-12), pp. 685–690. ISSN: 0001-0782. DOI: 10.1145/361604.361619.

[71] Y.-K. Kwok and I. Ahmad. "Benchmarking the task graph scheduling algorithms". In: *ipps*. IEEE. 1998, p. 0531.

[72] J. Ejarque et al. "Enabling Dynamic and Intelligent Workflows for HPC, Data Analytics, and AI Convergence". In: *ArXiv* abs/2204.09287 (2022). URL: https://api.semanticscholar.org/CorpusID:248266377.

[73] I. Paraskevakos et al. "Task-parallel Analysis of Molecular Dynamics Trajectories". In: *Proceedings of the 47th International Conference on Parallel Processing* (2018). URL: https://api.semanticscholar.org/CorpusID:4756113.

[74] E. Ogasawara et al. "Exploring many task computing in scientific workflows". In: *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*. MTAGS '09. Portland, Oregon: Association for Computing Machinery, 2009. ISBN: 9781605587141. DOI: 10.1145/1646468.1646470.

[75] Y. Babuji et al. "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.

[76] P. Moritz et al. "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.

[77] E. Tejedor et al. "PyCOMPSs: Parallel computational workflows in Python". In: *International Journal of High Performance Computing Applications* 31 (2015-08). DOI: 10.1177/1094342015594678.

[78] V. Cima et al. "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.

[79] O. Tange. *GNU Parallel*. GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them. 2023-07. DOI: 10.5281/zenodo.8175685.

[80] J. L. Peterson et al. "Enabling machine learning-ready HPC ensembles with Merlin". In: *Future Generation Computer Systems* 131 (2022), pp. 255–268. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2022.01.024.

[81] D. Manubens-Gil et al. "Seamless management of ensemble climate prediction experiments on HPC platforms". In: *2016 International Conference on High Performance Computing & Simulation (HPCS)*. 2016, pp. 895–900. DOI: 10.1109/HPCSim.2016.7568429.

[82] A. Jain et al. "FireWorks: a dynamic workflow system designed for high-throughput applications". In: *Concurrency and Computation: Practice and Experience* 27.17 (2015). CPE-14-0307.R2, pp. 5037–5059. ISSN: 1532-0634. DOI: 10.1002/cpe.3505.

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

[84] A. B. Yoo, M. A. Jette, and M. Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4. DOI: 10.1007/10968987_3.

[85] G. 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.

[86] *TORQUE resource manager*. URL: https://adaptivecomputing.com/cherry-services/torque-resource-manager/ (visited on 04/21/2024).

[87] *OpenPBS Open Source Project*. URL: https://www.openpbs.org/ (visited on 04/21/2024).

[88] E. Hataishi et al. "GLUME: A Strategy for Reducing Workflow Execution Times on Batch-Scheduled Platforms". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. Klusáček, W. Cirne, and G. P. Rodrigo. Cham: Springer International Publishing, 2021, pp. 210–230. ISBN: 978-3-030-88224-2.

[89] G. P. Rodrigo et al. "Enabling Workflow-Aware Scheduling on HPC Systems". In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '17. Washington, DC, USA, 2017, pp. 3–14. ISBN: 9781450346993. DOI: 10.1145/3078597.3078604.

[90] W. Fox et al. "E-HPC: a library for elastic resource management in HPC environments". In: *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*. WORKS '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351294. DOI: 10.1145/3150994.3150996.

[91] I. Raicu et al. "Falkon: a Fast and Light-weight tasK executiON framework". In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. SC '07. Reno, Nevada: Association for Computing Machinery, 2007. ISBN: 9781595937643. DOI: 10.1145/1362622.1362680.

[92] *Slurm High Throughput Computing*. URL: https://www.schedmd.com/slurm-support/our-services/high-performance-computing (visited on 04/22/2024).

[93] D. H. Ahn et al. "Flux: A Next-Generation Resource Management Framework for Large HPC Centers". In: *2014 43rd International Conference on Parallel Processing Workshops*. 2014, pp. 9–17. DOI: 10.1109/ICPPW.2014.15.

[94] *IT4Innovations Job Submission Policy*. URL: https://docs.it4i.cz/general/resource_allocation_and_job_execution/%5C#job-submission-and-execution (visited on 04/28/2023).

[95] A. E. Feldmann and L. Foschini. "Balanced Partitions of Trees and Applications". In: *Algorithmica* 71.2 (2015-02), pp. 354–376. ISSN: 1432-0541. DOI: 10.1007/s00453-013-9802-3.

[96] G. Lentner and L. Gorenstein. "HyperShell v2: Distributed Task Execution for HPC". In: *Practice and Experience in Advanced Research Computing*. PEARC '22. Association for Computing Machinery, 2022. ISBN: 9781450391610. DOI: 10.1145/3491418.3535138. URL: https://doi.org/10.1145/3491418.3535138.

[97] *NERSC SnakeMake recommendations*. URL: https://docs.nersc.gov/jobs/workflow/snakemake/ (visited on 08/03/2024).

[98] Y. Zhang, C. Koelbel, and K. Cooper. "Batch queue resource scheduling for workflow applications". In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, pp. 1–10. DOI: 10.1109/CLUSTR.2009.5289186.

[99] M. A. Salim et al. *Balsam: Automated Scheduling and Execution of Dynamic, Data-Intensive HPC Workflows*. 2019. arXiv: 1909.08704 [cs.DC]. URL: https://arxiv.org/abs/1909.08704.

[100] *Dask Jobqueue*. URL: https://jobqueue.dask.org/en/latest/index.html (visited on 07/24/2024).

[101] C. Harris et al. "Dynamic Provisioning and Execution of HPC Workflows Using Python". In: *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. 2016, pp. 1–8. DOI: 10.1109/PyHPC.2016.005.

[102] D. Hudak et al. "Open OnDemand: A web-based client portal for HPC centers". In: *Journal of Open Source Software* 3.25 (2018), p. 622. DOI: 10.21105/joss.00622.

[103] V. Svatoň et al. "HPC-as-a-Service via HEAppE Platform". In: *Complex, Intelligent, and Software Intensive Systems*. Ed. by L. Barolli, F. K. Hussain, and M. Ikeda. Cham: Springer International Publishing, 2020, pp. 280–293. ISBN: 978-3-030-22354-0.

[104] A. Scionti et al. “HPC, cloud and big-data convergent architectures: The lexis approach”. In: *Complex, Intelligent, and Software Intensive Systems: Proceedings of the 13th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS-2019)*. Springer. 2020, pp. 200–212.

[105] *HEAppE HyperQueue integration*. URL: `https://heappe.it4i.cz/docs/4-2-0/pages/hyperqueue.html` (visited on 07/29/2024).

[106] A. Khan et al. “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`.

[107] *Dask UCX Integration*. URL: `https://dask-cuda.readthedocs.io/en/stable/ucx.html` (visited on 04/21/2024).

[108] S. Di Girolamo et al. “PsPIN: A high-performance low-power architecture for flexible in-network compute”. In: *arXiv:2010.03536 [cs]* (2021-06).

[109] S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, J. Beránek, L. Benini, and T. Hoefler. “A RISC-V in-network accelerator for flexible high-performance low-power packet processing”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 2575-713X. 2021-06, pp. 958–971. DOI: `10.1109/ISCA52012.2021.00079`.

[110] *Task Graph Building Performance Issue in Dask*. URL: `https://github.com/dask/distributed/issues/3783` (visited on 05/25/2022).

[111] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[112] R. Buyya, T. Cortes, and H. Jin. “An Introduction to the InfiniBand Architecture”. In: *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. 2002, pp. 616–632. DOI: `10.1109/9780470544839.ch42`.

[113] P. Vergés et al. “Task-Level Checkpointing System for Task-Based Parallel Workflows”. In: *Euro-Par 2022: Parallel Processing Workshops*. Cham: Springer Nature Switzerland, 2023, pp. 251–262. ISBN: 978-3-031-31209-0.

[114] X. You et al. “L-DAG: Enabling Loopy Workflow in Scientific Application with Automatic DAG Transformation”. In: *2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. 2019, pp. 946–953. DOI: `10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00174`.

[115] J. Kim et al. “Technology-driven, highly-scalable dragonfly topology”. In: *ACM SIGARCH Computer Architecture News* 36.3 (2008), pp. 77–88.

[116] M. Besta and T. Hoefler. “Slim Fly: A Cost Effective Low-Diameter Network Topology”. In: *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 348–359. DOI: `10.1109/SC.2014.34`.

[117] O. Sinnen and L. Sousa. “Communication contention in task scheduling”. In: *IEEE Transactions on Parallel and Distributed Systems* 16.6 (2005), pp. 503–515. DOI: `10.1109/TPDS.2005.64`.

[118] A. Jarry, H. Casanova, and F. Berman. *DAGSim: A Simulator for DAG Scheduling Algorithms*. Research Report LIP RR-2000-46. Laboratoire de l’informatique du parallélisme, 2000-12, 2+8p.

[119] A. Zulianto, Kuspriyanto, and Y. S. Gondokaryono. “HPC resources scheduling simulation using SimDAG”. In: *2016 6th International Conference on Electronics Information and Emergency Communication (ICEIEC)*. 2016, pp. 334–337. DOI: `10.1109/ICEIEC.2016.7589751`.

[120] H. Casanova et al. “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms”. In: *Journal of Parallel and Distributed Computing* 74.10 (2014-06), pp. 2899–2917.

[121] J. Beránek, A. Böhm, and V. Cima. *ESTEE*. https://github.com/it4innovations/estee. 2021.

[122] E. Babulak and M. Wang. "Discrete Event Simulation: State of the Art". In: *International Journal of Online Engineering (iJOE)* 4 (2008-01), pp. 60–63. DOI: 10.5772/9894.

[123] X. Tang et al. "List scheduling with duplication for heterogeneous computing systems". In: *Journal of Parallel and Distributed Computing* 70 (2010-04), pp. 323–329. DOI: 10.1016/j.jpdc.2010.01.003.

[124] X. Yao, P. Geng, and X. Du. "A Task Scheduling Algorithm for Multi-core Processors". In: *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*. 2013, pp. 259–264. DOI: 10.1109/PDCAT.2013.47.

[125] Y.-K. Kwok and I. Ahmad. "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors". In: *IEEE Trans. Parallel Distrib. Syst.* 7.5 (1996-05), pp. 506–521. ISSN: 1045-9219. DOI: 10.1109/71.503776.

[126] D. Bertsekas and R. Gallager. *Data Networks (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN: 978-0-13-200916-4. DOI: 10.5555/121104.

[127] Y.-K. Kwok and I. Ahmad. "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors". In: *ACM Comput. Surv.* 31.4 (1999-12), pp. 406–471. ISSN: 0360-0300. DOI: 10.1145/344588.344618.

[128] G. Sih and E. Lee. "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 4.2 (1993), pp. 175–187. DOI: 10.1109/71.207593.

[129] M.-Y. Wu and D. Gajski. "Hypertool: a programming aid for message-passing systems". In: *IEEE Transactions on Parallel and Distributed Systems* 1.3 (1990), pp. 330–343. DOI: 10.1109/71.80160.

[130] D. Dolev and M. K. Warmuth. "Scheduling precedence graphs of bounded height". In: *Journal of Algorithms* 5.1 (1984), pp. 48–59. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(84)90039-7.

[131] F. A. Omara and M. M. Arafa. "Genetic algorithms for task scheduling problem". In: *Journal of Parallel and Distributed Computing* 70.1 (2010), pp. 13–22. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2009.09.009.

[132] R. F. d. Silva et al. "Community Resources for Enabling Research in Distributed Scientific Workflows". In: *2014 IEEE 10th International Conference on e-Science*. Vol. 1. 2014, pp. 177–184. DOI: 10.1109/eScience.2014.44.

[133] J. Beránek, S. Böhm, and V. Cima. *Task graphs for benchmarking schedulers*. Version 1.0. Zenodo, 2019-04. DOI: 10.5281/zenodo.2630385.

[134] J. Beránek, S. Böhm, and V. Cima. *Task Scheduler Performance Survey Results*. Version 1.0. Zenodo, 2019-04. DOI: 10.5281/zenodo.2630589.

[135] *IT4Innovations supercomputing center*. URL: https://docs.it4i.cz/ (visited on 01/19/2024).

[136] *Salomon supercomputer*. URL: https://docs.it4i.cz/salomon/introduction (visited on 07/21/2024).

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

[138] W. McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by S. van der Walt and J. Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[139] C. R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020-09), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[140] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[141] S. Gross. *Making the Global Interpreter Lock Optional in CPython*. PEP 703. 2023. URL: https://peps.python.org/pep-0703/.

[142] M. Dugré, V. Hayot-Sasson, and T. Glatard. "A performance comparison of Dask and Apache Spark for data-intensive neuroimaging pipelines". In: *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE. 2019.

[143] K. et al. *The NCEP/NCAR 40-year reanalysis project*. 1996.

[144] *Benchmarking Python Distributed AI Backends with Wordbatch*. URL: https://towardsdatascience.com/benchmarking-python-distributed-ai-backends-with-wordbatch-9872457b785c (visited on 04/20/2020).

[145] J. Beránek and A. Böhm. *RSDS*. https://github.com/it4innovations/rsds. 2020.

[146] T. Li et al. "Analysis of NUMA effects in modern multicore systems for the design of high-performance data transfer applications". In: *Future Generation Computer Systems* 74 (2017), pp. 41–50. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.04.001.

[147] R. Amela et al. "Executing linear algebra kernels in heterogeneous distributed infrastructures with PyCOMPSs". In: *Oil & Gas Science and Technology - Revue de l IFP* Forthcoming (2018-07). DOI: 10.2516/ogst/2018047.

[148] *Slurm resource sharding*. URL: https://slurm.schedmd.com/gres.html#Sharding (visited on 08/17/2024).

[149] S. Klabnik and C. Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284.

[150] J. Beránek et al. *HyperQueue*. https://github.com/it4innovations/hyperqueue. 2024.

[151] J. Beránek and A. Böhm. *HyperQueue documentation*. URL: https://it4innovations.github.io/hyperqueue/stable/ (visited on 06/22/2024).

[152] J. Beránek and A. Böhm. *HyperQueue Python package at Python Package Index (PyPi)*. URL: https://pypi.org/project/hyperqueue/ (visited on 06/22/2024).

[153] M. Hategan-Marandiuc et al. "PSI/J: A Portable Interface for Submitting, Monitoring, and Managing Jobs". In: *2023 IEEE 19th International Conference on e-Science (e-Science)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023-10, pp. 1–10. DOI: 10.1109/e-Science58273.2023.10254912.

[154] F. Lehmann et al. "How Workflow Engines Should Talk to Resource Managers: A Proposal for a Common Workflow Scheduling Interface". In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023-05, pp. 166–179. DOI: 10.1109/CCGrid57682.2023.00025.

[155] C. Vercellino et al. "A Machine Learning Approach for an HPC Use Case: the Jobs Queuing Time Prediction". In: *Future Generation Computer Systems* 143 (2023), pp. 215–230. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.01.020.

[156] M. H. Hilman, M. A. Rodriguez, and R. Buyya. "Task Runtime Prediction in Scientific Workflows Using an Online Incremental Learning Approach". In: *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. 2018, pp. 93–102. DOI: 10.1109/UCC.2018.00018.

[157] A. Beccari et al. "LiGen: A High Performance Workflow for Chemistry Driven de Novo Design". In: *Journal of chemical information and modeling* 53 (2013-04). DOI: 10.1021/ci400078g.

[158] D. Gadioli et al. "EXSCALATE: An Extreme-Scale Virtual Screening Platform for Drug Discovery Targeting Polypharmacology to Fight SARS-CoV-2". In: *IEEE Transactions on Emerging Topics in Computing* PP (2022-01), pp. 1–12. DOI: 10.1109/TETC.2022.3187134.

[159] D. Weininger. "SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules". In: *Journal of Chemical Information and Computer Sciences* 28.1 (1988), pp. 31–36. DOI: 10.1021/ci00057a005.

[160] *Tripos Mol2 File Format*. URL: https://zhanggroup.org/DockRMSD/mol2.pdf (visited on 07/20/2024).

[161] *Karolina supercomputer*. URL: https://docs.it4i.cz/karolina/introduction/ (visited on 06/01/2024).

[162] *LUMI supercomputer*. URL: https://www.lumi-supercomputer.eu/ (visited on 07/19/2024).

[163] M. Turisini, M. Cestari, and G. Amati. "LEONARDO: A Pan-European Pre-Exascale Supercomputer for HPC and AI applications". In: *Journal of large-scale research facilities JLSRF* 9 (2024-01). DOI: 10.17815/jlsrf-8-186.

[164] *E4 Computer Engineering cluster*. URL: https://www.e4company.com/en/high-performance-computing/ (visited on 07/19/2024).

[165] V. Lindahl, J. Lidmar, and B. Hess. "Accelerated weight histogram method for exploring free energy landscapes". In: *The Journal of Chemical Physics* 141.4 (2014-07), p. 044110. ISSN: 0021-9606. DOI: 10.1063/1.4890371.

[166] J. Beránek, S. Wingbermühle, and D. Gadioli. *LIGATE CADD and Virtual Screening Workflow*. https://github.com/LigateProject/Computer-Aided-Drug-Design-workflow-HyperQueue. 2024.

[167] M. Su et al. "Comparative Assessment of Scoring Functions: The CASF-2016 Update". In: *Journal of Chemical Information and Modeling* 59.2 (2019), pp. 895–913. DOI: 10.1021/acs.jcim.8b00545.

[168] *MeluXina supercomputer*. URL: https://www.luxprovide.lu/meluxina/ (visited on 07/19/2024).

[169] S. Wingbermühle. *LIGATE Pose Selector Workflow*. https://github.com/LigateProject/Pose-Selector-workflow. 2024.

[170] S. Wingbermühle et al. *Pose Selector Workflow - Structure Input Files for Machine Learning (Set 2)*. Version 1.0. Zenodo, 2024-06. DOI: 10.5281/zenodo.11397486.

[171] S. Wingbermühle et al. *Pose Selector Workflow - Docking Poses, Absolute Binding Free Energy Estimates and Structure Input Files for Machine Learning*. Version 1.0. Zenodo, 2024-06. DOI: 10.5281/zenodo.11397017.

[172] *Extreme Analytics for Mining Data spaces*. 2023. URL: https://exa4mind.eu/ (visited on 07/12/2024).

[173] G. Aad et al. "The ATLAS Experiment at the CERN Large Hadron Collider". In: *JINST* 3 (2008), S08003. DOI: 10.1088/1748-0221/3/08/S08003.

[174] M. Svatoš, J. Chudoba, and P. Vokáč. "ATLAS utilisation of the Czech national HPC center". In: *EPJ Web of Conferences* 214 (2019-01), p. 03005. DOI: 10.1051/epjconf/201921403005.

[175] M. Svatoš, J. Chudoba, and P. Vokáč. "Improvements in utilisation of the Czech national HPC center". In: *EPJ Web of Conferences* 245 (2020-01), p. 09010. DOI: 10.1051/epjconf/202024509010.

[176] M. Svatoš, J. Chudoba, and P. Vokáč. "ARC-CE+HyperQueue based submission system of ATLAS jobs for the Karolina HPC". In: (2022). URL: https://cds.cern.ch/record/2837871.

[177] *Dask Best Practices - Processes and Threads*. URL: https://docs.dask.org/en/stable/best-practices.html#processes-and-threads (visited on 07/18/2024).

[178] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995-12), pp. 19–25.

[179] H. Xiao, K. Rasul, and R. Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. 2017. arXiv: 1708.07747 [cs.LG]. URL: https://arxiv.org/abs/1708.07747.

[180] *Karolina GPU partition*. URL: https://docs.it4i.cz/karolina/compute-nodes/#compute-nodes-with-a-gpu-accelerator (visited on 08/21/2024).

[181] *CPU time limit on login nodes of the LEONARDO cluster*. URL: https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.2.2%3A+LEONARDO+DCGP+UserGuide (visited on 06/02/2024).

[182] *Karolina SCRATCH filesystem*. URL: https://docs.it4i.cz/karolina/storage/#scratch-file-system (visited on 07/16/2024).

[183] *Karolina PROJECT filesystem*. URL: https://docs.it4i.cz/storage/project-storage/ (visited on 07/16/2024).

[184] D. Thain, T. Tannenbaum, and M. Livny. "Distributed computing in practice: the Condor experience." In: *Concurrency - Practice and Experience* 17.2-4 (2005), pp. 323–356.

[185] *HyperQueue facilitates better utilization of LUMI's computational resources*. URL: https://www.lumi-supercomputer.eu/hyperqueue-facilitates-better-utilization-of-lumis-computational-resources (visited on 01/27/2024).

[186] *High-throughput computing and workflows on the Puhti cluster*. URL: https://docs.csc.fi/computing/running/throughput/ (visited on 07/24/2024).

[187] *HyperQueue guide for the CSC clusters*. URL: https://docs.csc.fi/apps/hyperqueue/ (visited on 08/03/2024).

[188] *HyperQueue documentation for the IT4Innovations cluster*. URL: https://docs.it4i.cz/general/hyperqueue (visited on 01/27/2024).

[189] *HPC Cineca website*. URL: https://www.hpc.cineca.it (visited on 06/01/2024).

[190] *Aiida HyperQueue integration*. URL: https://github.com/aiidateam/aiida-hyperqueue (visited on 01/27/2024).

[191] *Nextflow HyperQueue integration*. URL: https://docs.csc.fi/support/tutorials/nextflow-hq (visited on 01/27/2024).

[192] L. Seelinger et al. *Democratizing Uncertainty Quantification*. 2024. arXiv: 2402.13768.

[193] *StreamFlow HyperQueue integration*. URL: http://gitlab.linksfoundation.com/across-public/orchestrator/components/hyperqueue-streamflow-plugin (visited on 06/11/2024).

[194] *Ensemble based Reservoir Tool*. URL: https://github.com/equinor/ert (visited on 01/27/2024).

[195] C. Pilato et al. "EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 1320–1325. DOI: 10.23919/DATE51398.2021.9473940.

[196] M. Aldinucci et al. "The Italian research on HPC key technologies across EuroHPC". In: CF '21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 178–184. ISBN: 9781450384049. DOI: 10.1145/3457388.3458508.

[197] *MaX (MAterials design at the eXascale) Centre of Excellence Project*. 2023. URL: https://www.max-centre.eu/ (visited on 06/01/2024).

Appendix A

Benchmark configurations

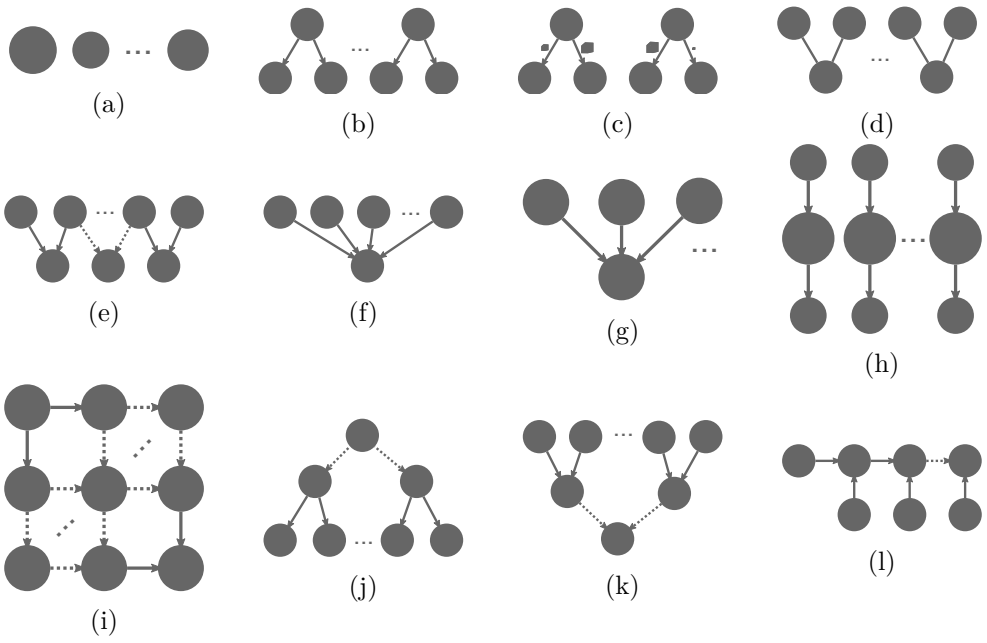


Figure A.1: Task graph shapes in the *elementary* ESTEE benchmark data set

Graph	D	#T	#O	TS	LP	Description
plain1n	e	380	0	0.00	1	Independent tasks; normally distributed durations (Fig. A.1a)
plain1e	e	380	0	0.00	1	Independent tasks; exponentially distributed durations (Fig. A.1a)
plain1cpus	e	380	0	0.00	1	Independent tasks with varying core requirements (Fig. A.1a)
triplets	e	330	220	17.19	3	Task triplets; middle task requires 4 cores (Fig. A.1h)
merge_neighb.	e	214	107	10.36	2	Merge of adjacent task pairs (Fig. A.1e)
merge_triplets	e	148	111	10.77	2	Merge of task triplets (Fig. A.1g)
merge_sm-big	e	240	160	7.74	2	Merge of two results (0.5 MiB and 100 MiB data objects) (Fig. A.1d)
fork1	e	300	100	9.77	2	Tasks with a pair of consumers each consuming the same output (Fig. A.1b)
fork2	e	300	200	19.53	2	Tasks with a pair of consumers each consuming different output (Fig. A.1c)
bigmerge	e	321	320	31.25	2	Merge of a large number of tasks (variant of Fig. A.1f)
duration_stairs	e	380	0	0.00	1	Independent tasks; task durations range from 1 to 190 s (Fig. A.1a)
size_stairs	e	191	190	17.53	2	1 producer 190 outputs / 190 consumers; sizes range from 1 to 190 MiB
splitters	e	255	255	32.25	8	Binary tree of splitting tasks (Fig. A.1j)
conflux	e	255	255	31.88	8	Merging task pairs (inverse of *splitters*) (Fig. A.1k)
grid	e	361	361	45.12	37	Tasks organized in a 2D grid (i.e. *splitters* followed by *conflux*) (Fig. A.1i)
fern	e	401	401	11.11	201	Long task sequence with side tasks (Fig. A.1l)
gridcat	i	401	401	115.71	4	Merge of pairs of 300 MiB files
crossv	i	94	90	8.52	5	Cross validation
crossvx	i	200	200	32.66	5	Several instances of cross validation
fastcrossv	i	94	90	8.52	5	Same as *crossv* but tasks are 50× shorter
mapreduce	i	321	25 760	439.06	3	Map-reduce pattern
nestedcrossv	i	266	270	28.41	8	Nested cross validation
montage	p	77	150	0.21	6	Montage workflow from Pegasus
cybershake	p	104	106	0.84	4	Cybershake workflow from Pegasus
epigenomics	p	204	305	1.36	8	Epigenomics workflow from Pegasus
ligo	p	186	186	0.11	6	Ligo workflow from Pegasus
sipht	p	64	136	0.12	5	Sipht workflow from Pegasus

D = Dataset (e = elementary, i = irw, p = pegasus); #T = Number of tasks; #O = Number of outputs; TS = Sum of all output object sizes (GiB); LP = longest oriented path in the graph

Table A.1: Estee scheduler benchmark task graph properties

Task graph	#T	#I	S	AD	LP	API
merge-10K	10 001	10 000	0.027	0.006	1	F
merge-15K	15 001	15 000	0.027	0.006	1	F
merge-20K	20 001	20 000	0.027	0.006	1	F
merge-25K	25 001	25 000	0.027	0.006	1	F
merge-30K	30 001	30 000	0.027	0.006	1	F
merge-50K	50 001	50 000	0.027	0.006	1	F
merge-100K	100 001	100 000	0.027	0.006	1	F
merge_slow-5K-0.1	5001	5000	0.023	100	1	F
merge_slow-20K-0.1	20 001	20 000	0.023	100	1	F
tree-15	32 767	32 766	0.027	0.007	14	F
xarray-25	552	862	55.7	3.1	10	X
xarray-5	9258	14 976	3.3	0.4	10	X
bag-25K-10	236	415	292	1233	6	B
bag-25K-100	21 631	41 430	3.2	13.9	8	B
bag-25K-200	86 116	165 715	0.8	3.6	9	B
bag-25K-50	5458	10 357	12.6	54.9	7	B
bag-50K-50	5458	10 357	25.2	214	7	B
numpy-50K-10	209	228	70 108	169	7	A
numpy-50K-100	19 334	21 783	760	2.6	10	A
numpy-50K-200	77 067	86 966	191	0.9	11	A
numpy-50K-50	4892	5491	2999	8.3	9	A
groupby-2880-1S-16H	22 842	31 481	1005	11.9	9	D
groupby-2880-1S-8H	45 674	62 953	503	7.7	9	D
groupby-1440-1S-1H	182 682	251 801	64.3	3.8	10	D
groupby-1440-1S-8H	22 842	31 481	503	7.7	9	D
groupby-360-1S-1H	45 674	62 953	64.3	3.8	9	D
groupby-360-1S-8H	5714	7873	503	8.0	8	D
groupby-90-1S-1H	11 424	15 743	64.3	3.9	8	D
groupby-90-1S-8H	1434	1973	501	7.7	7	D
join-1-1S-1H	673	1224	15.3	33.0	5	D
join-1-1S-1T	72 001	125 568	3.7	1.7	11	D
join-1-2s-1H	673	1224	9.3	9.8	5	D
vectorizer-1M-300	301	0	10 226	1504	0	F
wordbag-100K-50	250	200	5136	301	2	F

#T = Number of tasks; #I = Number of dependencies;
S = Average task output size [KiB]; AD = Average task duration [ms];
LP = longest oriented path in the graph;
D = DataFrame; B = Bag; A = Arrays; F = Futures; X = XArray

Table A.2: Properties of Dask benchmark task graphs