Ergonomics and efficiency of workflows on HPC clusters

Jakub Beránek

Supervisor
Ing. Jan Martinovič, Ph.D.

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Thesis goal

Design approches for executing workflows
on HPC clusters in an easy & efficient way

Thesis objectives

Thesis objectives

1. Identify HPC workflow challenges

Thesis objectives

1. Identify HPC workflow challenges
2. Design approaches for overcoming them

Thesis objectives

1. Identify HPC workflow challenges
2. Design approaches for overcoming them
3. Implement them in a task runtime

Thesis objectives

1. Identify HPC workflow challenges
2. Design approaches for overcoming them
3. Implement them in a task runtime
4. Analyze results on real use-cases

Ergonomics and efficiency
of **workflows on HPC clusters**

Workflows

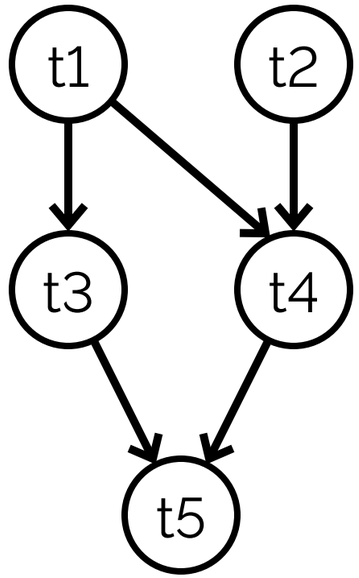
= task graphs, pipelines, task DAGs*, …

* Directed Acyclic Graph

Workflows

= task graphs, pipelines, task DAGs*, …

Popular programming model

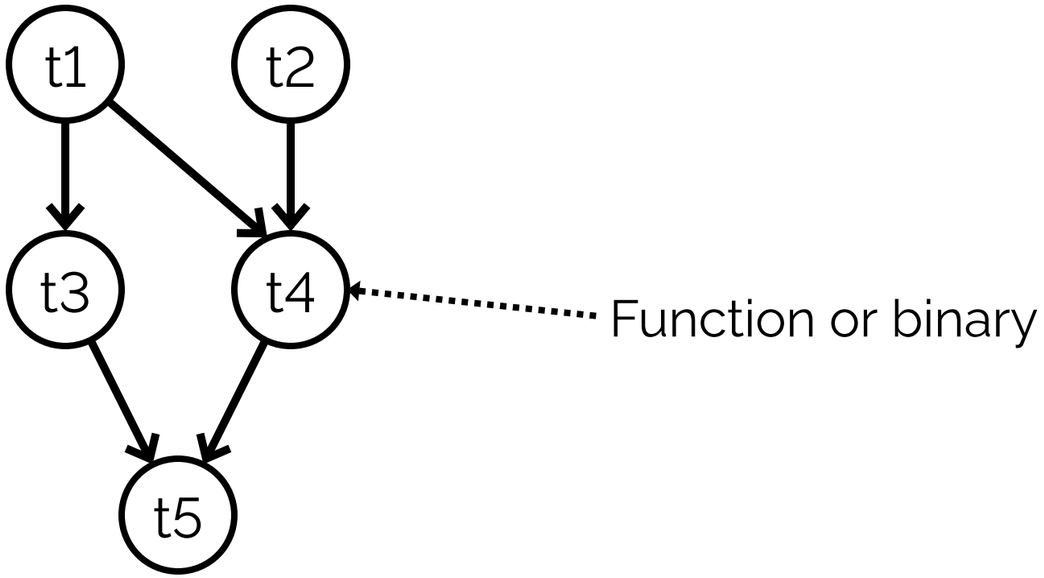


* Directed Acyclic Graph

Workflows

= task graphs, pipelines, task DAGs*, …

Popular programming model

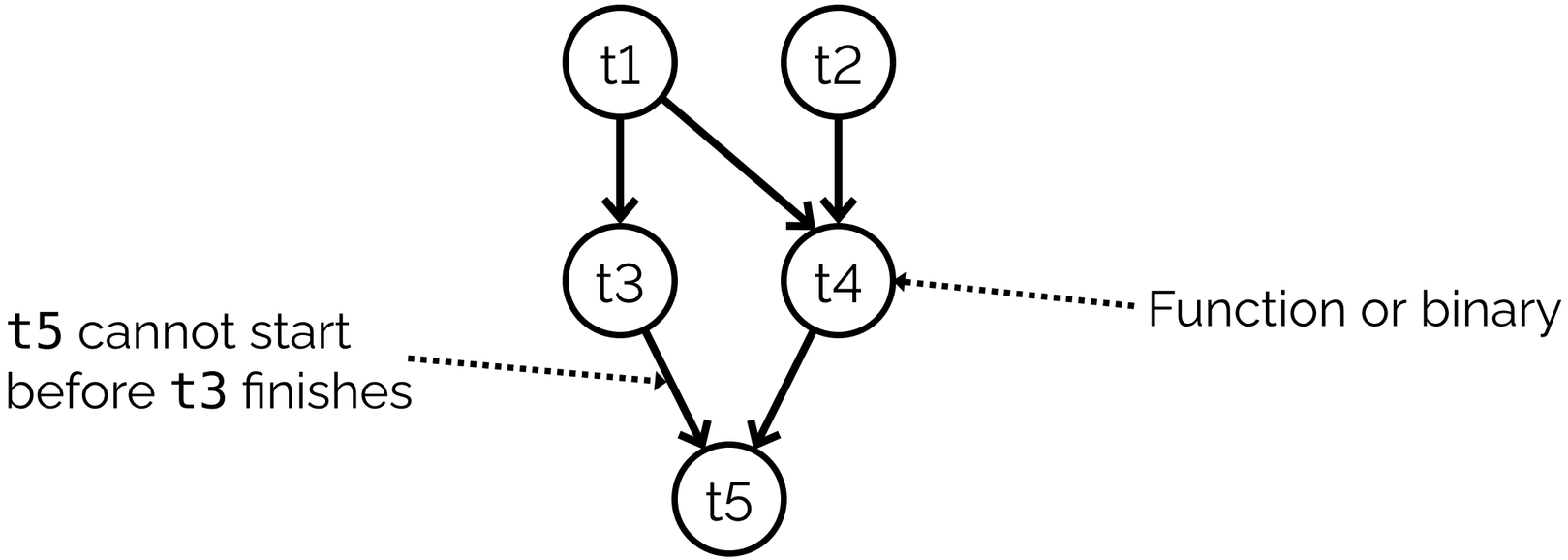


* Directed Acyclic Graph

Workflows

= task graphs, pipelines, task DAGs*, …

Popular programming model



* Directed Acyclic Graph

Benefits of task-based programming

- **Implicit parallelism**
  - Create a DAG vs. explicit MPI calls
  - Extracted by a task runtime

Benefits of task-based programming

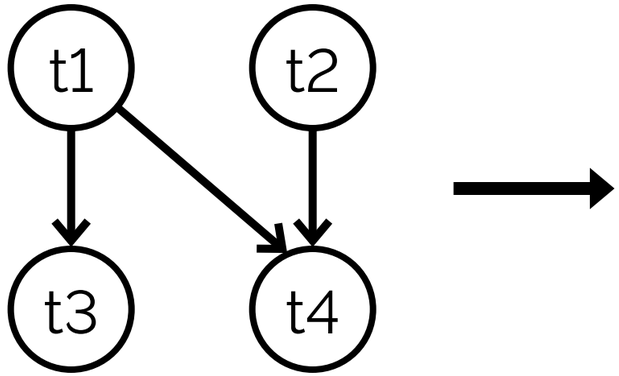
- **Implicit parallelism**
  - Create a DAG vs. explicit MPI calls
  - Extracted by a task runtime
- High-level description
  - Python/DSL vs. C/C++

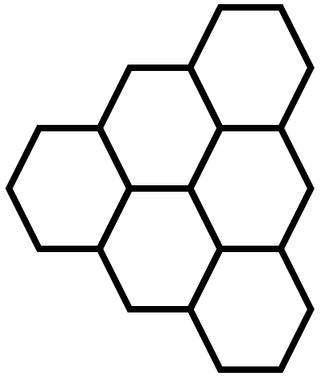
Benefits of task-based programming

- **Implicit parallelism**
  - Create a DAG vs. explicit MPI calls
  - Extracted by a task runtime
- High-level description
  - Python/DSL vs. C/C++
- Portability

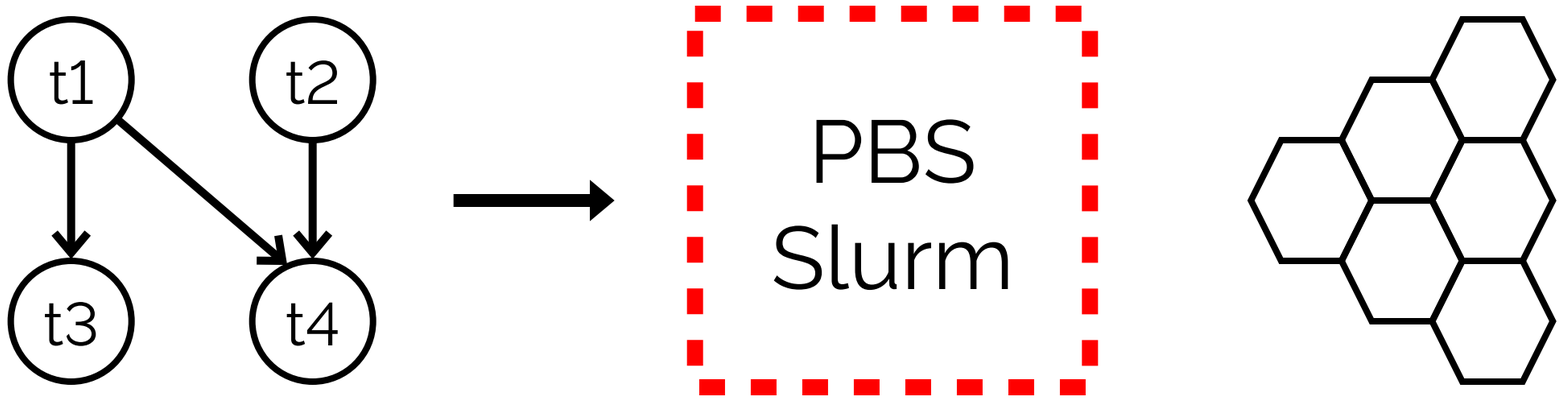
Workflow challenges on HPC clusters

Allocation manager

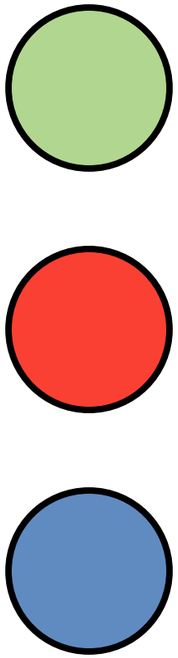


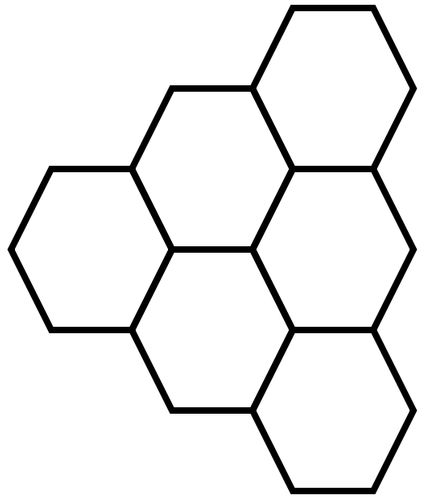


Allocation manager

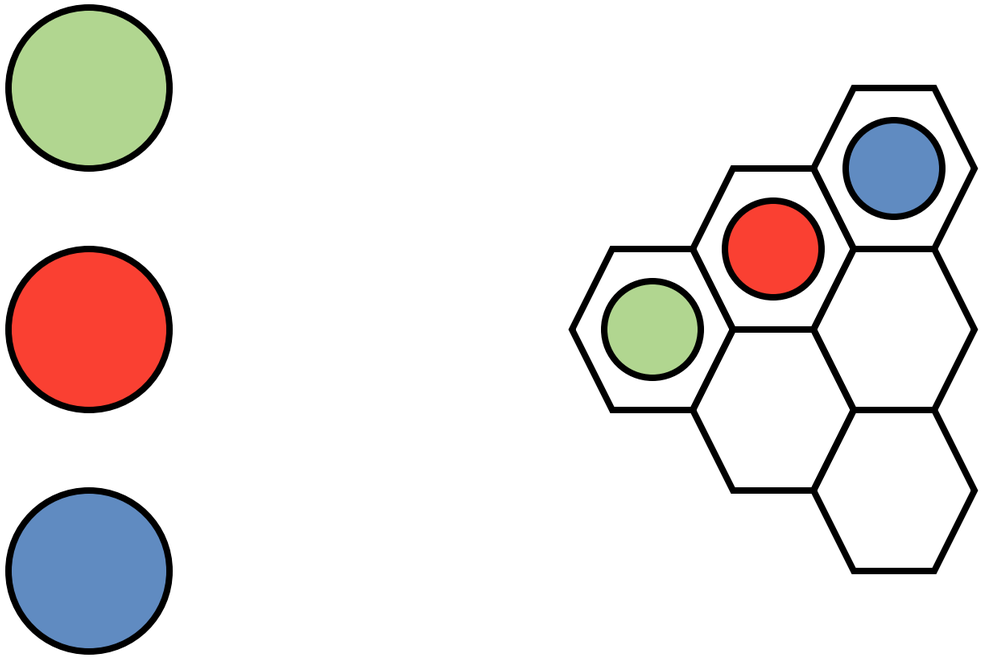


A few large tasks ✅

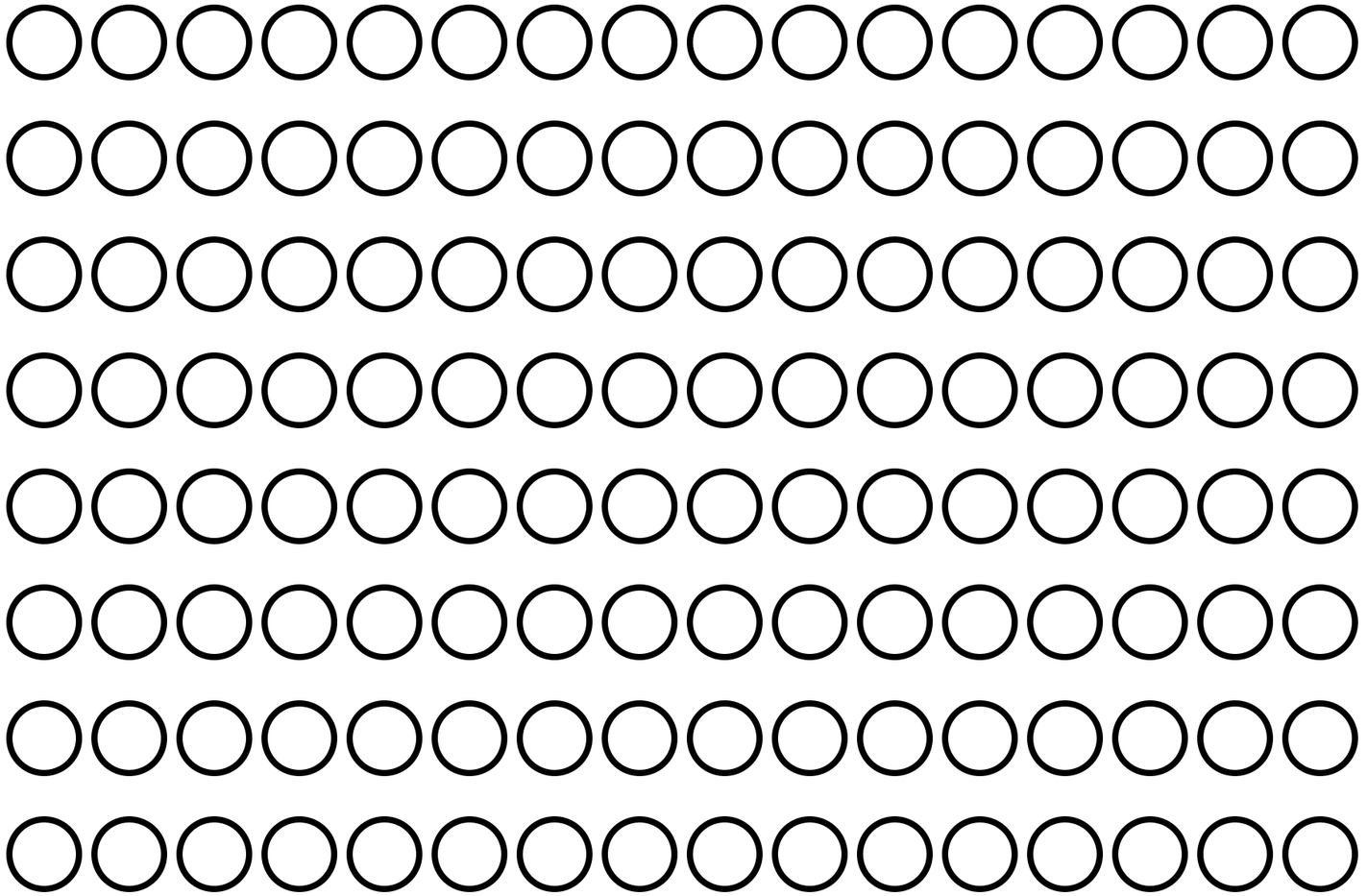




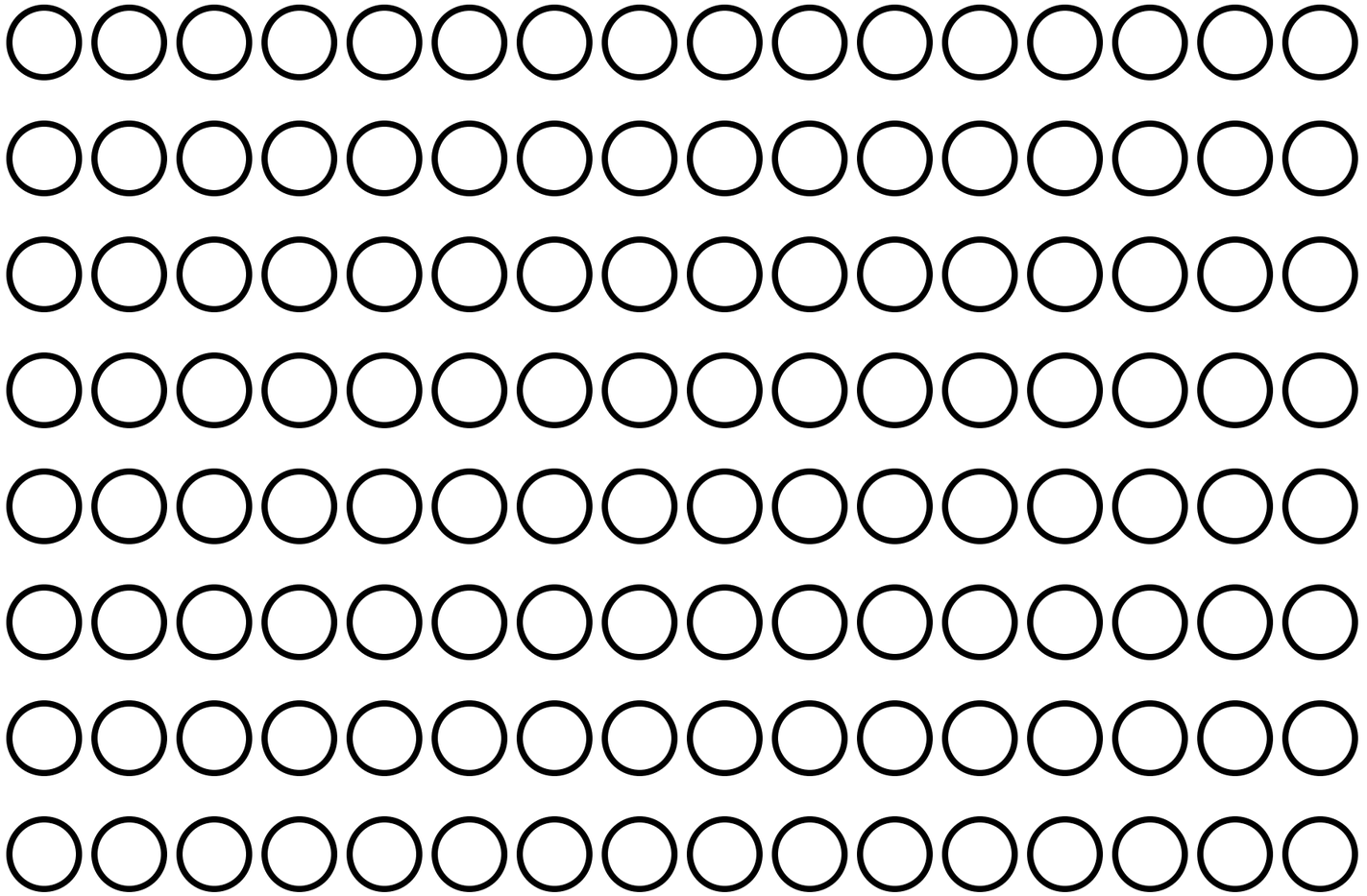
A few large tasks ✅



Many tasks

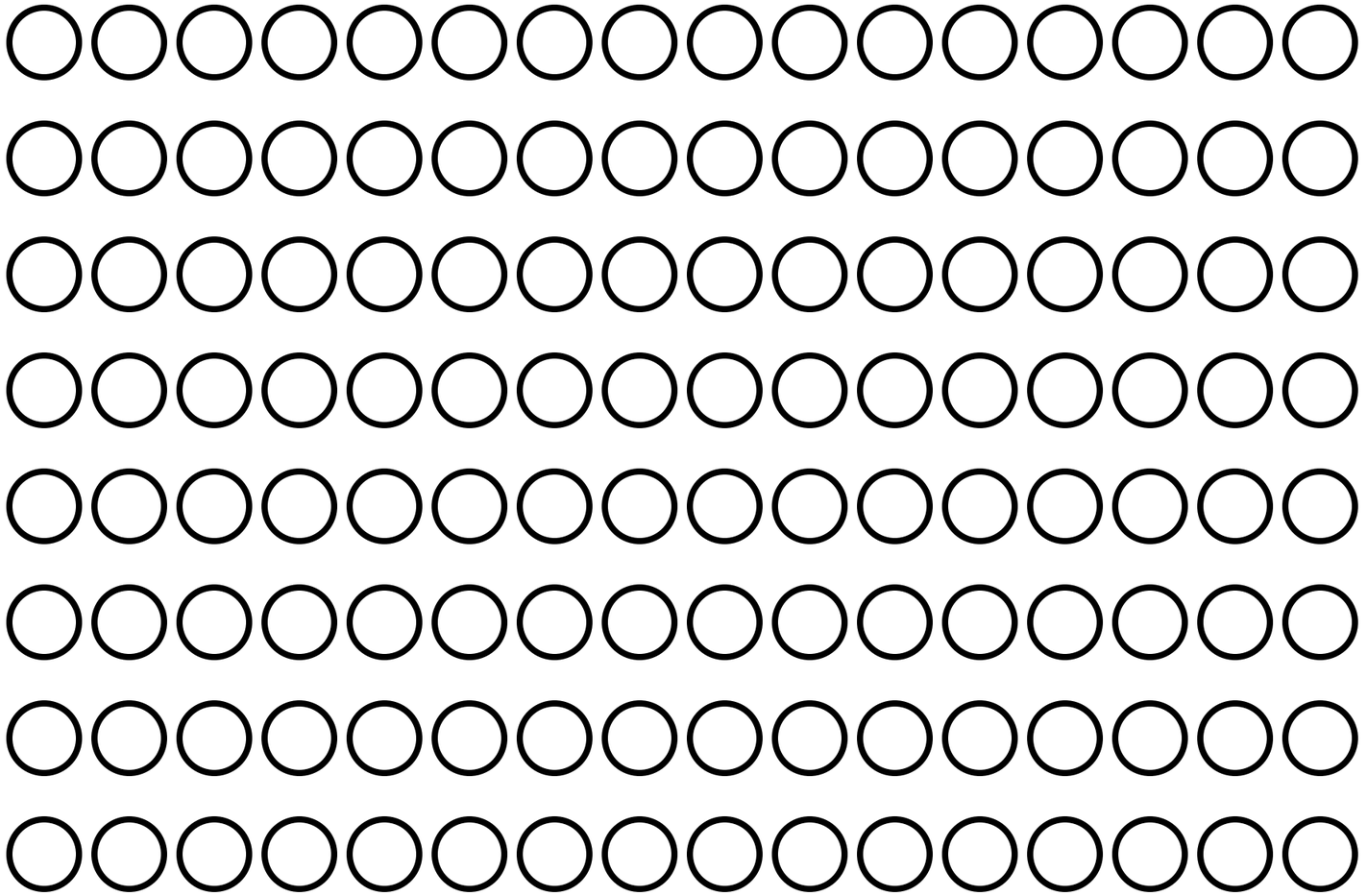


Many tasks



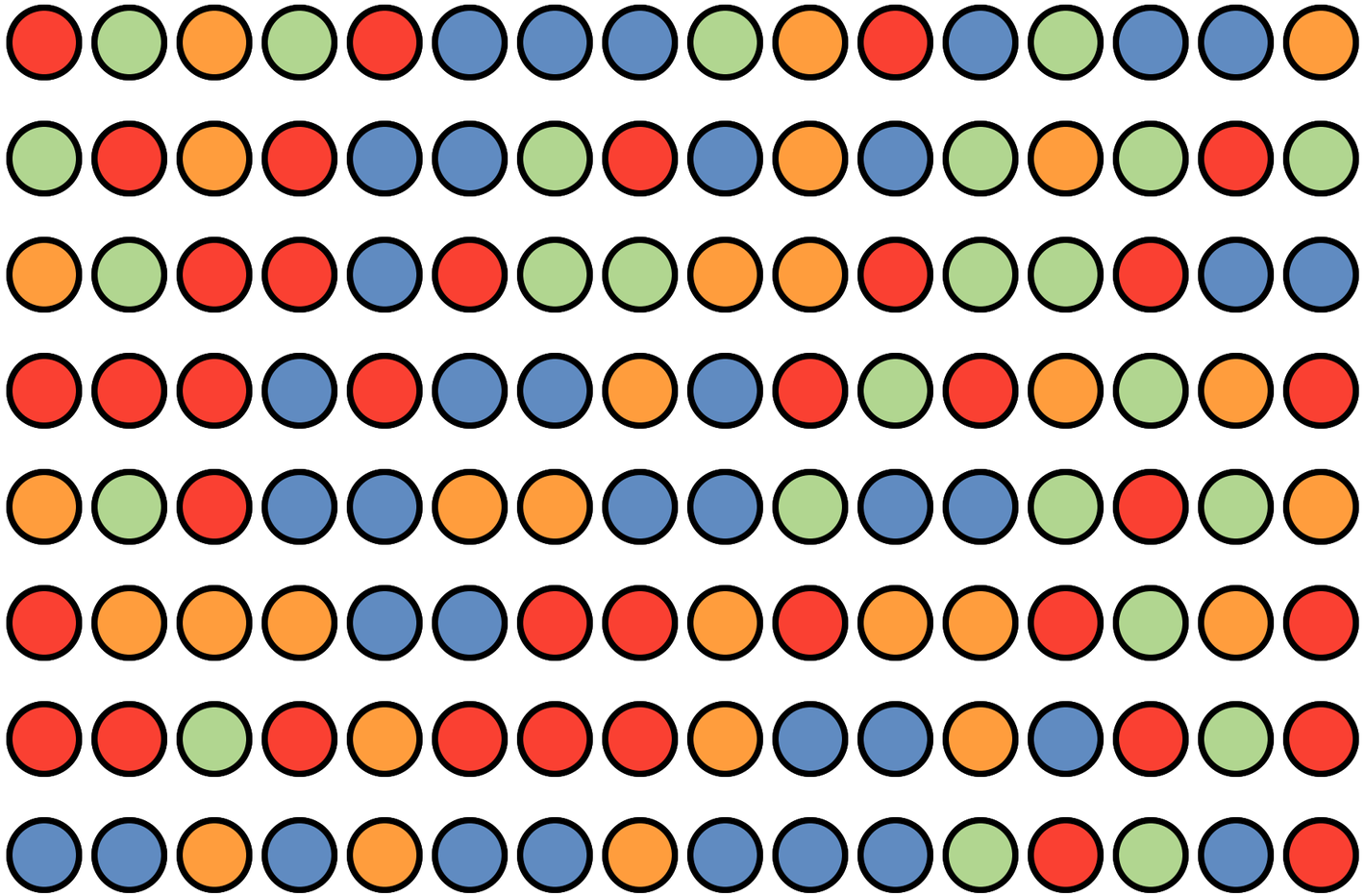
```
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
...
```

Many tasks

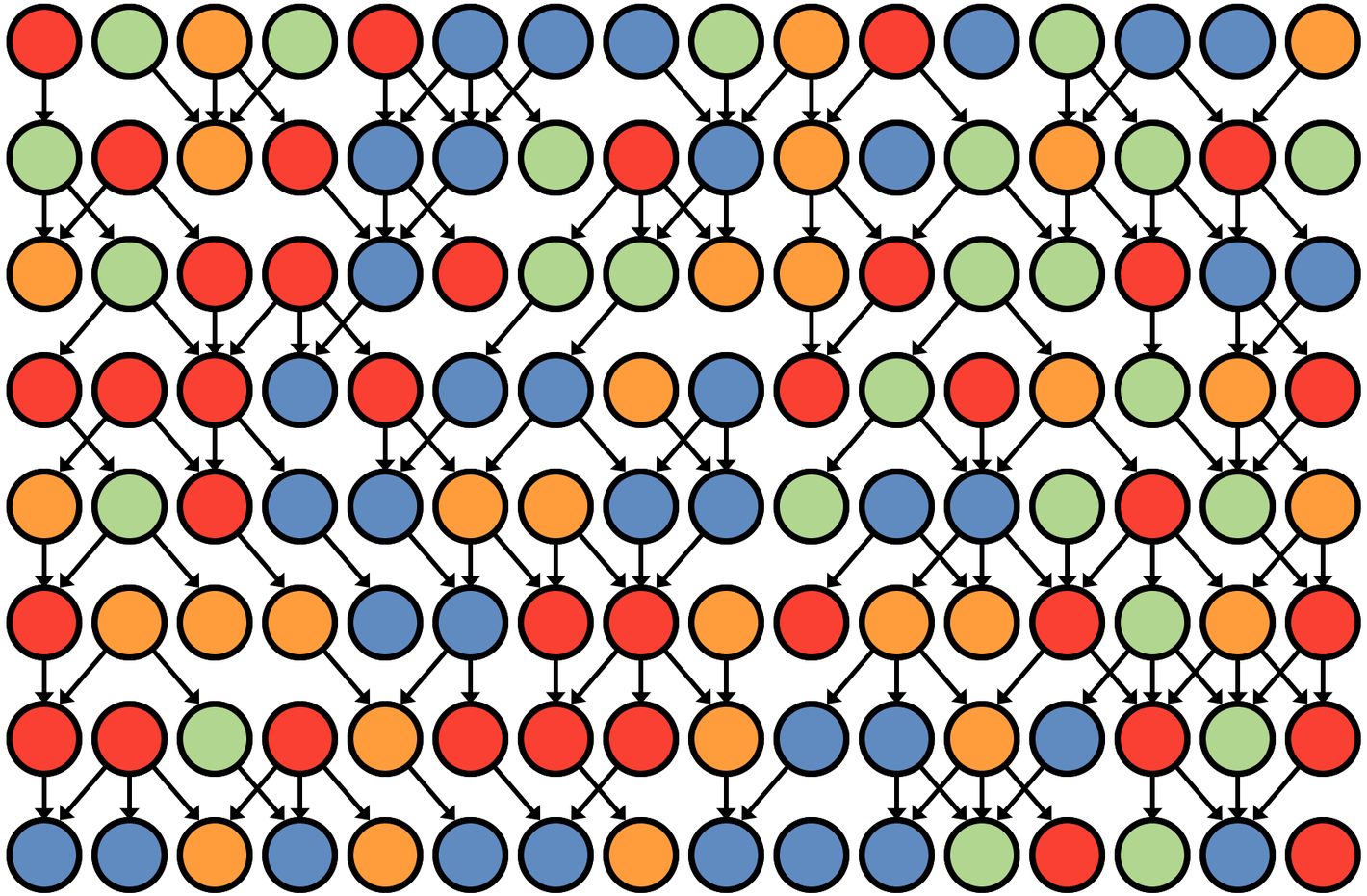


sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
sbatch script.sh
...

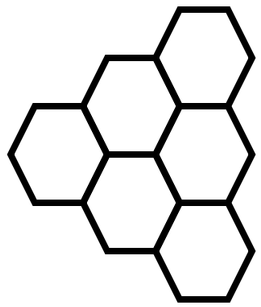
Heterogeneous tasks



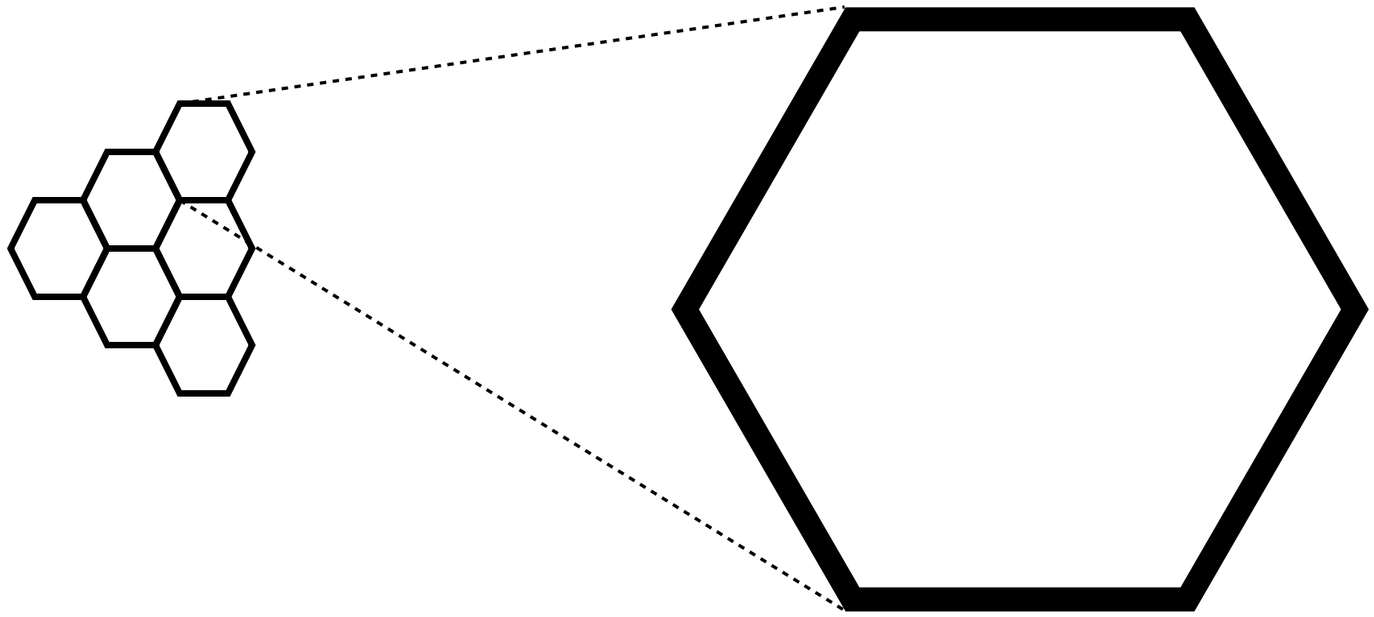
Task dependencies



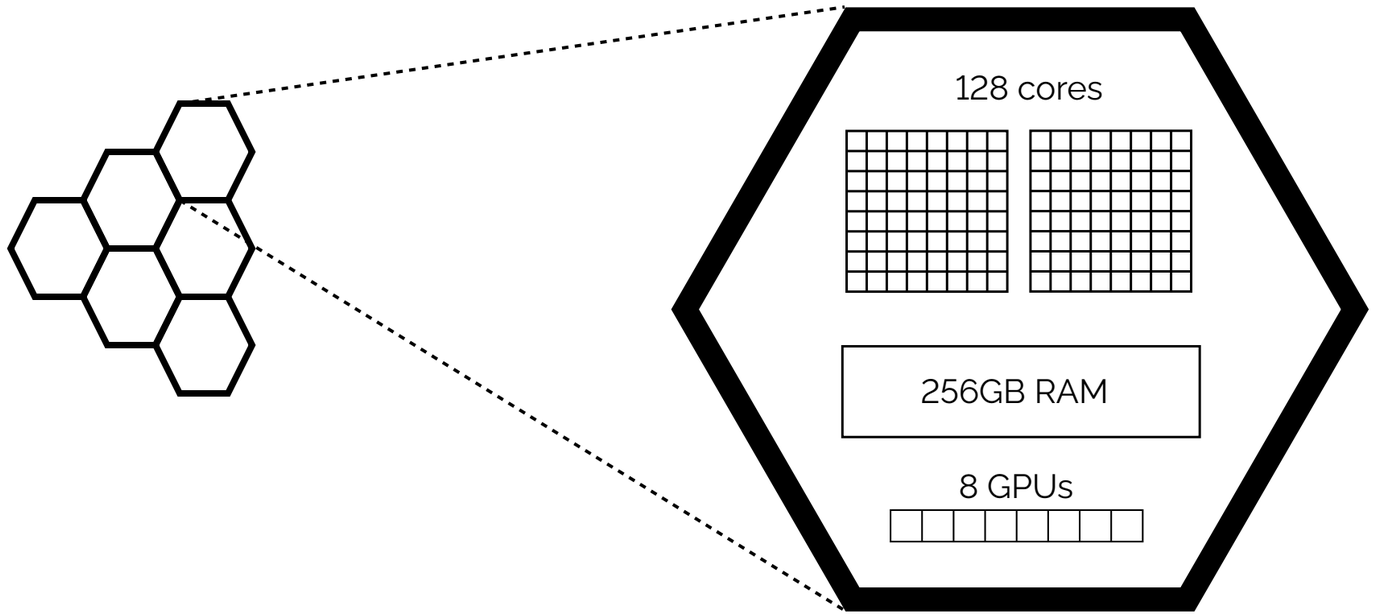
Heterogeneous clusters



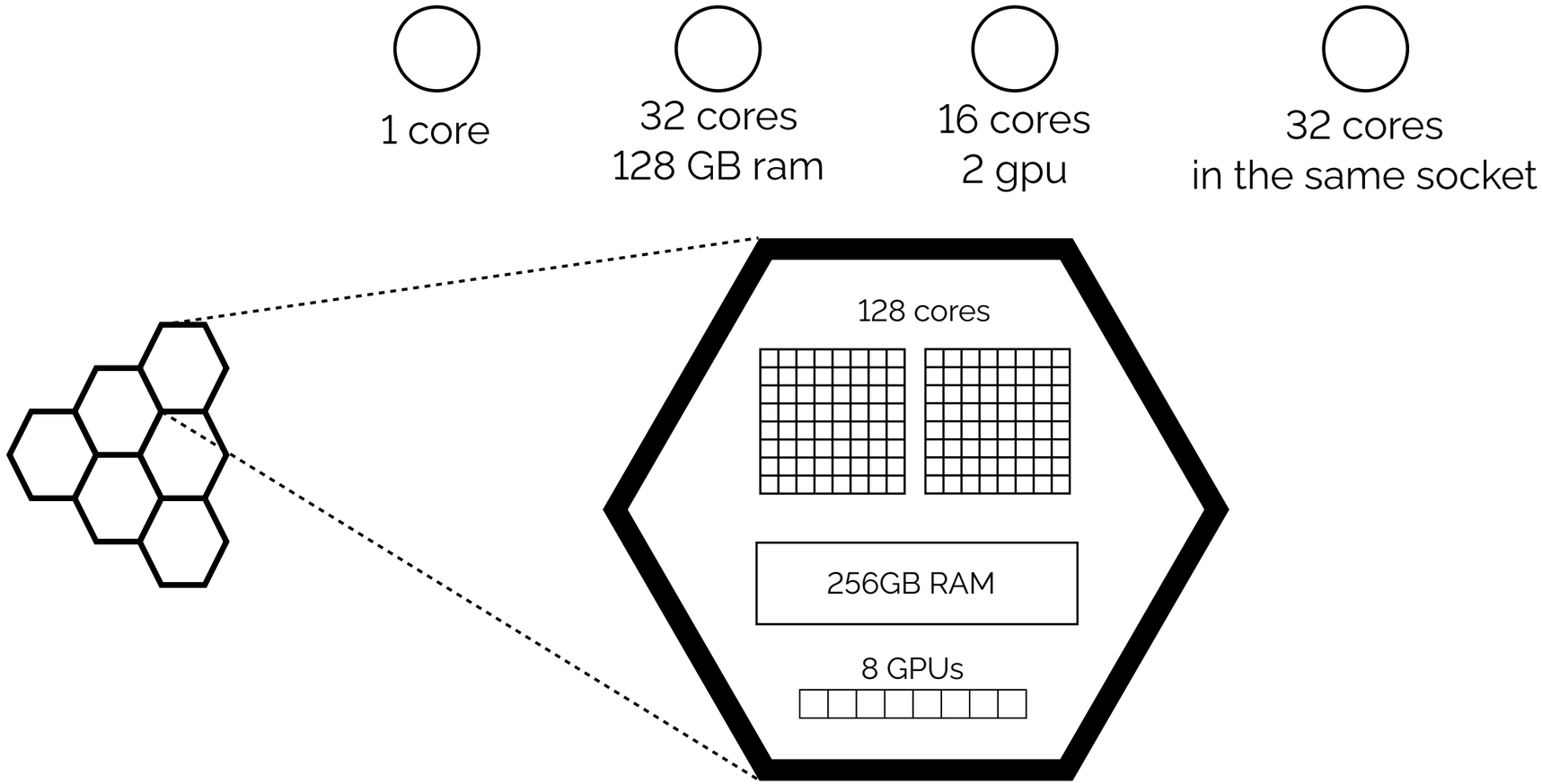
Heterogeneous clusters



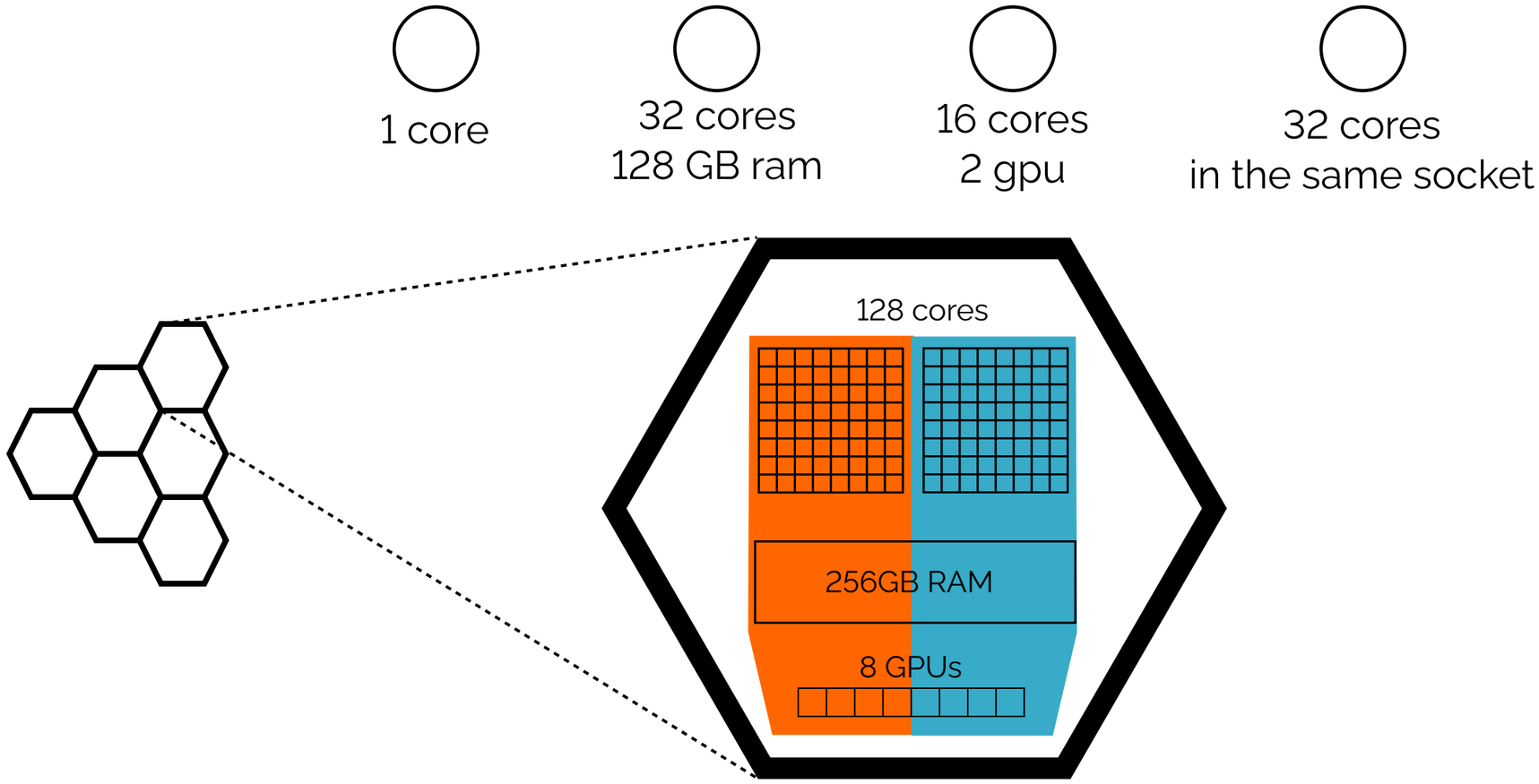
Heterogeneous clusters



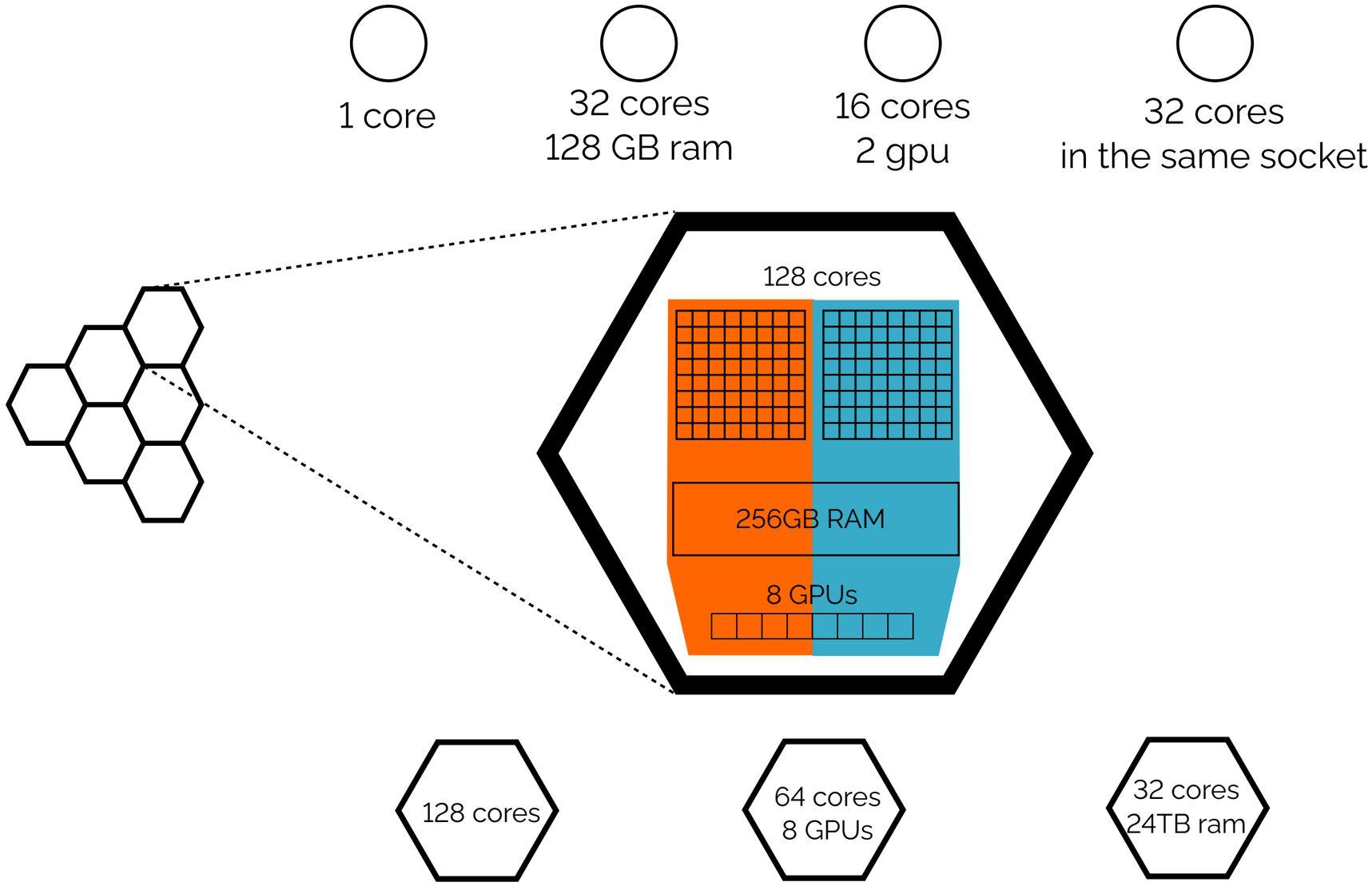
Heterogeneous clusters



Heterogeneous clusters



Heterogeneous clusters



Other workflow challenges on HPC

Other workflow challenges on HPC

- Scalability

Other workflow challenges on HPC

- Scalability
- Multi-node tasks

Other workflow challenges on HPC

- Scalability
- Multi-node tasks
- Fault tolerance

Other workflow challenges on HPC

- Scalability
- Multi-node tasks
- Fault tolerance
- Iterative computation

Ergonomics and **efficiency**
of workflows on HPC clusters

The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04438-y

Check for updates

Analysis of workflow schedulers in simulated distributed environments

Jakub Beránek[1] · Stanislav Böhm[1] · Vojtěch Cima[1]

Accepted: 10 March 2022

Abstract
Task graphs provide a simple way to describe scientific workflows (sets of tasks with dependencies) that can be executed on both HPC clusters and in the cloud. An important aspect of executing such graphs is the used scheduling algorithm. Many scheduling heuristics have been proposed in existing works; nevertheless, they are often tested in oversimplified environments. We provide an extensible simulation environment designed for prototyping and benchmarking task schedulers, which contains implementations of various scheduling algorithms and is open-sourced, in order to be fully reproducible. We use this environment to perform a comprehensive analysis of workflow scheduling algorithms with a focus on quantifying the effect of scheduling challenges that have so far been mostly neglected, such as delays between scheduler invocations or partially unknown task durations. Our results indicate that network models used by many previous works might produce results that are off by an order of magnitude in comparison to a more realistic model. Additionally, we show that certain implementation details of scheduling algorithms which are often neglected can have a large effect on the scheduler's performance, and they should thus be described in great detail to enable proper evaluation.

Keywords Distributed computing · DAG scheduling · Task Scheduling · Network models

Jakub Beránek, Stanislav Böhm and Vojtech Cima these authors contributed equally to this work.

✉ Jakub Beránek
jakub.beranek@vsb.cz

Stanislav Böhm
stanislav.bohm@vsb.cz

Vojtěch Cima
vojtech.cima@vsb.cz

[1] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic

Published online: 14 April 2022

Springer

Analysis of workflow schedulers in simulated distributed environments

Jakub Beránek, Ada Böhm, Vojtěch Cima
(The Journal of Supercomputing 2022)

The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04438-y

Check for updates

Analysis of workflow schedulers in simulated distributed environments

Jakub Beránek[1] · Stanislav Böhm[1] · Vojtěch Cima[1]

Accepted: 10 March 2022

Abstract
Task graphs provide a simple way to describe scientific workflows (sets of tasks with dependencies) that can be executed on both HPC clusters and in the cloud. An important aspect of executing such graphs is the used scheduling algorithm. Many scheduling heuristics have been proposed in existing works; nevertheless, they are often tested in oversimplified environments. We provide an extensible simulation environment designed for prototyping and benchmarking task schedulers, which contains implementations of various scheduling algorithms and is open-sourced, in order to be fully reproducible. We use this environment to perform a comprehensive analysis of workflow scheduling algorithms with a focus on quantifying the effect of scheduling challenges that have so far been mostly neglected, such as delays between scheduler invocations or partially unknown task durations. Our results indicate that network models used by many previous works might produce results that are off by an order of magnitude in comparison to a more realistic model. Additionally, we show that certain implementation details of scheduling algorithms which are often neglected can have a large effect on the scheduler's performance, and they should thus be described in great detail to enable proper evaluation.

Keywords Distributed computing · DAG scheduling · Task Scheduling · Network models

Jakub Beránek, Stanislav Böhm and Vojtech Cima these authors contributed equally to this work.

✉ Jakub Beránek
jakub.beranek@vsb.cz

Stanislav Böhm
stanislav.bohm@vsb.cz

Vojtěch Cima
vojtech.cima@vsb.cz

[1] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic

Published online: 14 April 2022

Springer

Analysis of workflow schedulers in simulated distributed environments

Jakub Beránek, Ada Böhm, Vojtěch Cima
(The Journal of Supercomputing 2022)

- Which scheduling algorithms are the best for HPC?

The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04438-y

Check for updates

Analysis of workflow schedulers in simulated distributed environments

Jakub Beránek[1] · Stanislav Böhm[1] · Vojtěch Cima[1]

Accepted: 10 March 2022

Abstract
Task graphs provide a simple way to describe scientific workflows (sets of tasks with dependencies) that can be executed on both HPC clusters and in the cloud. An important aspect of executing such graphs is the used scheduling algorithm. Many scheduling heuristics have been proposed in existing works; nevertheless, they are often tested in oversimplified environments. We provide an extensible simulation environment designed for prototyping and benchmarking task schedulers, which contains implementations of various scheduling algorithms and is open-sourced, in order to be fully reproducible. We use this environment to perform a comprehensive analysis of workflow scheduling algorithms with a focus on quantifying the effect of scheduling challenges that have so far been mostly neglected, such as delays between scheduler invocations or partially unknown task durations. Our results indicate that network models used by many previous works might produce results that are off by an order of magnitude in comparison to a more realistic model. Additionally, we show that certain implementation details of scheduling algorithms which are often neglected can have a large effect on the scheduler's performance, and they should thus be described in great detail to enable proper evaluation.

Keywords Distributed computing · DAG scheduling · Task Scheduling · Network models

Jakub Beránek, Stanislav Böhm and Vojtech Cima these authors contributed equally to this work.

✉ Jakub Beránek
jakub.beranek@vsb.cz

Stanislav Böhm
stanislav.bohm@vsb.cz

Vojtěch Cima
vojtech.cima@vsb.cz

[1] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic

Published online: 14 April 2022

Springer

Analysis of workflow schedulers in simulated distributed environments

Jakub Beránek, Ada Böhm, Vojtěch Cima
(The Journal of Supercomputing 2022)

- Which scheduling algorithms are the best for HPC?
- Which factors affect scheduling the most?

The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04438-y

Check for updates

Analysis of workflow schedulers in simulated distributed environments

Jakub Beránek[1] · Stanislav Böhm[1] · Vojtěch Cima[1]

Accepted: 10 March 2022

Abstract
Task graphs provide a simple way to describe scientific workflows (sets of tasks with dependencies) that can be executed on both HPC clusters and in the cloud. An important aspect of executing such graphs is the used scheduling algorithm. Many scheduling heuristics have been proposed in existing works; nevertheless, they are often tested in oversimplified environments. We provide an extensible simulation environment designed for prototyping and benchmarking task schedulers, which contains implementations of various scheduling algorithms and is open-sourced, in order to be fully reproducible. We use this environment to perform a comprehensive analysis of workflow scheduling algorithms with a focus on quantifying the effect of scheduling challenges that have so far been mostly neglected, such as delays between scheduler invocations or partially unknown task durations. Our results indicate that network models used by many previous works might produce results that are off by an order of magnitude in comparison to a more realistic model. Additionally, we show that certain implementation details of scheduling algorithms which are often neglected can have a large effect on the scheduler's performance, and they should thus be described in great detail to enable proper evaluation.

Keywords Distributed computing · DAG scheduling · Task Scheduling · Network models

Jakub Beránek, Stanislav Böhm and Vojtech Cima these authors contributed equally to this work.

✉ Jakub Beránek
jakub.beranek@vsb.cz

Stanislav Böhm
stanislav.bohm@vsb.cz

Vojtěch Cima
vojtech.cima@vsb.cz

[1] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic

Published online: 14 April 2022

Springer

Analysis of workflow schedulers in simulated distributed environments

Jakub Beránek, Ada Böhm, Vojtěch Cima
(The Journal of Supercomputing 2022)

- Which scheduling algorithms are the best for HPC?
- Which factors affect scheduling the most?
- How can we simplify scheduler prototyping?

ESTEE

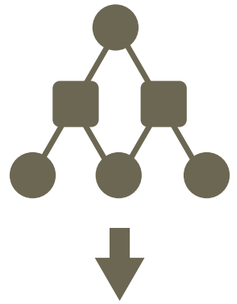
Framework for simulating workflow execution

ESTEE

Framework for simulating workflow execution

```
dag = TaskGraph()
t0 = dag.new_task(duration=1, cpus=1, output_size=50)
t1 = dag.new_task(duration=1, cpus=1)
t1.add_input(t0)
t2 = dag.new_task(duration=1, cpus=1)
t2.add_input(t0)
```

Task graph

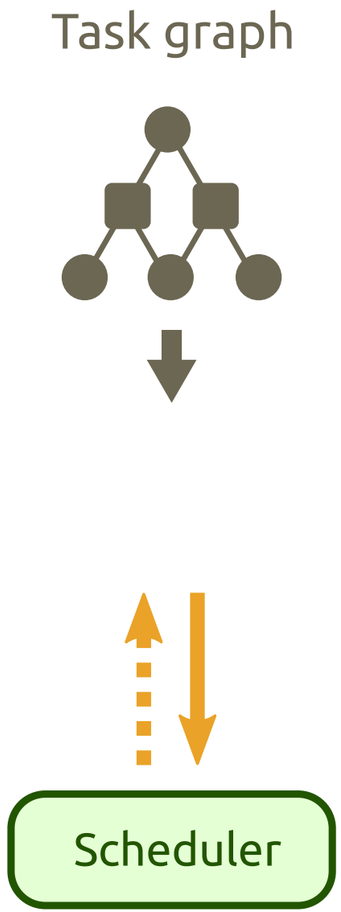


ESTEE

Framework for simulating workflow execution

```
dag = TaskGraph()
t0 = dag.new_task(duration=1, cpus=1, output_size=50)
t1 = dag.new_task(duration=1, cpus=1)
t1.add_input(t0)
t2 = dag.new_task(duration=1, cpus=1)
t2.add_input(t0)

scheduler = BlevelGtScheduler()
```

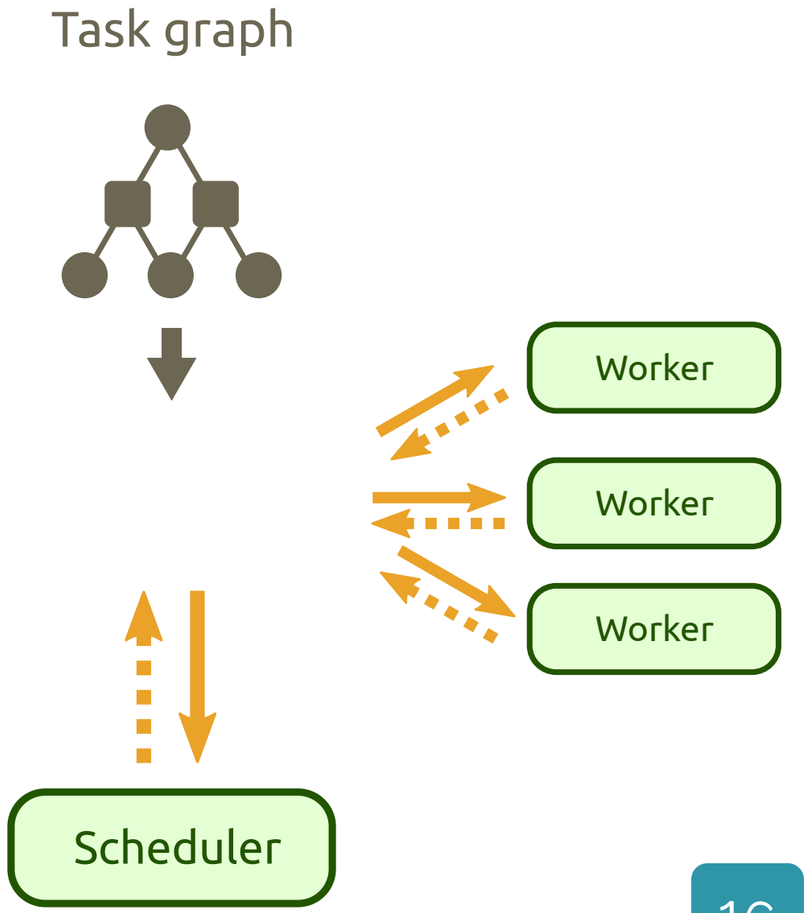


ESTEE

Framework for simulating workflow execution

```
dag = TaskGraph()
t0 = dag.new_task(duration=1, cpus=1, output_size=50)
t1 = dag.new_task(duration=1, cpus=1)
t1.add_input(t0)
t2 = dag.new_task(duration=1, cpus=1)
t2.add_input(t0)

scheduler = BlevelGtScheduler()
cluster   = [Worker(cpus=8) for _ in range(3)]
```

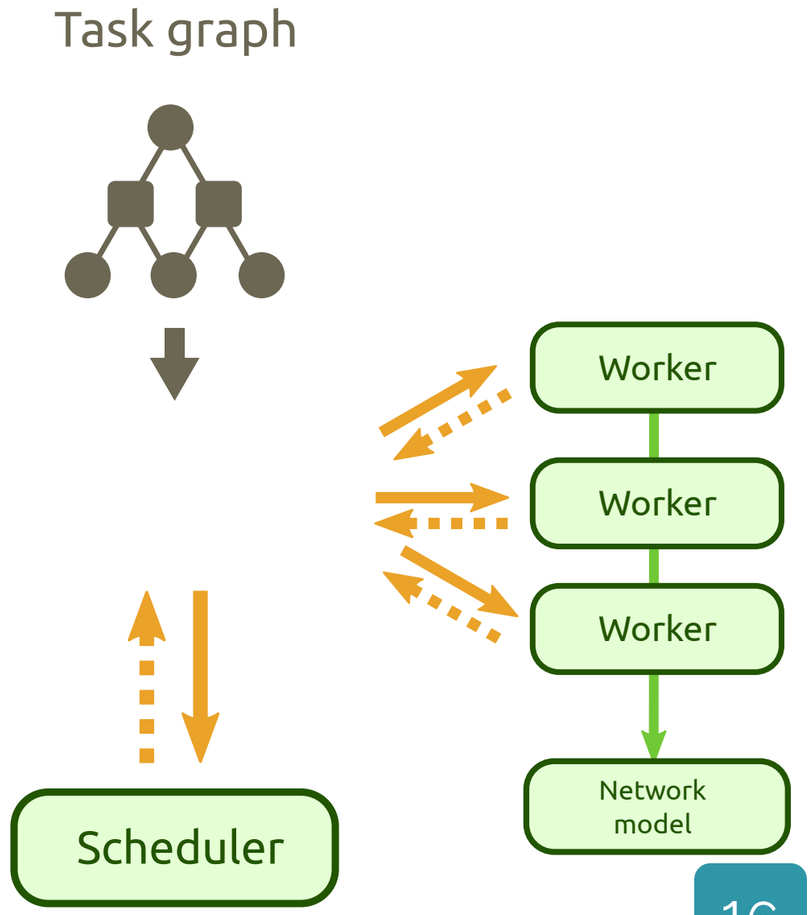


ESTEE

Framework for simulating workflow execution

```
dag = TaskGraph()
t0 = dag.new_task(duration=1, cpus=1, output_size=50)
t1 = dag.new_task(duration=1, cpus=1)
t1.add_input(t0)
t2 = dag.new_task(duration=1, cpus=1)
t2.add_input(t0)

scheduler = BlevelGtScheduler()
cluster   = [Worker(cpus=8) for _ in range(3)]
network   = MaxMinFlowNetModel(bandwidth=10*1024)
```



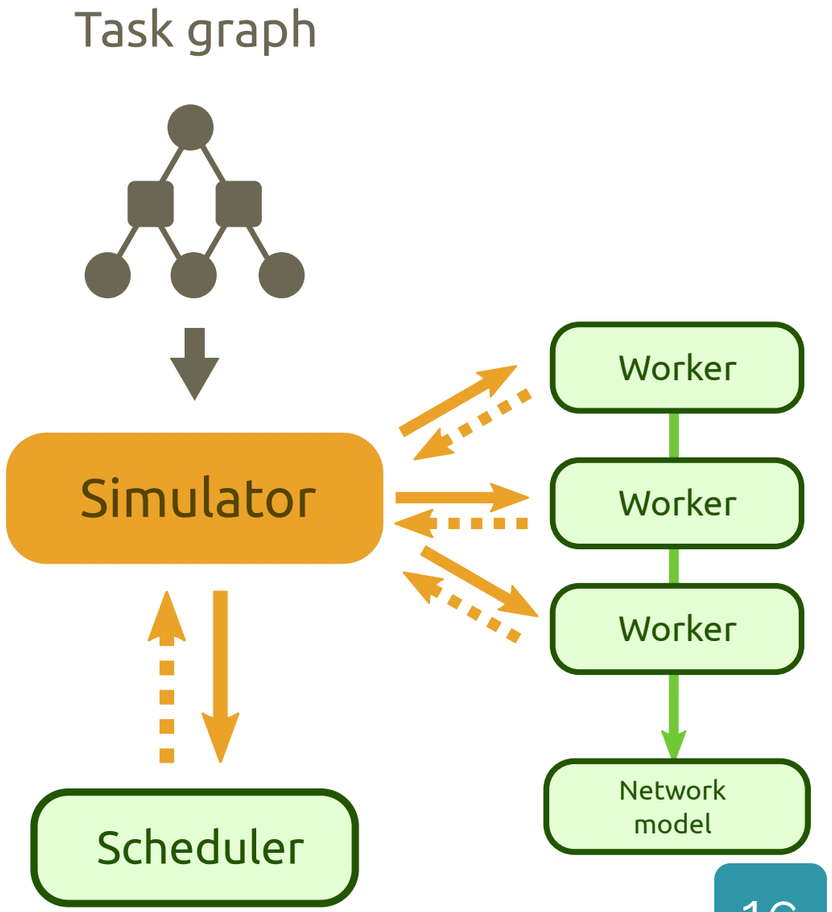
ESTEE

Framework for simulating workflow execution

```
dag = TaskGraph()
t0 = dag.new_task(duration=1, cpus=1, output_size=50)
t1 = dag.new_task(duration=1, cpus=1)
t1.add_input(t0)
t2 = dag.new_task(duration=1, cpus=1)
t2.add_input(t0)

scheduler = BlevelGtScheduler()
cluster   = [Worker(cpus=8) for _ in range(3)]
network   = MaxMinFlowNetModel(bandwidth=10*1024)

simulator = Simulator(task_graph, cluster, scheduler, network)
makespan  = simulator.run()
```



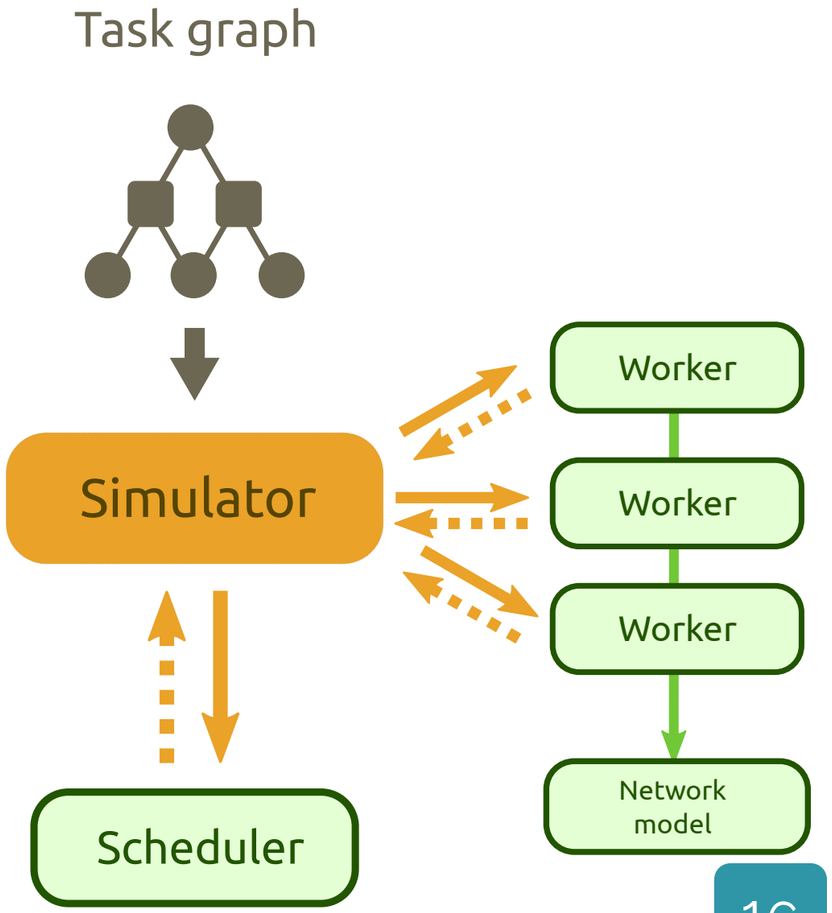
ESTEE

Framework for simulating workflow execution

```
dag = TaskGraph()
t0 = dag.new_task(duration=1, cpus=1, output_size=50)
t1 = dag.new_task(duration=1, cpus=1)
t1.add_input(t0)
t2 = dag.new_task(duration=1, cpus=1)
t2.add_input(t0)

scheduler = BlevelGtScheduler()
cluster   = [Worker(cpus=8) for _ in range(3)]
network   = MaxMinFlowNetModel(bandwidth=10*1024)

simulator = Simulator(task_graph, cluster, scheduler, network)
makespan  = simulator.run()
```



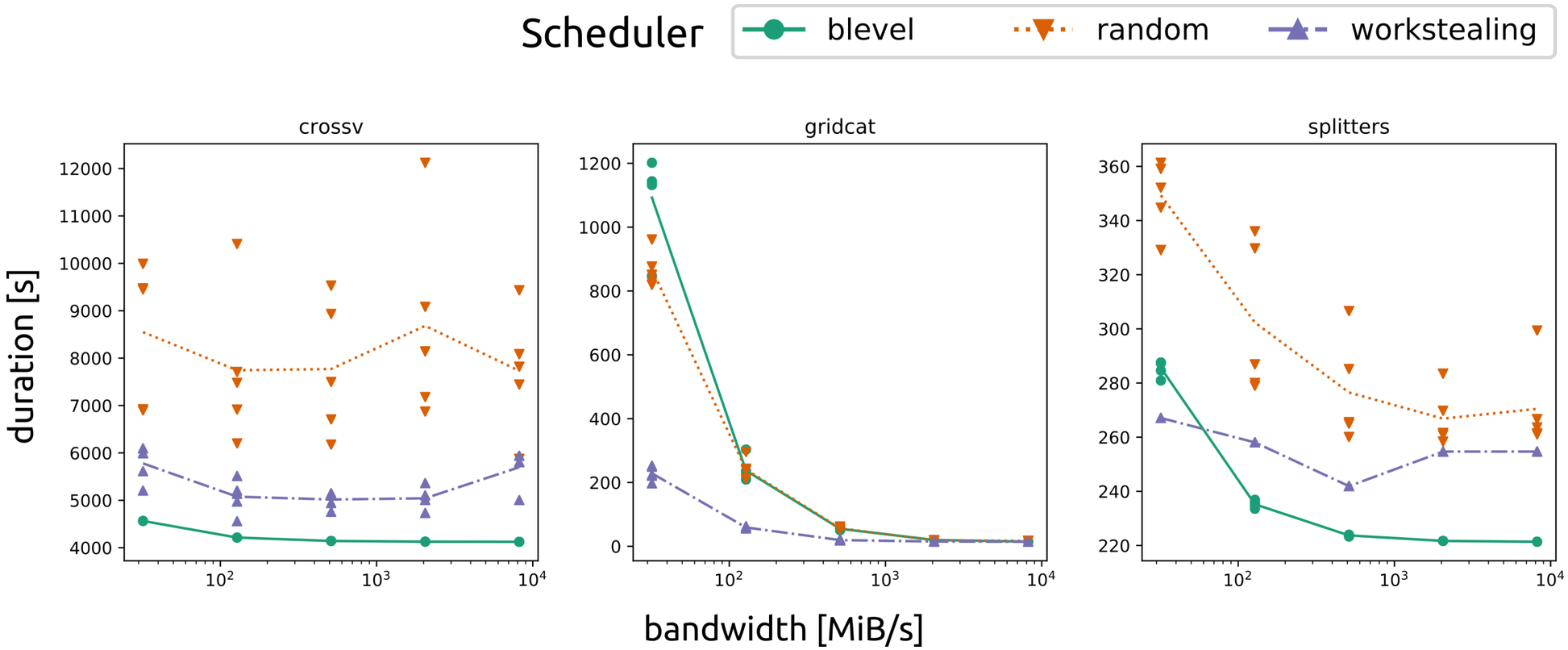
Scheduler analysis

- Compare scheduler performance

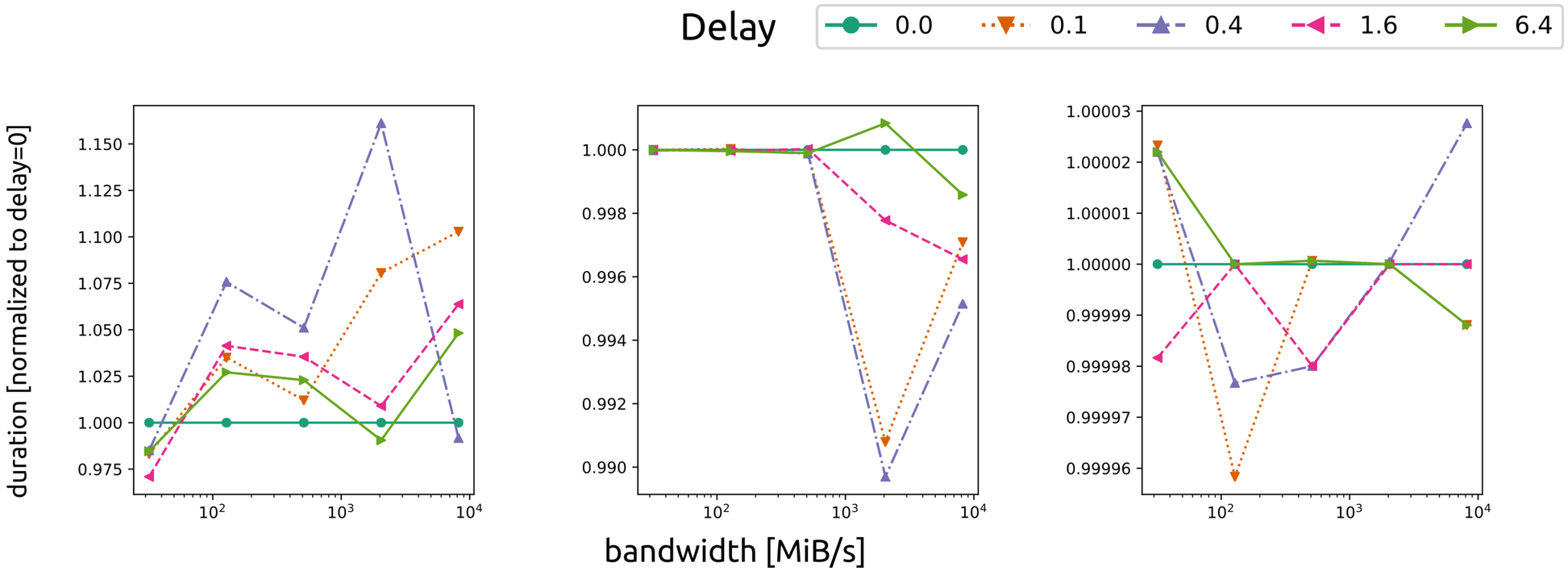
Scheduler analysis

- Compare scheduler performance
- Analyze factors that affect scheduling

Runtime vs. network speed



Scheduling frequency effect



Outcome

- Most competitive schedulers

Outcome

- Most competitive schedulers
  - B-level

Outcome

- Most competitive schedulers
  - B-level
  - Work-stealing

Outcome

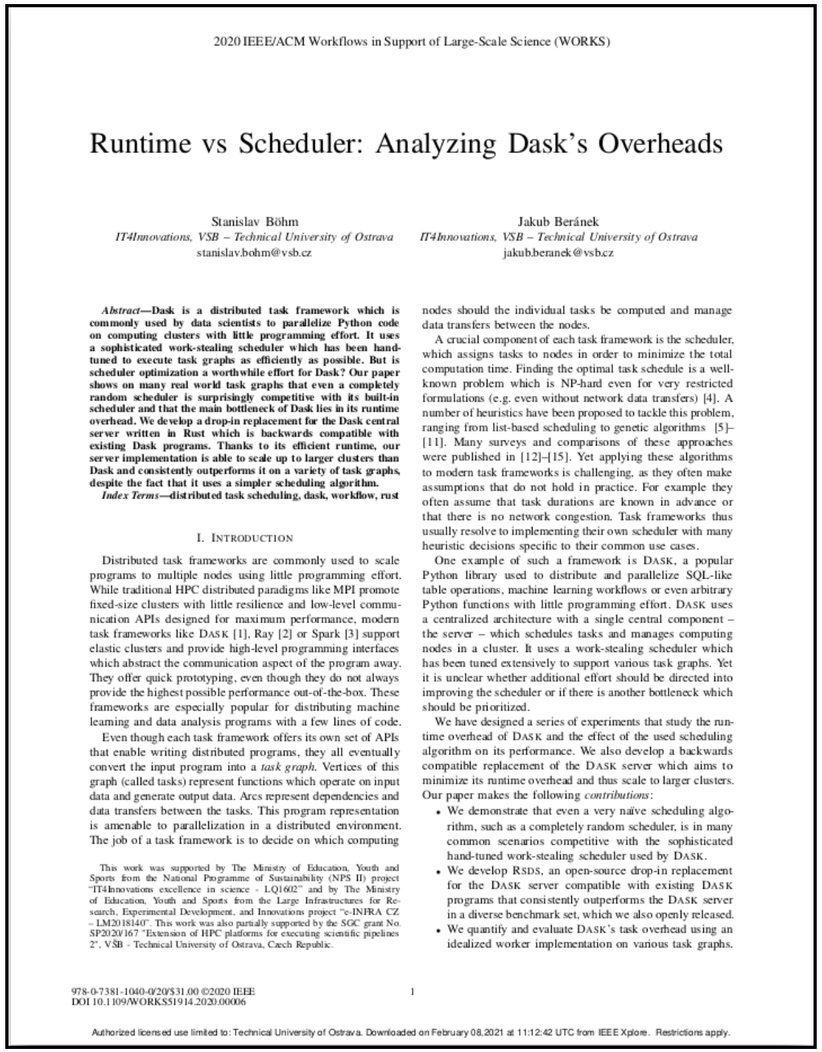
- Most competitive schedulers
  - B-level
  - Work-stealing
- Implementation details matter a lot

Outcome

- Most competitive schedulers
  - B-level
  - Work-stealing
- Implementation details matter a lot
- Open source scheduler simulator ESTEE
  -  `github.com/it4innovations/estee`

Outcome

- Most competitive schedulers
  - B-level
  - Work-stealing
- Implementation details matter a lot
- Open source scheduler simulator ESTEE
  -  `github.com/it4innovations/estee`
  - Benchmark datasets and results @ Zenodo



2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)

Runtime vs Scheduler: Analyzing Dask's Overheads

Stanislav Böhm
IT4Innovations, VSB – Technical University of Ostrava
stanislav.bohm@vsb.cz

Jakub Beránek
IT4Innovations, VSB – Technical University of Ostrava
jakub.beranek@vsb.cz

***Abstract*—Dask is a distributed task framework which is commonly used by data scientists to parallelize Python code on computing clusters with little programming effort. It uses a sophisticated work-stealing scheduler which has been hand-tuned to execute task graphs as efficiently as possible. But is scheduler optimization a worthwhile effort for Dask? Our paper shows on many real world task graphs that even a completely random scheduler is surprisingly competitive with its built-in scheduler and that the main bottleneck of Dask lies in its runtime overhead. We develop a drop-in replacement for the Dask central server written in Rust which is backwards compatible with existing Dask programs. Thanks to its efficient runtime, our server implementation is able to scale up to larger clusters than Dask and consistently outperforms it on a variety of task graphs, despite the fact that it uses a simpler scheduling algorithm.**

***Index Terms*—distributed task scheduling, dask, workflow, rust**

I. Introduction

Distributed task frameworks are commonly used to scale programs to multiple nodes using little programming effort. While traditional HPC distributed paradigms like MPI promote fixed-size clusters with little resilience and low-level communication APIs designed for maximum performance, modern task frameworks like DASK [1], Ray [2] or Spark [3] support elastic clusters and provide high-level programming interfaces which abstract the communication aspect of the program away. They offer quick prototyping, even though they do not always provide the highest possible performance out-of-the-box. These frameworks are especially popular for distributing machine learning and data analysis programs with a few lines of code.

Even though each task framework offers its own set of APIs that enable writing distributed programs, they all eventually convert the input program into a *task graph*. Vertices of this graph (called tasks) represent functions which operate on input data and generate output data. Arcs represent dependencies and data transfers between the tasks. This program representation is amenable to parallelization in a distributed environment. The job of a task framework is to decide on which computing nodes should the individual tasks be computed and manage data transfers between the nodes.

A crucial component of each task framework is the scheduler, which assigns tasks to nodes in order to minimize the total computation time. Finding the optimal task schedule is a well-known problem which is NP-hard even for very restricted formulations (e.g. even without network data transfers) [4]. A number of heuristics have been proposed to tackle this problem, ranging from list-based scheduling to genetic algorithms [5]–[11]. Many surveys and comparisons of these approaches were published in [12]–[15]. Yet applying these algorithms to modern task frameworks is challenging, as they often make assumptions that do not hold in practice. For example they often assume that task durations are known in advance or that there is no network congestion. Task frameworks thus usually resolve to implementing their own scheduler with many heuristic decisions specific to their common use cases.

One example of such a framework is DASK, a popular Python library used to distribute and parallelize SQL-like table operations, machine learning workflows or even arbitrary Python functions with little programming effort. DASK uses a centralized architecture with a single central component – the server – which schedules tasks and manages computing nodes in a cluster. It uses a work-stealing scheduler which has been tuned extensively to support various task graphs. Yet it is unclear whether additional effort should be directed into improving the scheduler or if there is another bottleneck which should be prioritized.

We have designed a series of experiments that study the runtime overhead of DASK and the effect of the used scheduling algorithm on its performance. We also develop a backwards compatible replacement of the DASK server which aims to minimize its runtime overhead and thus scale to larger clusters. Our paper makes the following *contributions*:

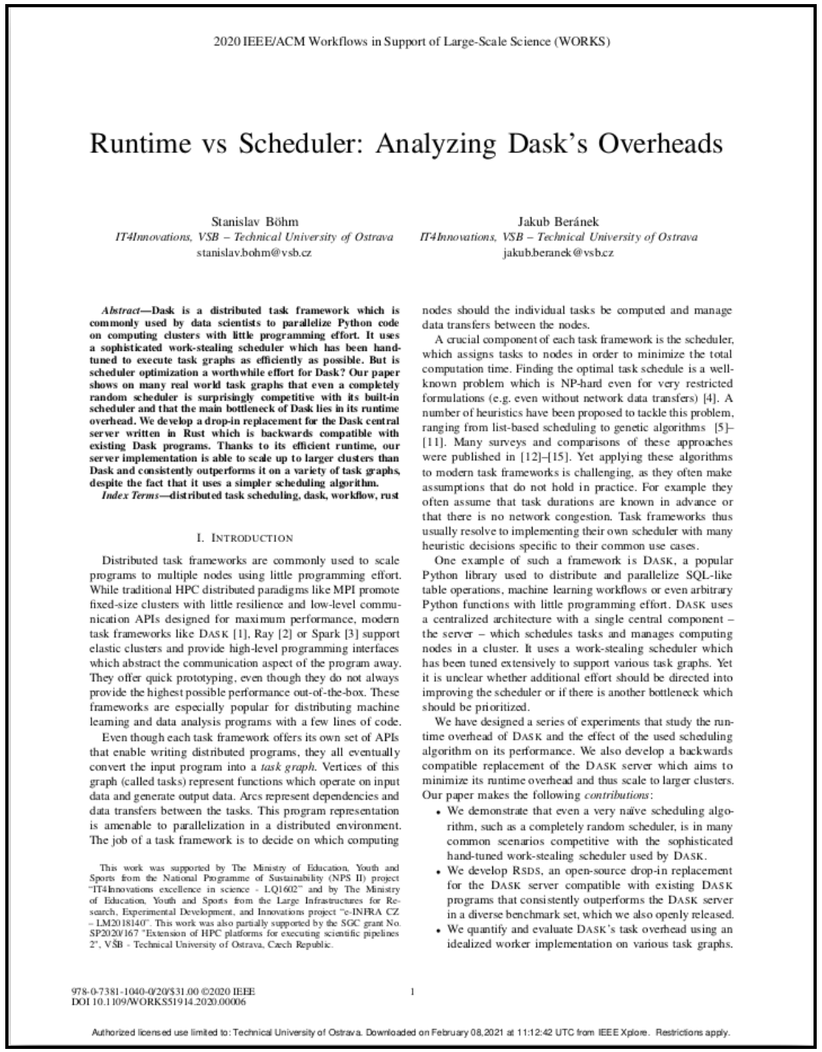
- We demonstrate that even a very naïve scheduling algorithm, such as a completely random scheduler, is in many common scenarios competitive with the sophisticated hand-tuned work-stealing scheduler used by DASK.
- We develop RSDS, an open-source drop-in replacement for the DASK server compatible with existing DASK programs that consistently outperforms the DASK server in a diverse benchmark set, which we also openly released.
- We quantify and evaluate DASK's task overhead using an idealized worker implementation on various task graphs.

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPS II) project "IT4Innovations excellence in science - LQ1602" and by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development, and Innovations project "e-INFRA CZ – LM2018140". This work was also partially supported by the SGC grant No. SP2020/167 "Extension of HPC platforms for executing scientific pipelines 2", VŠB - Technical University of Ostrava, Czech Republic.

978-0-7381-1040-0/20/$31.00 ©2020 IEEE
DOI 10.1109/WORKS51914.2020.00006

Runtime vs Scheduler: Analyzing Dask's Overheads

Ada Böhm, Jakub Beránek
(IEEE/ACM Workflows in Support of Large-Scale Science 2020)



2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)

Runtime vs Scheduler: Analyzing Dask's Overheads

Stanislav Böhm
IT4Innovations, VSB – Technical University of Ostrava
stanislav.bohm@vsb.cz

Jakub Beránek
IT4Innovations, VSB – Technical University of Ostrava
jakub.beranek@vsb.cz

***Abstract*—Dask is a distributed task framework which is commonly used by data scientists to parallelize Python code on computing clusters with little programming effort. It uses a sophisticated work-stealing scheduler which has been hand-tuned to execute task graphs as efficiently as possible. But is scheduler optimization a worthwhile effort for Dask? Our paper shows on many real world task graphs that even a completely random scheduler is surprisingly competitive with its built-in scheduler and that the main bottleneck of Dask lies in its runtime overhead. We develop a drop-in replacement for the Dask central server written in Rust which is backwards compatible with existing Dask programs. Thanks to its efficient runtime, our server implementation is able to scale up to larger clusters than Dask and consistently outperforms it on a variety of task graphs, despite the fact that it uses a simpler scheduling algorithm.**

***Index Terms*—distributed task scheduling, dask, workflow, rust**

I. Introduction

Distributed task frameworks are commonly used to scale programs to multiple nodes using little programming effort. While traditional HPC distributed paradigms like MPI promote fixed-size clusters with little resilience and low-level communication APIs designed for maximum performance, modern task frameworks like DASK [1], Ray [2] or Spark [3] support elastic clusters and provide high-level programming interfaces which abstract the communication aspect of the program away. They offer quick prototyping, even though they do not always provide the highest possible performance out-of-the-box. These frameworks are especially popular for distributing machine learning and data analysis programs with a few lines of code.

Even though each task framework offers its own set of APIs that enable writing distributed programs, they all eventually convert the input program into a *task graph*. Vertices of this graph (called tasks) represent functions which operate on input data and generate output data. Arcs represent dependencies and data transfers between the tasks. This program representation is amenable to parallelization in a distributed environment. The job of a task framework is to decide on which computing nodes should the individual tasks be computed and manage data transfers between the nodes.

A crucial component of each task framework is the scheduler, which assigns tasks to nodes in order to minimize the total computation time. Finding the optimal task schedule is a well-known problem which is NP-hard even for very restricted formulations (e.g. even without network data transfers) [4]. A number of heuristics have been proposed to tackle this problem, ranging from list-based scheduling to genetic algorithms [5]–[11]. Many surveys and comparisons of these approaches were published in [12]–[15]. Yet applying these algorithms to modern task frameworks is challenging, as they often make assumptions that do not hold in practice. For example they often assume that task durations are known in advance or that there is no network congestion. Task frameworks thus usually resolve to implementing their own scheduler with many heuristic decisions specific to their common use cases.

One example of such a framework is DASK, a popular Python library used to distribute and parallelize SQL-like table operations, machine learning workflows or even arbitrary Python functions with little programming effort. DASK uses a centralized architecture with a single central component – the server – which schedules tasks and manages computing nodes in a cluster. It uses a work-stealing scheduler which has been tuned extensively to support various task graphs. Yet it is unclear whether additional effort should be directed into improving the scheduler or if there is another bottleneck which should be prioritized.

We have designed a series of experiments that study the runtime overhead of DASK and the effect of the used scheduling algorithm on its performance. We also develop a backwards compatible replacement of the DASK server which aims to minimize its runtime overhead and thus scale to larger clusters. Our paper makes the following *contributions*:

- We demonstrate that even a very naïve scheduling algorithm, such as a completely random scheduler, is in many common scenarios competitive with the sophisticated hand-tuned work-stealing scheduler used by DASK.
- We develop RSDS, an open-source drop-in replacement for the DASK server compatible with existing DASK programs that consistently outperforms the DASK server in a diverse benchmark set, which we also openly released.
- We quantify and evaluate DASK's task overhead using an idealized worker implementation on various task graphs.

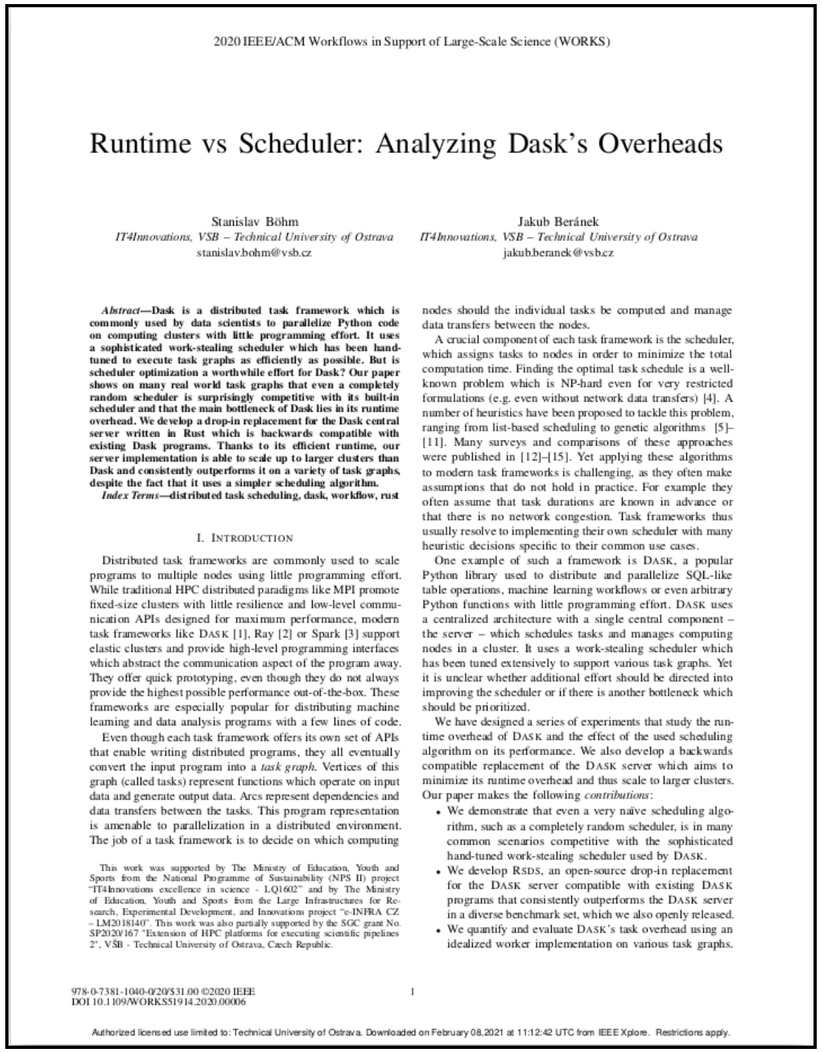
This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPS II) project "IT4Innovations excellence in science - LQ1602" and by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development, and Innovations project "e-INFRA CZ – LM2018140". This work was also partially supported by the SGC grant No. SP2020/167 "Extension of HPC platforms for executing scientific pipelines 2", VŠB - Technical University of Ostrava, Czech Republic.

978-0-7381-1040-0/20/$31.00 ©2020 IEEE
DOI 10.1109/WORKS51914.2020.00006

Runtime vs Scheduler: Analyzing Dask's Overheads

Ada Böhm, Jakub Beránek
(IEEE/ACM Workflows in Support of Large-Scale Science 2020)

- How do schedulers perform "in the wild"?



2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)

Runtime vs Scheduler: Analyzing Dask's Overheads

Stanislav Böhm
IT4Innovations, VSB – Technical University of Ostrava
stanislav.bohm@vsb.cz

Jakub Beránek
IT4Innovations, VSB – Technical University of Ostrava
jakub.beranek@vsb.cz

***Abstract*—Dask is a distributed task framework which is commonly used by data scientists to parallelize Python code on computing clusters with little programming effort. It uses a sophisticated work-stealing scheduler which has been hand-tuned to execute task graphs as efficiently as possible. But is scheduler optimization a worthwhile effort for Dask? Our paper shows on many real world task graphs that even a completely random scheduler is surprisingly competitive with its built-in scheduler and that the main bottleneck of Dask lies in its runtime overhead. We develop a drop-in replacement for the Dask central server written in Rust which is backwards compatible with existing Dask programs. Thanks to its efficient runtime, our server implementation is able to scale up to larger clusters than Dask and consistently outperforms it on a variety of task graphs, despite the fact that it uses a simpler scheduling algorithm.**

***Index Terms*—distributed task scheduling, dask, workflow, rust**

I. Introduction

Distributed task frameworks are commonly used to scale programs to multiple nodes using little programming effort. While traditional HPC distributed paradigms like MPI promote fixed-size clusters with little resilience and low-level communication APIs designed for maximum performance, modern task frameworks like DASK [1], Ray [2] or Spark [3] support elastic clusters and provide high-level programming interfaces which abstract the communication aspect of the program away. They offer quick prototyping, even though they do not always provide the highest possible performance out-of-the-box. These frameworks are especially popular for distributing machine learning and data analysis programs with a few lines of code.

Even though each task framework offers its own set of APIs that enable writing distributed programs, they all eventually convert the input program into a *task graph*. Vertices of this graph (called tasks) represent functions which operate on input data and generate output data. Arcs represent dependencies and data transfers between the tasks. This program representation is amenable to parallelization in a distributed environment. The job of a task framework is to decide on which computing nodes should the individual tasks be computed and manage data transfers between the nodes.

A crucial component of each task framework is the scheduler, which assigns tasks to nodes in order to minimize the total computation time. Finding the optimal task schedule is a well-known problem which is NP-hard even for very restricted formulations (e.g. even without network data transfers) [4]. A number of heuristics have been proposed to tackle this problem, ranging from list-based scheduling to genetic algorithms [5]–[11]. Many surveys and comparisons of these approaches were published in [12]–[15]. Yet applying these algorithms to modern task frameworks is challenging, as they often make assumptions that do not hold in practice. For example they often assume that task durations are known in advance or that there is no network congestion. Task frameworks thus usually resolve to implementing their own scheduler with many heuristic decisions specific to their common use cases.

One example of such a framework is DASK, a popular Python library used to distribute and parallelize SQL-like table operations, machine learning workflows or even arbitrary Python functions with little programming effort. DASK uses a centralized architecture with a single central component – the server – which schedules tasks and manages computing nodes in a cluster. It uses a work-stealing scheduler which has been tuned extensively to support various task graphs. Yet it is unclear whether additional effort should be directed into improving the scheduler or if there is another bottleneck which should be prioritized.

We have designed a series of experiments that study the runtime overhead of DASK and the effect of the used scheduling algorithm on its performance. We also develop a backwards compatible replacement of the DASK server which aims to minimize its runtime overhead and thus scale to larger clusters. Our paper makes the following *contributions*:

- We demonstrate that even a very naïve scheduling algorithm, such as a completely random scheduler, is in many common scenarios competitive with the sophisticated hand-tuned work-stealing scheduler used by DASK.
- We develop RSDS, an open-source drop-in replacement for the DASK server compatible with existing DASK programs that consistently outperforms the DASK server in a diverse benchmark set, which we also openly released.
- We quantify and evaluate DASK's task overhead using an idealized worker implementation on various task graphs.

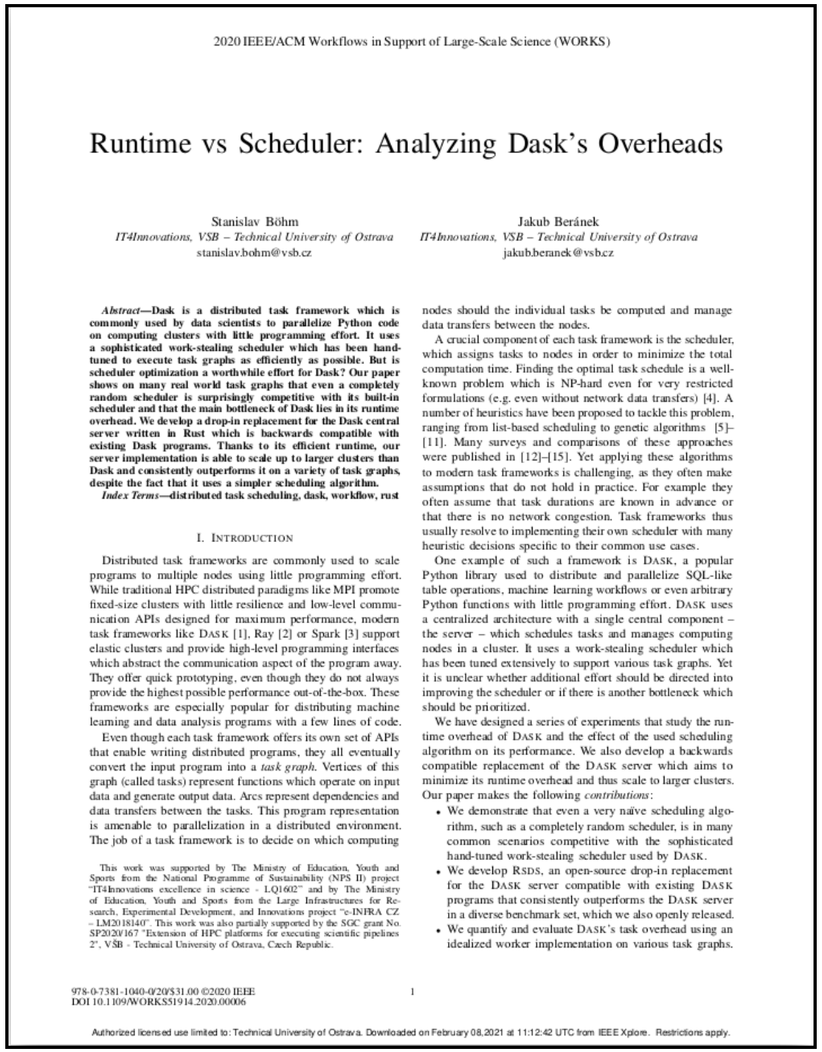
This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPS II) project "IT4Innovations excellence in science - LQ1602" and by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development, and Innovations project "e-INFRA CZ – LM2018140". This work was also partially supported by the SGC grant No. SP2020/167 "Extension of HPC platforms for executing scientific pipelines 2", VŠB - Technical University of Ostrava, Czech Republic.

978-0-7381-1040-0/20/$31.00 ©2020 IEEE
DOI 10.1109/WORKS51914.2020.00006

Runtime vs Scheduler: Analyzing Dask's Overheads

Ada Böhm, Jakub Beránek
(IEEE/ACM Workflows in Support of Large-Scale Science 2020)

- How do schedulers perform "in the wild"?
- What are the perf. characteristics of Dask?



2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)

Runtime vs Scheduler: Analyzing Dask's Overheads

Stanislav Böhm
IT4Innovations, VSB – Technical University of Ostrava
stanislav.bohm@vsb.cz

Jakub Beránek
IT4Innovations, VSB – Technical University of Ostrava
jakub.beranek@vsb.cz

***Abstract*—Dask is a distributed task framework which is commonly used by data scientists to parallelize Python code on computing clusters with little programming effort. It uses a sophisticated work-stealing scheduler which has been hand-tuned to execute task graphs as efficiently as possible. But is scheduler optimization a worthwhile effort for Dask? Our paper shows on many real world task graphs that even a completely random scheduler is surprisingly competitive with its built-in scheduler and that the main bottleneck of Dask lies in its runtime overhead. We develop a drop-in replacement for the Dask central server written in Rust which is backwards compatible with existing Dask programs. Thanks to its efficient runtime, our server implementation is able to scale up to larger clusters than Dask and consistently outperforms it on a variety of task graphs, despite the fact that it uses a simpler scheduling algorithm.**

***Index Terms*—distributed task scheduling, dask, workflow, rust**

I. Introduction

Distributed task frameworks are commonly used to scale programs to multiple nodes using little programming effort. While traditional HPC distributed paradigms like MPI promote fixed-size clusters with little resilience and low-level communication APIs designed for maximum performance, modern task frameworks like DASK [1], Ray [2] or Spark [3] support elastic clusters and provide high-level programming interfaces which abstract the communication aspect of the program away. They offer quick prototyping, even though they do not always provide the highest possible performance out-of-the-box. These frameworks are especially popular for distributing machine learning and data analysis programs with a few lines of code.

Even though each task framework offers its own set of APIs that enable writing distributed programs, they all eventually convert the input program into a *task graph*. Vertices of this graph (called tasks) represent functions which operate on input data and generate output data. Arcs represent dependencies and data transfers between the tasks. This program representation is amenable to parallelization in a distributed environment. The job of a task framework is to decide on which computing nodes should the individual tasks be computed and manage data transfers between the nodes.

A crucial component of each task framework is the scheduler, which assigns tasks to nodes in order to minimize the total computation time. Finding the optimal task schedule is a well-known problem which is NP-hard even for very restricted formulations (e.g. even without network data transfers) [4]. A number of heuristics have been proposed to tackle this problem, ranging from list-based scheduling to genetic algorithms [5]–[11]. Many surveys and comparisons of these approaches were published in [12]–[15]. Yet applying these algorithms to modern task frameworks is challenging, as they often make assumptions that do not hold in practice. For example they often assume that task durations are known in advance or that there is no network congestion. Task frameworks thus usually resolve to implementing their own scheduler with many heuristic decisions specific to their common use cases.

One example of such a framework is DASK, a popular Python library used to distribute and parallelize SQL-like table operations, machine learning workflows or even arbitrary Python functions with little programming effort. DASK uses a centralized architecture with a single central component – the server – which schedules tasks and manages computing nodes in a cluster. It uses a work-stealing scheduler which has been tuned extensively to support various task graphs. Yet it is unclear whether additional effort should be directed into improving the scheduler or if there is another bottleneck which should be prioritized.

We have designed a series of experiments that study the runtime overhead of DASK and the effect of the used scheduling algorithm on its performance. We also develop a backwards compatible replacement of the DASK server which aims to minimize its runtime overhead and thus scale to larger clusters. Our paper makes the following *contributions*:

- We demonstrate that even a very naïve scheduling algorithm, such as a completely random scheduler, is in many common scenarios competitive with the sophisticated hand-tuned work-stealing scheduler used by DASK.
- We develop RSDS, an open-source drop-in replacement for the DASK server compatible with existing DASK programs that consistently outperforms the DASK server in a diverse benchmark set, which we also openly released.
- We quantify and evaluate DASK's task overhead using an idealized worker implementation on various task graphs.

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPS II) project "IT4Innovations excellence in science - LQ1602" and by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development, and Innovations project "e-INFRA CZ – LM2018140". This work was also partially supported by the SGC grant No. SP2020/167 "Extension of HPC platforms for executing scientific pipelines 2", VŠB - Technical University of Ostrava, Czech Republic.

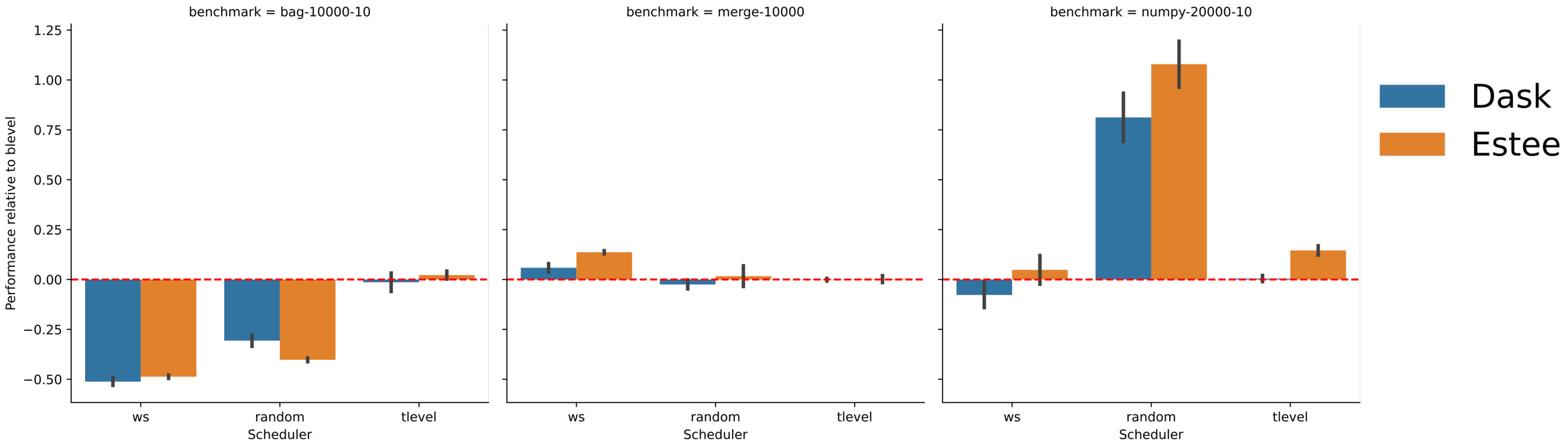
978-0-7381-1040-0/20/$31.00 ©2020 IEEE
DOI 10.1109/WORKS51914.2020.00006

Runtime vs Scheduler: Analyzing Dask's Overheads

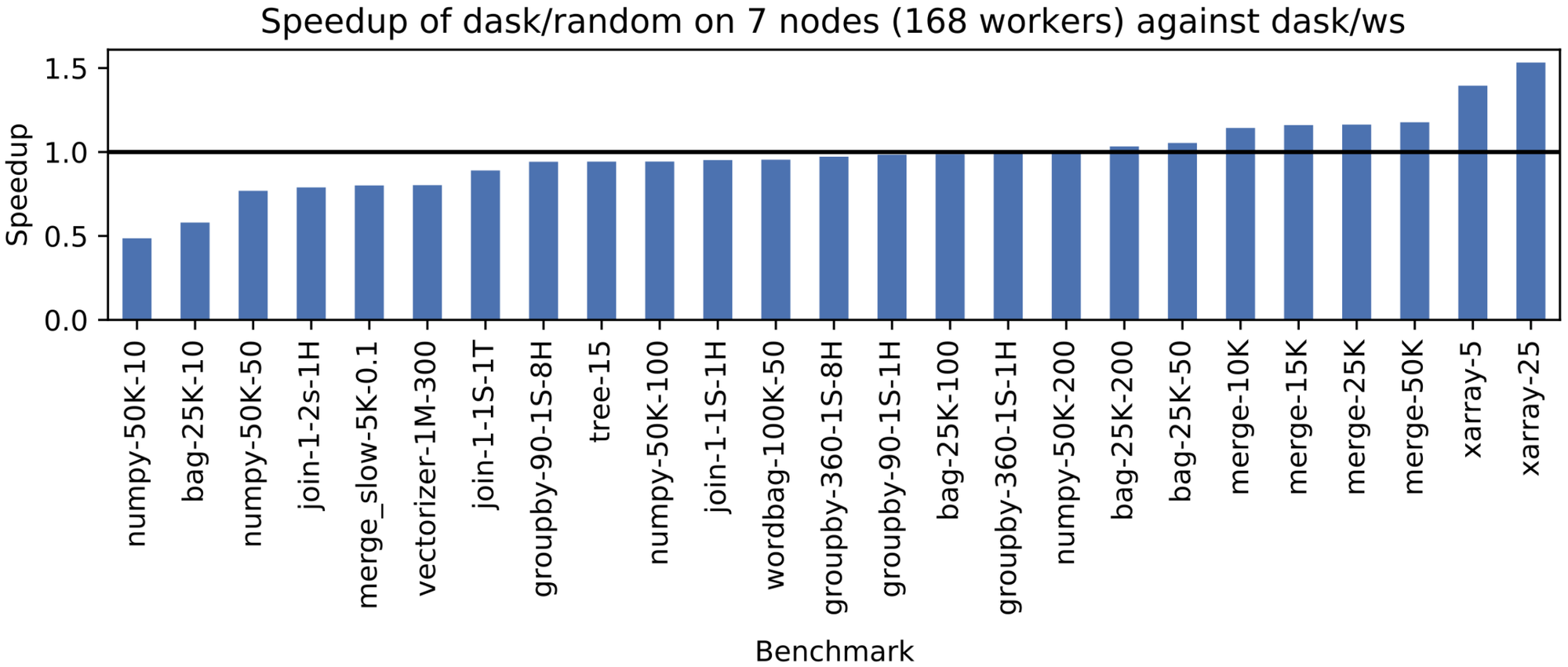
Ada Böhm, Jakub Beránek
(IEEE/ACM Workflows in Support of Large-Scale Science 2020)

- How do schedulers perform "in the wild"?
- What are the perf. characteristics of Dask?
- Can we make Dask more efficient on HPC?

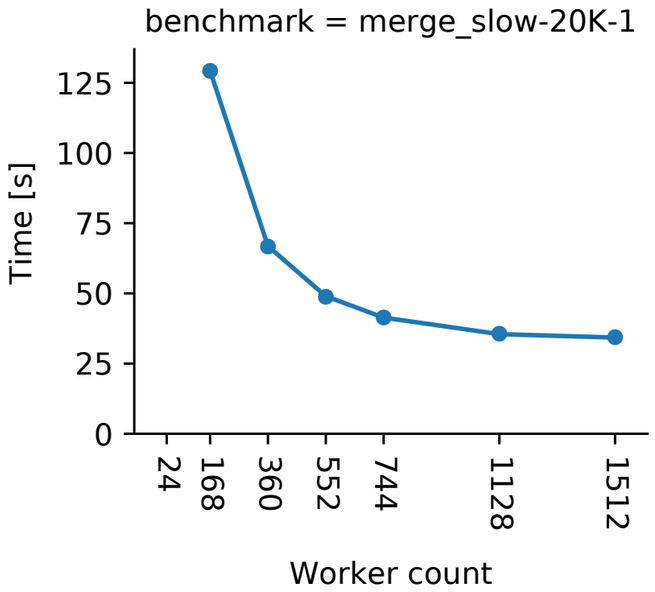
Validation of ESTEE results

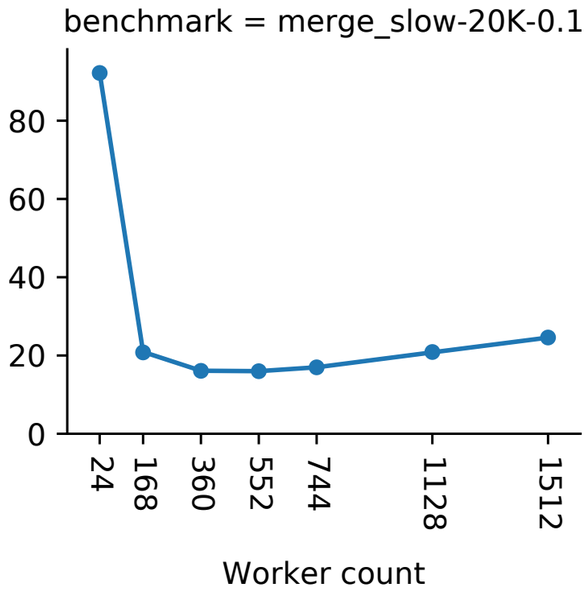


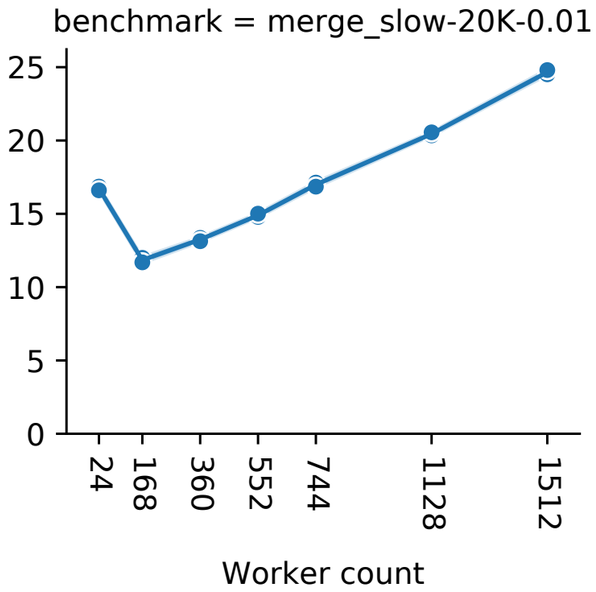
Random vs. work-stealing scheduler



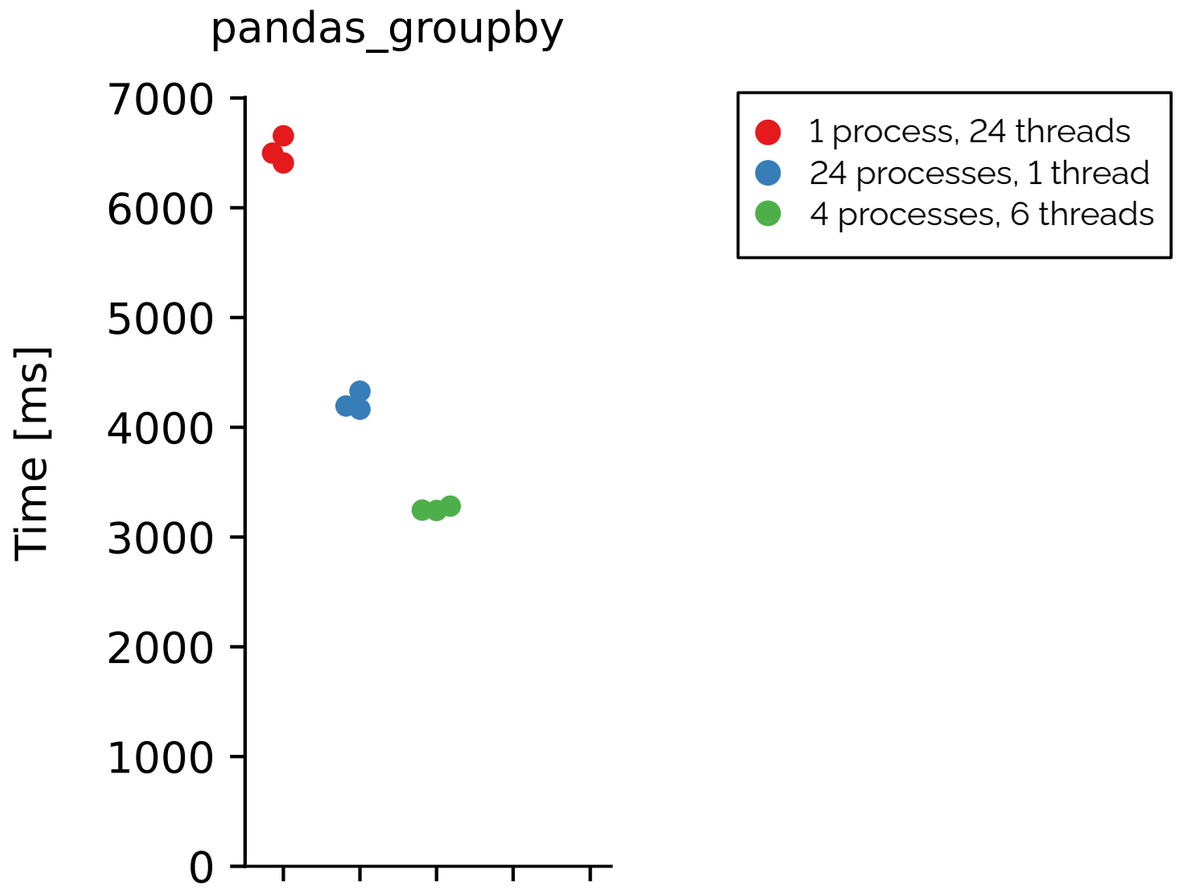
Dask scaling







Effect of GIL on Dask's performance



Observations

- Scheduler cannot be easily replaced

Observations

- Scheduler cannot be easily replaced
- Dask does not scale well in HPC contexts

Observations

- Scheduler cannot be easily replaced
- Dask does not scale well in HPC contexts
- Python implementation is limited by the GIL

RSDS

- Alternative Dask server implementation

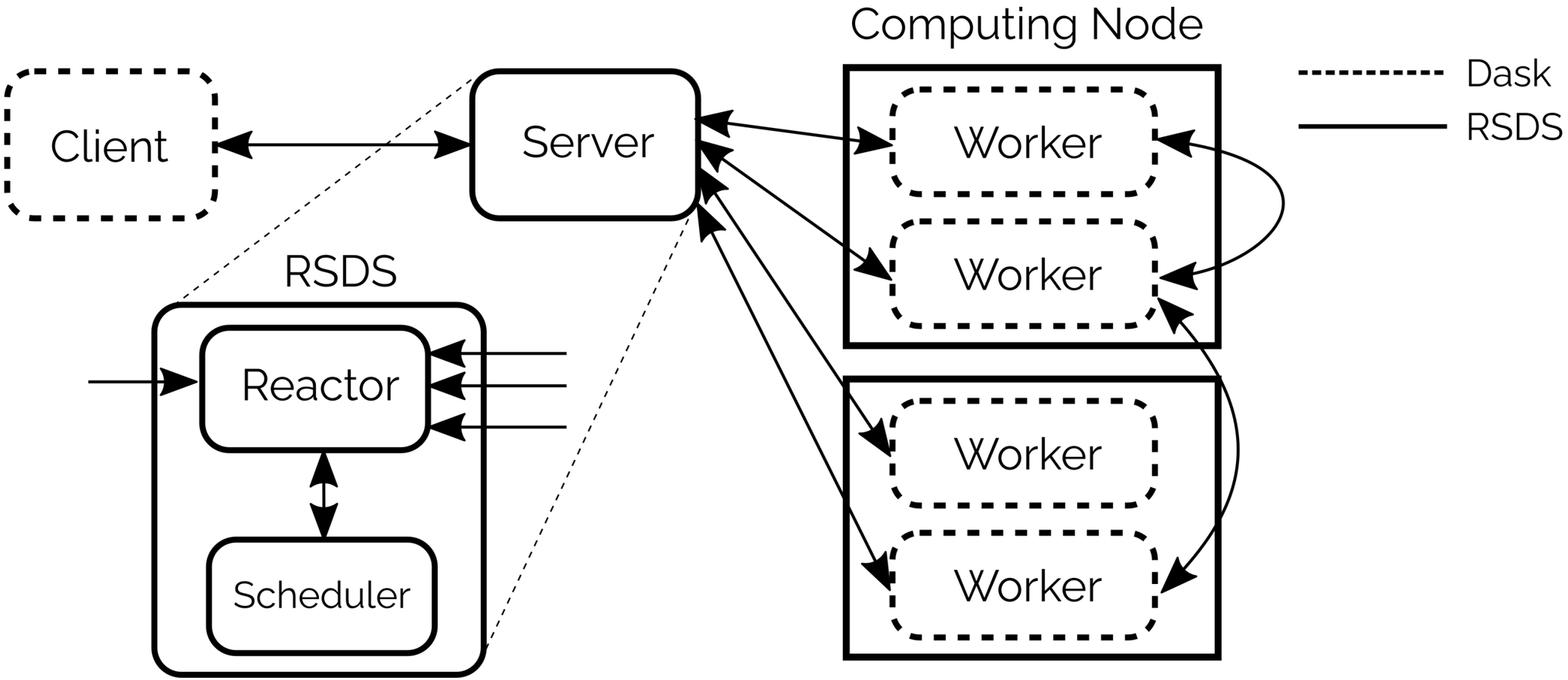
RSDS

- Alternative Dask server implementation
- Written in Rust, designed for low overhead

RSDS

- Alternative Dask server implementation
- Written in Rust, designed for low overhead
- Explicitly designed for a pluggable scheduler

RSDS architecture



Dask vs. RSDS: work-stealing scheduler

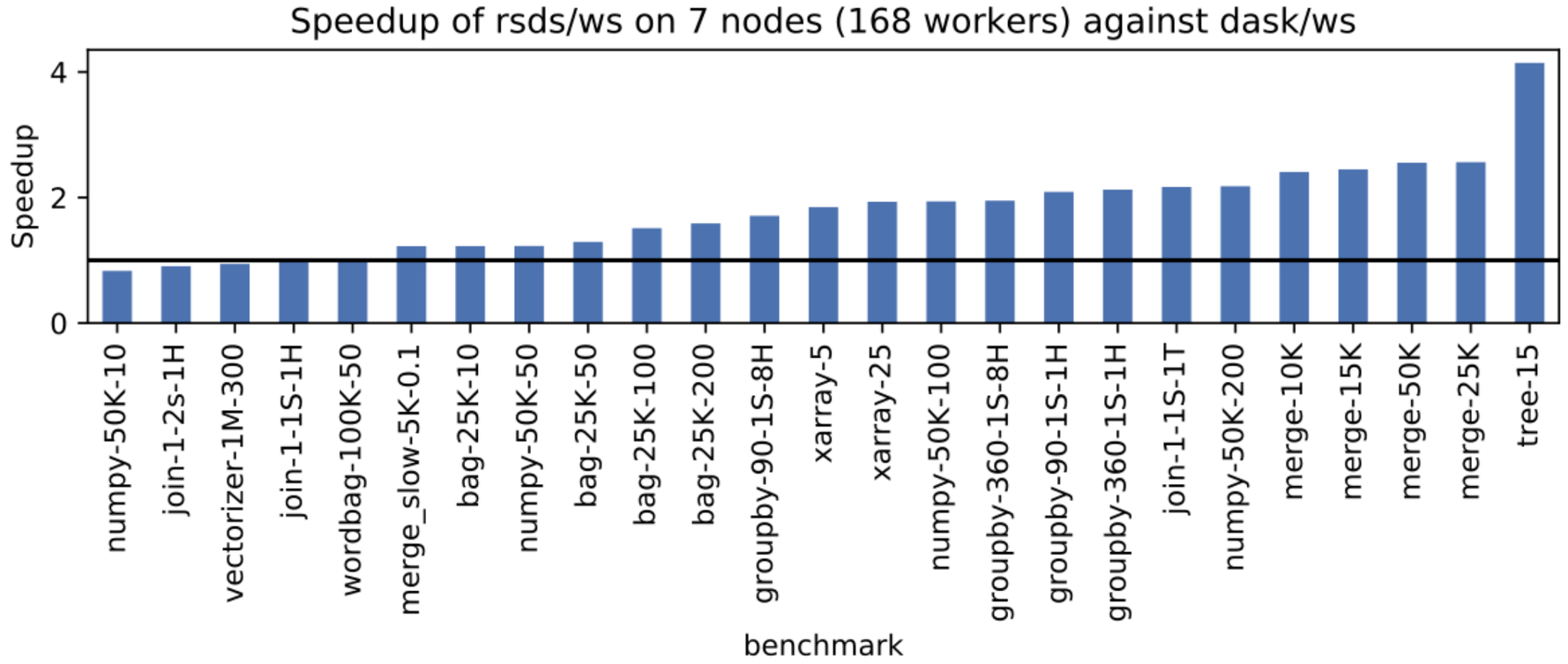
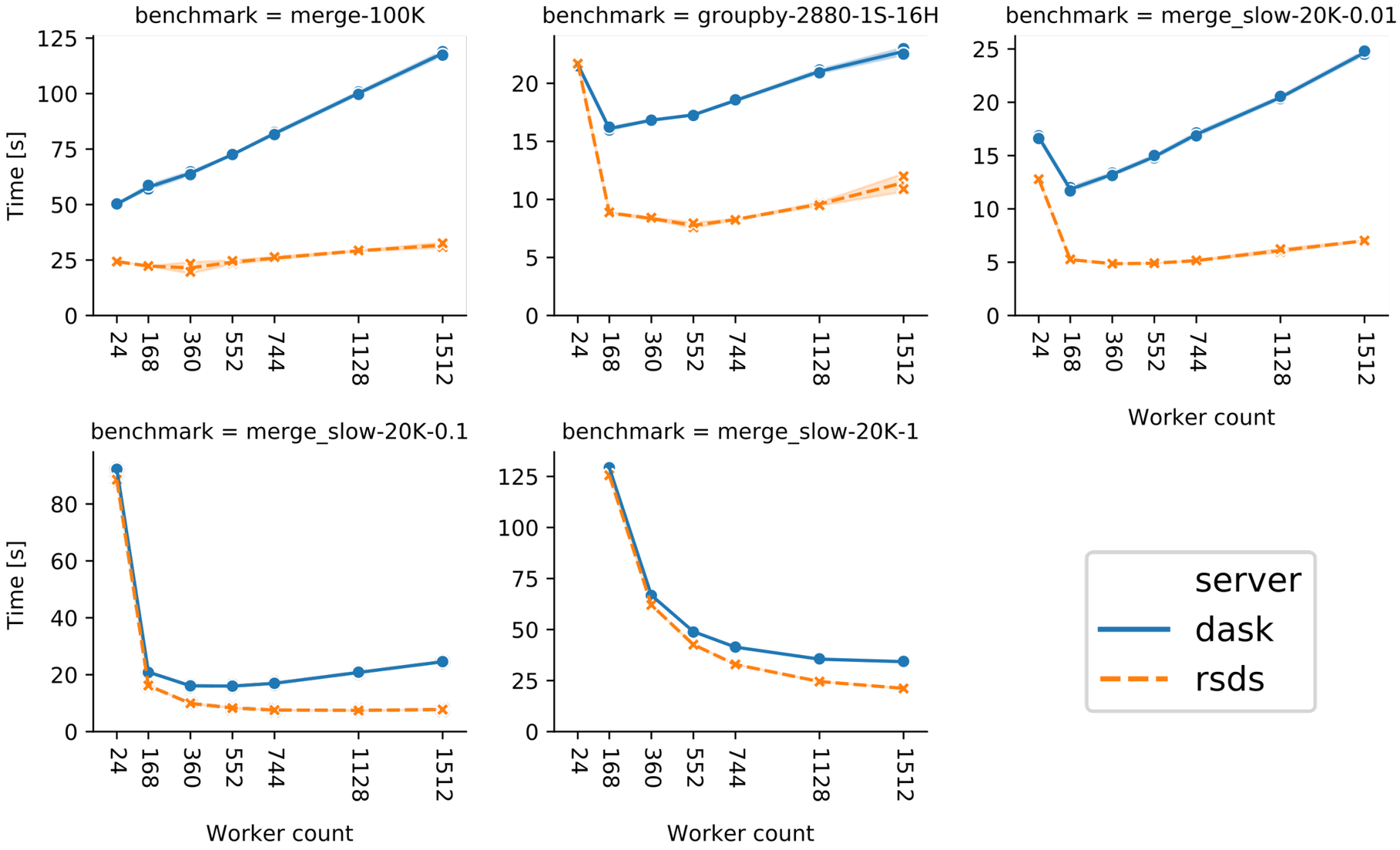


Fig. 3. Speedup of RSDS/ws scheduler; DASK/ws is baseline.

Dask vs. RSDS: scaling



Outcome

- Runtime efficiency as important as scheduling
  - Even a random scheduler can be competitive!

Outcome

- Runtime efficiency as important as scheduling
  - Even a random scheduler can be competitive!
- Dask scaled poorly on HPC
  - Caused by inefficient Python runtime and GIL

Outcome

- Runtime efficiency as important as scheduling
  - Even a random scheduler can be competitive!
- Dask scaled poorly on HPC
  - Caused by inefficient Python runtime and GIL
- RSDS: open source alternative to Dask's server
  -  `github.com/it4innovations/rsds`

Outcome

- Runtime efficiency as important as scheduling
  - Even a random scheduler can be competitive!
- Dask scaled poorly on HPC
  - Caused by inefficient Python runtime and GIL
- RSDS: open source alternative to Dask's server
  - `github.com/it4innovations/rsds`
  - Backward-compatible with existing Dask programs

Outcome

- Runtime efficiency as important as scheduling
  - Even a random scheduler can be competitive!
- Dask scaled poorly on HPC
  - Caused by inefficient Python runtime and GIL
- RSDS: open source alternative to Dask's server
  -  `github.com/it4innovations/rsds`
  - Backward-compatible with existing Dask programs
- Some ideas from RSDS were integrated into Dask

Ergonomics and efficiency
of workflows on HPC clusters

SoftwareX 27 (2024) 101814

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek [a,*], Ada Böhm [a], Gianluca Palermo [b], Jan Martinovič [a], Branislav Jansík [a]

[a] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic
[b] Politecnico di Milano - DEIB, Milano, Italy

ARTICLE INFO

Dataset link: https://github.com/it4innovations/hyperqueue

Keywords:
Distributed computing
Task scheduling
High performance computing
Job manager

ABSTRACT

Task graphs are a popular method for defining complex scientific simulations and experiments that run on distributed and HPC (High-performance computing) clusters, because they allow their authors to focus on the problem domain, instead of low-level communication between nodes, and also enable quick prototyping. However, executing task graphs on HPC clusters can be problematic in the presence of allocation managers like PBS or Slurm, which are not designed for executing a large number of potentially short-lived tasks with dependencies. To make task graph execution on HPC clusters more efficient and ergonomic, we have created HyperQueue, an open-source task graph execution runtime tailored for HPC use-cases. It enables the execution of large task graphs on top of an allocation manager by aggregating tasks into a smaller amount of PBS/Slurm allocations and dynamically load balances tasks amongst all available nodes. It can also automatically submit allocations on behalf of the user, it supports arbitrary task resource requirements and heterogeneous HPC clusters, it is trivial to deploy and does not require elevated privileges.

Code metadata

Current code version	v0.19
Permanent link to repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00113
Permanent link to Reproducible Capsule	
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Rust, Python
Compilation requirements, operating environments & dependencies	Runtc 1.76.0+, Python 3.8+, Linux
If available Link to developer documentation/manual	https://it4innovations.github.io/hyperqueue/
Support email for questions	jakub.beranek@vsb.cz

1. Motivation and significance

High-performance Computing (HPC) applications, which are crucial for the advancement of scientific research, have traditionally been implemented using rather low-level technologies for distribution and parallelization of computation, such as MPI [1] or OpenMP [2]. However, as scientific software becomes more complex, it becomes unfeasible for scientists to implement large applications using such low-level technologies directly. Large scientific workflows are typically composed of many individual [3] programs, and the focus shifts on how to combine these parts together and efficiently distribute their computation on multiple machines (nodes).

To achieve that, it is becoming quite popular to develop scientific applications using the task-based programming model [4]. This model describes computational patterns with a task graph, where vertices (tasks) represent atomic computations, and edges represent dependencies and data transfers between tasks. It allows users to easily express complex distributed applications in a declarative manner, without having to deal with network communication strategies, as the available parallelization is extracted out of the graph automatically by a *task runtime* that executes it.

A large variety of task-based tools exists, ranging from very fine-grained approaches with tasks that span only a few instructions [2,5] to coarse-grained task workflows that execute functions or binaries that

* Corresponding author.
E-mail address: jakub.beranek@vsb.cz (Jakub Beránek).

https://doi.org/10.1016/j.softx.2024.101814
Received 20 February 2024; Received in revised form 3 May 2024; Accepted 28 June 2024
Available online 11 July 2024

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek, Ada Böhm, Gianluca Palermo, Jan Martinovič, Branislav Jansík
(SoftwareX 2024)

SoftwareX 27 (2024) 101814

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek [a,*], Ada Böhm [a], Gianluca Palermo [b], Jan Martinovič [a], Branislav Jansík [a]

[a] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic
[b] Politecnico di Milano - DEIB, Milano, Italy

ARTICLE INFO

Dataset link: https://github.com/it4innovations/hyperqueue

Keywords:
Distributed computing
Task scheduling
High performance computing
Job manager

ABSTRACT

Task graphs are a popular method for defining complex scientific simulations and experiments that run on distributed and HPC (High-performance computing) clusters, because they allow their authors to focus on the problem domain, instead of low-level communication between nodes, and also enable quick prototyping. However, executing task graphs on HPC clusters can be problematic in the presence of allocation managers like PBS or Slurm, which are not designed for executing a large number of potentially short-lived tasks with dependencies. To make task graph execution on HPC clusters more efficient and ergonomic, we have created HyperQueue, an open-source task graph execution runtime tailored for HPC use-cases. It enables the execution of large task graphs on top of an allocation manager by aggregating tasks into a smaller amount of PBS/Slurm allocations and dynamically load balances tasks amongst all available nodes. It can also automatically submit allocations on behalf of the user, it supports arbitrary task resource requirements and heterogeneous HPC clusters, it is trivial to deploy and does not require elevated privileges.

Code metadata

Current code version	v0.19
Permanent link to repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00113
Permanent link to Reproducible Capsule	
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Rust, Python
Compilation requirements, operating environments & dependencies	Runtc 1.76.0+, Python 3.8+, Linux
If available Link to developer documentation/manual	https://it4innovations.github.io/hyperqueue/
Support email for questions	jakub.beranek@vsb.cz

1. Motivation and significance

High-performance Computing (HPC) applications, which are crucial for the advancement of scientific research, have traditionally been implemented using rather low-level technologies for distribution and parallelization of computation, such as MPI [1] or OpenMP [2]. However, as scientific software becomes more complex, it becomes unfeasible for scientists to implement large applications using such low-level technologies directly. Large scientific workflows are typically composed of many individual [3] programs, and the focus shifts on how to combine these parts together and efficiently distribute their computation on multiple machines (nodes).

To achieve that, it is becoming quite popular to develop scientific applications using the task-based programming model [4]. This model describes computational patterns with a task graph, where vertices (tasks) represent atomic computations, and edges represent dependencies and data transfers between tasks. It allows users to easily express complex distributed applications in a declarative manner, without having to deal with network communication strategies, as the available parallelization is extracted out of the graph automatically by a *task runtime* that executes it.

A large variety of task-based tools exists, ranging from very fine-grained approaches with tasks that span only a few instructions [2,5] to coarse-grained task workflows that execute functions or binaries that

* Corresponding author.
E-mail address: jakub.beranek@vsb.cz (Jakub Beránek).

https://doi.org/10.1016/j.softx.2024.101814
Received 20 February 2024; Received in revised form 3 May 2024; Accepted 28 June 2024
Available online 11 July 2024

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek, Ada Böhm, Gianluca Palermo, Jan Martinovič, Branislav Jansík
(SoftwareX 2024)

- How to deal with HPC workflow challenges?

SoftwareX 27 (2024) 101814

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek [a,*], Ada Böhm [a], Gianluca Palermo [b], Jan Martinovič [a], Branislav Jansík [a]

[a] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic
[b] Politecnico di Milano - DEIB, Milano, Italy

ARTICLE INFO

Dataset link: https://github.com/it4innovations/hyperqueue

Keywords:
Distributed computing
Task scheduling
High performance computing
Job manager

ABSTRACT

Task graphs are a popular method for defining complex scientific simulations and experiments that run on distributed and HPC (High-performance computing) clusters, because they allow their authors to focus on the problem domain, instead of low-level communication between nodes, and also enable quick prototyping. However, executing task graphs on HPC clusters can be problematic in the presence of allocation managers like PBS or Slurm, which are not designed for executing a large number of potentially short-lived tasks with dependencies. To make task graph execution on HPC clusters more efficient and ergonomic, we have created HyperQueue, an open-source task graph execution runtime tailored for HPC use-cases. It enables the execution of large task graphs on top of an allocation manager by aggregating tasks into a smaller amount of PBS/Slurm allocations and dynamically load balances tasks amongst all available nodes. It can also automatically submit allocations on behalf of the user, it supports arbitrary task resource requirements and heterogeneous HPC clusters, it is trivial to deploy and does not require elevated privileges.

Code metadata

Current code version	v0.19
Permanent link to repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00113
Permanent link to Reproducible Capsule	
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Rust, Python
Compilation requirements, operating environments & dependencies	Runtc 1.76.0+, Python 3.8+, Linux
If available Link to developer documentation/manual	https://it4innovations.github.io/hyperqueue/
Support email for questions	jakub.beranek@vsb.cz

1. Motivation and significance

High-performance Computing (HPC) applications, which are crucial for the advancement of scientific research, have traditionally been implemented using rather low-level technologies for distribution and parallelization of computation, such as MPI [1] or OpenMP [2]. However, as scientific software becomes more complex, it becomes unfeasible for scientists to implement large applications using such low-level technologies directly. Large scientific workflows are typically composed of many individual [3] programs, and the focus shifts on how to combine these parts together and efficiently distribute their computation on multiple machines (nodes).

To achieve that, it is becoming quite popular to develop scientific applications using the task-based programming model [4]. This model describes computational patterns with a task graph, where vertices (tasks) represent atomic computations, and edges represent dependencies and data transfers between tasks. It allows users to easily express complex distributed applications in a declarative manner, without having to deal with network communication strategies, as the available parallelization is extracted out of the graph automatically by a *task runtime* that executes it.

A large variety of task-based tools exists, ranging from very fine-grained approaches with tasks that span only a few instructions [2,5] to coarse-grained task workflows that execute functions or binaries that

* Corresponding author.
E-mail address: jakub.beranek@vsb.cz (Jakub Beránek).

https://doi.org/10.1016/j.softx.2024.101814
Received 20 February 2024; Received in revised form 3 May 2024; Accepted 28 June 2024
Available online 11 July 2024

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek, Ada Böhm, Gianluca Palermo, Jan Martinovič, Branislav Jansík
(SoftwareX 2024)

- How to deal with HPC workflow challenges?
  - Interaction with HPC allocation managers

SoftwareX 27 (2024) 101814

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek [a,*], Ada Böhm [a], Gianluca Palermo [b], Jan Martinovič [a], Branislav Jansík [a]

[a] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic
[b] Politecnico di Milano - DEIB, Milano, Italy

ARTICLE INFO

Dataset link: https://github.com/it4innovations/hyperqueue

Keywords:
Distributed computing
Task scheduling
High performance computing
Job manager

ABSTRACT

Task graphs are a popular method for defining complex scientific simulations and experiments that run on distributed and HPC (High-performance computing) clusters, because they allow their authors to focus on the problem domain, instead of low-level communication between nodes, and also enable quick prototyping. However, executing task graphs on HPC clusters can be problematic in the presence of allocation managers like PBS or Slurm, which are not designed for executing a large number of potentially short-lived tasks with dependencies. To make task graph execution on HPC clusters more efficient and ergonomic, we have created HyperQueue, an open-source task graph execution runtime tailored for HPC use-cases. It enables the execution of large task graphs on top of an allocation manager by aggregating tasks into a smaller amount of PBS/Slurm allocations and dynamically load balances tasks amongst all available nodes. It can also automatically submit allocations on behalf of the user, it supports arbitrary task resource requirements and heterogeneous HPC clusters, it is trivial to deploy and does not require elevated privileges.

Code metadata

Current code version	v0.19
Permanent link to repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00113
Permanent link to Reproducible Capsule	
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Rust, Python
Compilation requirements, operating environments & dependencies	Runtc 1.76.0+, Python 3.8+, Linux
If available Link to developer documentation/manual	https://it4innovations.github.io/hyperqueue/
Support email for questions	jakub.beranek@vsb.cz

1. Motivation and significance

High-performance Computing (HPC) applications, which are crucial for the advancement of scientific research, have traditionally been implemented using rather low-level technologies for distribution and parallelization of computation, such as MPI [1] or OpenMP [2]. However, as scientific software becomes more complex, it becomes unfeasible for scientists to implement large applications using such low-level technologies directly. Large scientific workflows are typically composed of many individual [3] programs, and the focus shifts on how to combine these parts together and efficiently distribute their computation on multiple machines (nodes).

To achieve that, it is becoming quite popular to develop scientific applications using the task-based programming model [4]. This model describes computational patterns with a task graph, where vertices (tasks) represent atomic computations, and edges represent dependencies and data transfers between tasks. It allows users to easily express complex distributed applications in a declarative manner, without having to deal with network communication strategies, as the available parallelization is extracted out of the graph automatically by a *task runtime* that executes it.

A large variety of task-based tools exists, ranging from very fine-grained approaches with tasks that span only a few instructions [2,5] to coarse-grained task workflows that execute functions or binaries that

* Corresponding author.
E-mail address: jakub.beranek@vsb.cz (Jakub Beránek).

https://doi.org/10.1016/j.softx.2024.101814
Received 20 February 2024; Received in revised form 3 May 2024; Accepted 28 June 2024
Available online 11 July 2024

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek, Ada Böhm, Gianluca Palermo, Jan Martinovič, Branislav Jansík
(SoftwareX 2024)

- How to deal with HPC workflow challenges?
  - Interaction with HPC allocation managers
  - Heterogeneous resource management

SoftwareX 27 (2024) 101814

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek [a,*], Ada Böhm [a], Gianluca Palermo [b], Jan Martinovič [a], Branislav Jansík [a]

[a] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic
[b] Politecnico di Milano - DEIB, Milano, Italy

ARTICLE INFO

Dataset link: https://github.com/it4innovations/hyperqueue

Keywords:
Distributed computing
Task scheduling
High performance computing
Job manager

ABSTRACT

Task graphs are a popular method for defining complex scientific simulations and experiments that run on distributed and HPC (High-performance computing) clusters, because they allow their authors to focus on the problem domain, instead of low-level communication between nodes, and also enable quick prototyping. However, executing task graphs on HPC clusters can be problematic in the presence of allocation managers like PBS or Slurm, which are not designed for executing a large number of potentially short-lived tasks with dependencies. To make task graph execution on HPC clusters more efficient and ergonomic, we have created HyperQueue, an open-source task graph execution runtime tailored for HPC use-cases. It enables the execution of large task graphs on top of an allocation manager by aggregating tasks into a smaller amount of PBS/Slurm allocations and dynamically load balances tasks amongst all available nodes. It can also automatically submit allocations on behalf of the user, it supports arbitrary task resource requirements and heterogeneous HPC clusters, it is trivial to deploy and does not require elevated privileges.

Code metadata

Current code version	v0.19
Permanent link to repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00113
Permanent link to Reproducible Capsule	
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Rust, Python
Compilation requirements, operating environments & dependencies	Runtc 1.76.0+, Python 3.8+, Linux
If available Link to developer documentation/manual	https://it4innovations.github.io/hyperqueue/
Support email for questions	jakub.beranek@vsb.cz

1. Motivation and significance

High-performance Computing (HPC) applications, which are crucial for the advancement of scientific research, have traditionally been implemented using rather low-level technologies for distribution and parallelization of computation, such as MPI [1] or OpenMP [2]. However, as scientific software becomes more complex, it becomes unfeasible for scientists to implement large applications using such low-level technologies directly. Large scientific workflows are typically composed of many individual [3] programs, and the focus shifts on how to combine these parts together and efficiently distribute their computation on multiple machines (nodes).

To achieve that, it is becoming quite popular to develop scientific applications using the task-based programming model [4]. This model describes computational patterns with a task graph, where vertices (tasks) represent atomic computations, and edges represent dependencies and data transfers between tasks. It allows users to easily express complex distributed applications in a declarative manner, without having to deal with network communication strategies, as the available parallelization is extracted out of the graph automatically by a *task runtime* that executes it.

A large variety of task-based tools exists, ranging from very fine-grained approaches with tasks that span only a few instructions [2,5] to coarse-grained task workflows that execute functions or binaries that

* Corresponding author.
E-mail address: jakub.beranek@vsb.cz (Jakub Beránek).

https://doi.org/10.1016/j.softx.2024.101814
Received 20 February 2024; Received in revised form 3 May 2024; Accepted 28 June 2024
Available online 11 July 2024

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek, Ada Böhm, Gianluca Palermo, Jan Martinovič, Branislav Jansík
(SoftwareX 2024)

- How to deal with HPC workflow challenges?
  - Interaction with HPC allocation managers
  - Heterogeneous resource management
  - Scalability

SoftwareX 27 (2024) 101814

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

Jakub Beránek [a,*], Ada Böhm [a], Gianluca Palermo [b], Jan Martinovič [a], Branislav Jansík [a]

[a] IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic
[b] Politecnico di Milano - DEIB, Milano, Italy

ARTICLE INFO

Dataset link: https://github.com/it4innovations/hyperqueue

Keywords:
Distributed computing
Task scheduling
High performance computing
Job manager

ABSTRACT

Task graphs are a popular method for defining complex scientific simulations and experiments that run on distributed and HPC (High-performance computing) clusters, because they allow their authors to focus on the problem domain, instead of low-level communication between nodes, and also enable quick prototyping. However, executing task graphs on HPC clusters can be problematic in the presence of allocation managers like PBS or Slurm, which are not designed for executing a large number of potentially short-lived tasks with dependencies. To make task graph execution on HPC clusters more efficient and ergonomic, we have created HyperQueue, an open-source task graph execution runtime tailored for HPC use-cases. It enables the execution of large task graphs on top of an allocation manager by aggregating tasks into a smaller amount of PBS/Slurm allocations and dynamically load balances tasks amongst all available nodes. It can also automatically submit allocations on behalf of the user, it supports arbitrary task resource requirements and heterogeneous HPC clusters, it is trivial to deploy and does not require elevated privileges.

Code metadata

Current code version	v0.19
Permanent link to repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00113
Permanent link to Reproducible Capsule	
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Rust, Python
Compilation requirements, operating environments & dependencies	Runtc 1.76.0+, Python 3.8+, Linux
If available Link to developer documentation/manual	https://it4innovations.github.io/hyperqueue/
Support email for questions	jakub.beranek@vsb.cz

1. Motivation and significance

High-performance Computing (HPC) applications, which are crucial for the advancement of scientific research, have traditionally been implemented using rather low-level technologies for distribution and parallelization of computation, such as MPI [1] or OpenMP [2]. However, as scientific software becomes more complex, it becomes unfeasible for scientists to implement large applications using such low-level technologies directly. Large scientific workflows are typically composed of many individual [3] programs, and the focus shifts on how to combine these parts together and efficiently distribute their computation on multiple machines (nodes).

To achieve that, it is becoming quite popular to develop scientific applications using the task-based programming model [4]. This model describes computational patterns with a task graph, where vertices (tasks) represent atomic computations, and edges represent dependencies and data transfers between tasks. It allows users to easily express complex distributed applications in a declarative manner, without having to deal with network communication strategies, as the available parallelization is extracted out of the graph automatically by a *task runtime* that executes it.

A large variety of task-based tools exists, ranging from very fine-grained approaches with tasks that span only a few instructions [2,5] to coarse-grained task workflows that execute functions or binaries that

* Corresponding author.
E-mail address: jakub.beranek@vsb.cz (Jakub Beránek).

https://doi.org/10.1016/j.softx.2024.101814
Received 20 February 2024; Received in revised form 3 May 2024; Accepted 28 June 2024
Available online 11 July 2024

HyperQueue: Efficient and ergonomic task graphs on HPC clusters

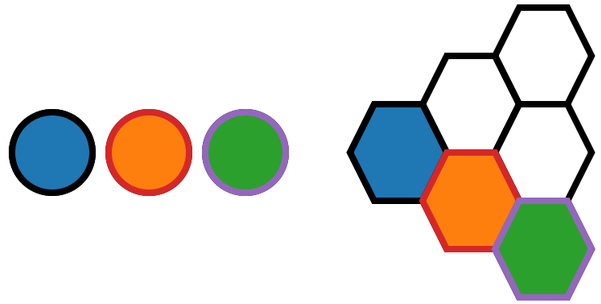
Jakub Beránek, Ada Böhm, Gianluca Palermo, Jan Martinovič, Branislav Jansík
(SoftwareX 2024)

- How to deal with HPC workflow challenges?
  - Interaction with HPC allocation managers
  - Heterogeneous resource management
  - Scalability
- Can we integrate the solutions into a single task runtime?

Challenge: how to map tasks to allocations?

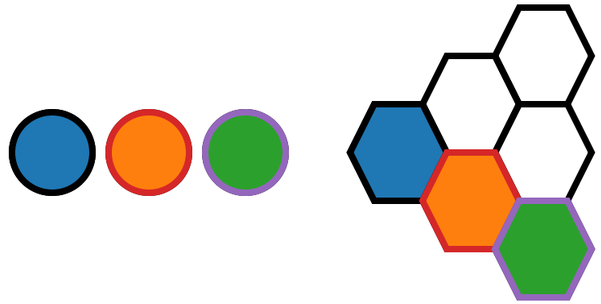
Challenge: how to map tasks to allocations?

- Each task is a separate allocation
  - Massive overhead (millions of allocations)
  - Allocation count limits
  - Node granularity

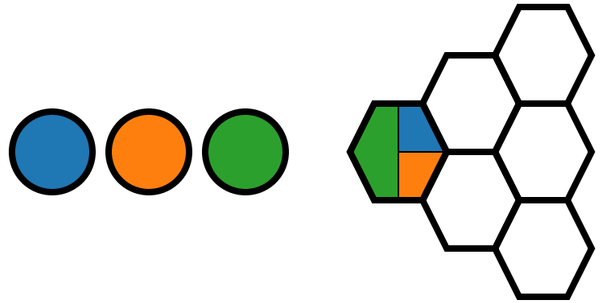


Challenge: how to map tasks to allocations?

- Each task is a separate allocation
  - Massive overhead (millions of allocations)
  - Allocation count limits
  - Node granularity

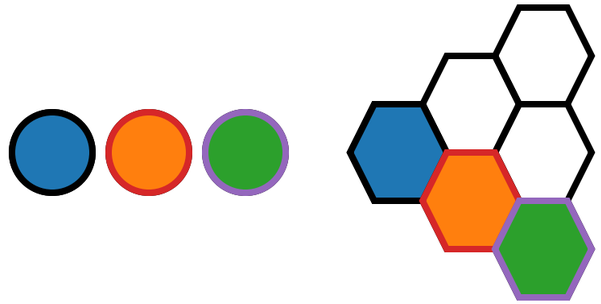


- One allocation for the whole workflow
  - Only for small workflows
  - Leads to resource waste

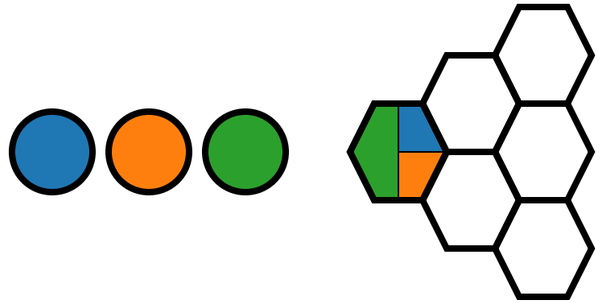


Challenge: how to map tasks to allocations?

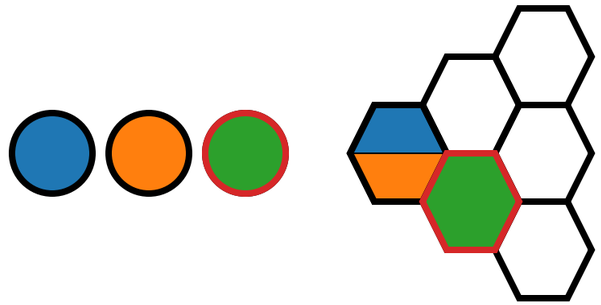
- Each task is a separate allocation
  - Massive overhead (millions of allocations)
  - Allocation count limits
  - Node granularity



- One allocation for the whole workflow
  - Only for small workflows
  - Leads to resource waste



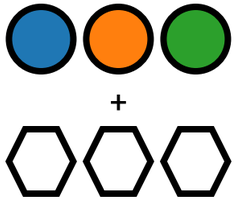
- Split workflow into multiple allocations
  - Challenging to find good partitioning
  - No load balancing across allocations
  - Dependencies and fault tolerance are problematic



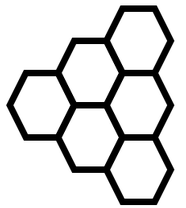
Workflow partitioning

Workflow partitioning

Manual (Slurm)

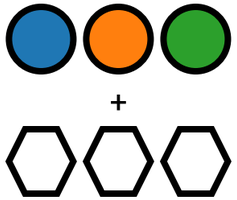


→ Slurm →



Workflow partitioning

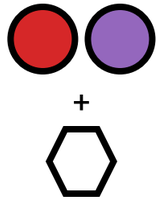
Manual (Slurm)



→ Slurm →

Workflow partitioning

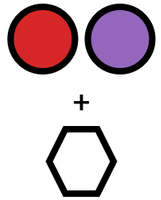
Manual (Slurm)



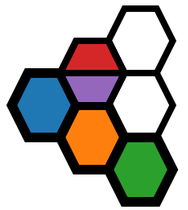
→ Slurm →

Workflow partitioning

Manual (Slurm)

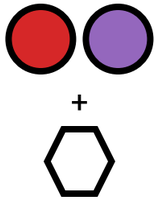


→ Slurm →



Workflow partitioning

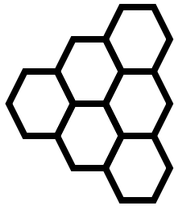
Manual (Slurm)



→ Slurm →

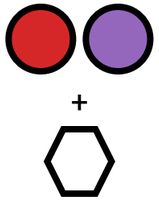
Meta-scheduling

→ Meta-scheduler → Slurm →



Workflow partitioning

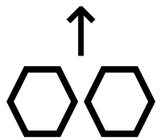
Manual (Slurm)

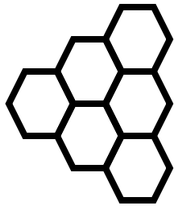


→ Slurm →

Meta-scheduling

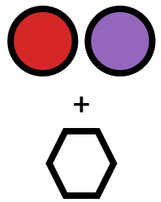
→ Meta-scheduler → Slurm →



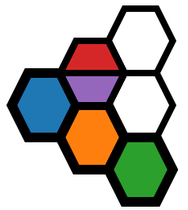


Workflow partitioning

Manual (Slurm)

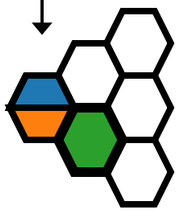


→ Slurm →



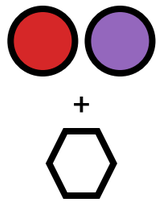
Meta-scheduling

→ Meta-scheduler → Slurm →

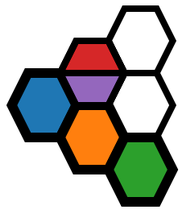


Workflow partitioning

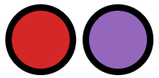
Manual (Slurm)



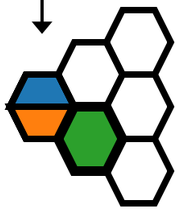
→ Slurm →

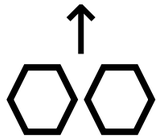


Meta-scheduling



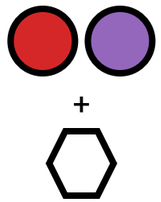
→ Meta-scheduler → Slurm →



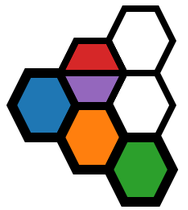


Workflow partitioning

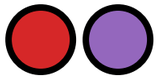
Manual (Slurm)



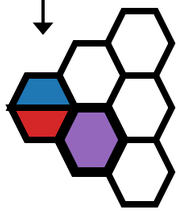
→ Slurm →



Meta-scheduling

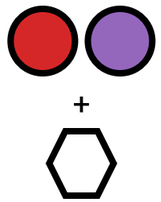


→ Meta-scheduler → Slurm →

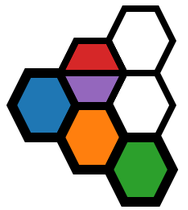


Workflow partitioning

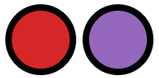
Manual (Slurm)



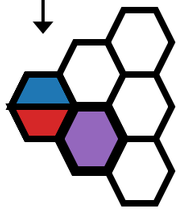
→ Slurm →



Meta-scheduling

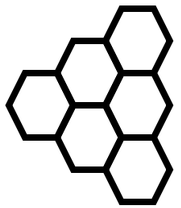


→ Meta-scheduler → Slurm →



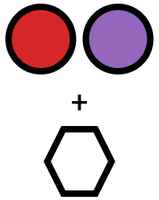
Meta-scheduling + automatic allocation

→ Meta-scheduler → Slurm →



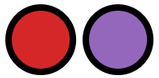
Workflow partitioning

Manual (Slurm)

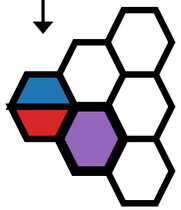


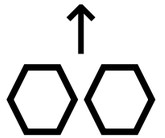
→ Slurm →

Meta-scheduling



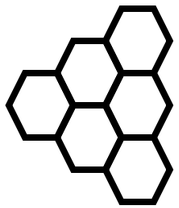
→ Meta-scheduler → Slurm →





Meta-scheduling + automatic allocation

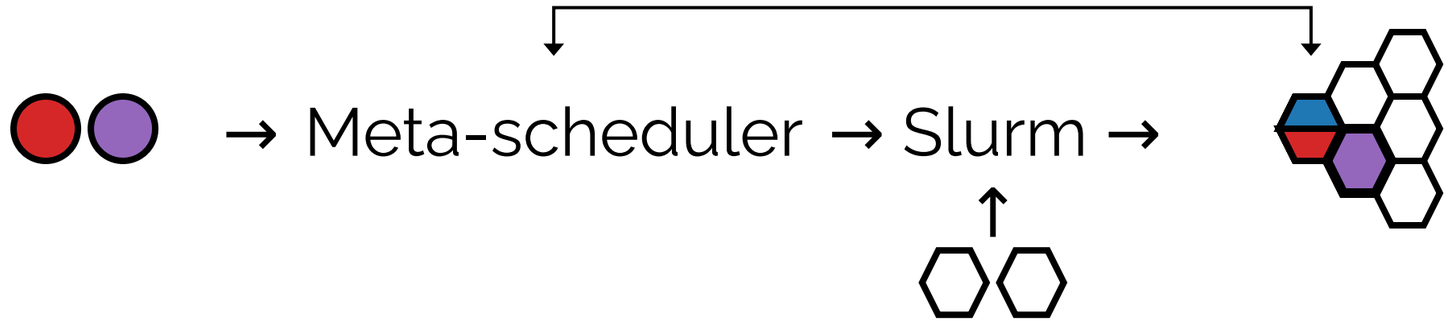
→ Meta-scheduler → Slurm →



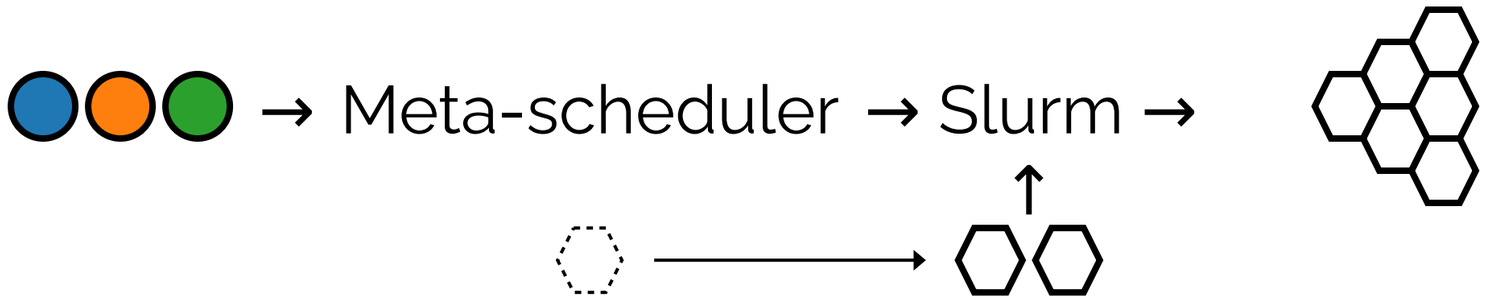
Workflow partitioning

Manual (Slurm)

Meta-scheduling



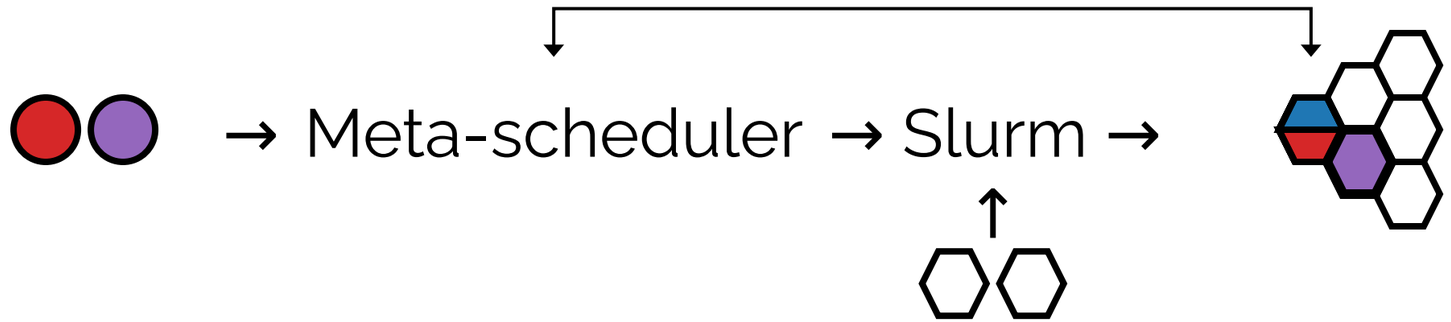
Meta-scheduling + automatic allocation



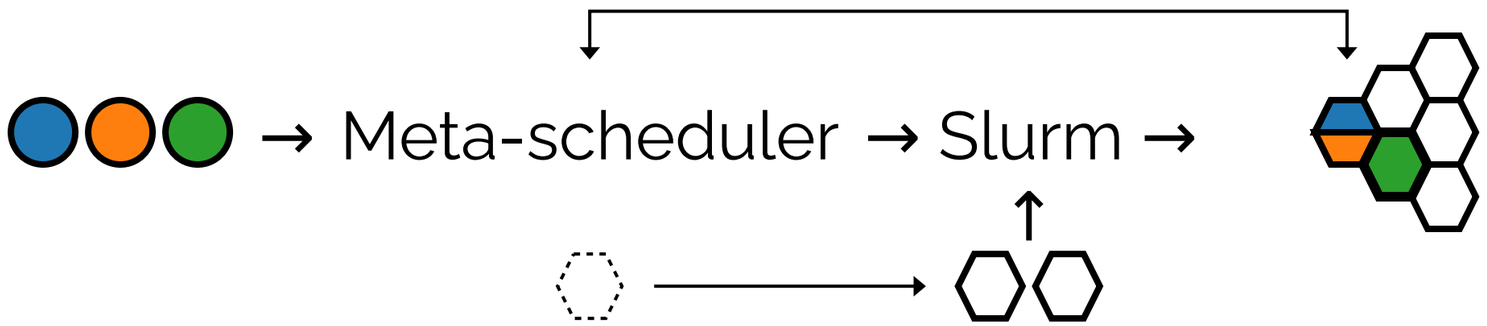
Workflow partitioning

Manual (Slurm)

Meta-scheduling



Meta-scheduling +
automatic allocation



Challenge: heterogeneous resource management

Challenge: heterogeneous resource management

- Arbitrary resources
  - GPUs, FPGAs, NICs, …

Challenge: heterogeneous resource management

- Arbitrary resources
  - GPUs, FPGAs, NICs, …
- Non-fungible resources
  - Resources have identity: "Execute task on GPUs 0 and 3"

Challenge: heterogeneous resource management

- Arbitrary resources
  - GPUs, FPGAs, NICs, …
- Non-fungible resources
  - Resources have identity: "Execute task on GPUs 0 and 3"
- Related resources
  - Task requires 4 cores in the same NUMA node

Challenge: heterogeneous resource management

- Arbitrary resources
  - GPUs, FPGAs, NICs, …
- Non-fungible resources
  - Resources have identity: "Execute task on GPUs 0 and 3"
- Related resources
  - Task requires 4 cores in the same NUMA node
- Fractional resources
  - Task requires 0.5 of a GPU

Challenge: heterogeneous resource management

- Arbitrary resources
  - GPUs, FPGAs, NICs, ...
- Non-fungible resources
  - Resources have identity: "Execute task on GPUs 0 and 3"
- Related resources
  - Task requires 4 cores in the same NUMA node
- Fractional resources
  - Task requires 0.5 of a GPU
- Resource variants
  - Task requires (1 GPU AND 1 CPU) OR 16 CPUs

HyperQueue

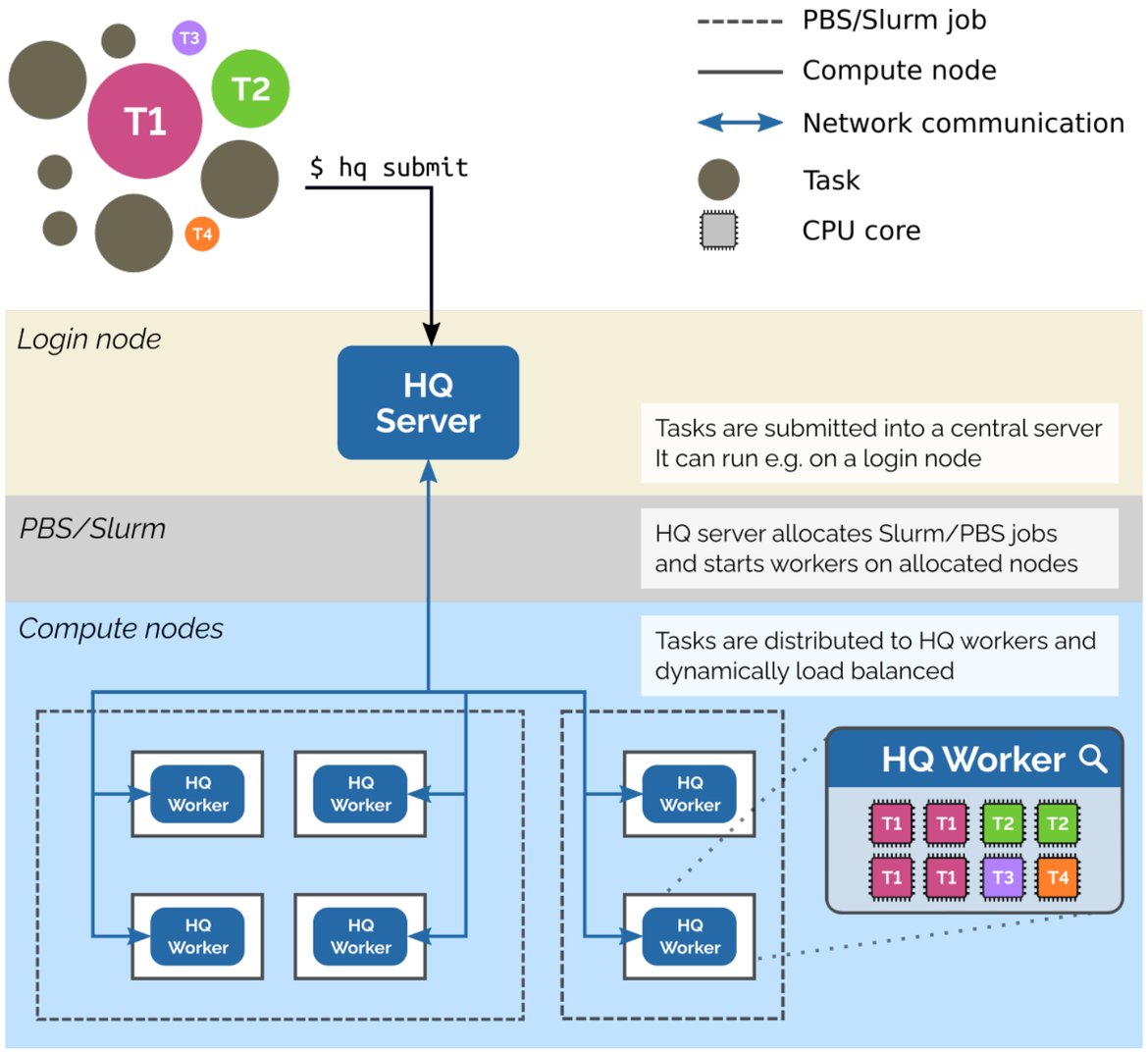
github.com/it4innovations/hyperqueue

HyperQueue

`github.com/it4innovations/hyperqueue`

Task runtime designed for HPC use-cases

Architecture



HyperQueue solutions for workflow challenges

- Meta-scheduling
  - Automatic allocation

HyperQueue solutions for workflow challenges

- Meta-scheduling
  - Automatic allocation
- Heterogeneous resource scheduling

HyperQueue solutions for workflow challenges

- Meta-scheduling
  - Automatic allocation
- Heterogeneous resource scheduling
- Multi-node tasks
  - Integrated with single node tasks

HyperQueue solutions for workflow challenges

- Meta-scheduling
  - Automatic allocation
- Heterogeneous resource scheduling
- Multi-node tasks
  - Integrated with single node tasks
- Fault-tolerance
  - Worker and server resilience

HyperQueue solutions for workflow challenges

- Meta-scheduling
  - Automatic allocation
- Heterogeneous resource scheduling
- Multi-node tasks
  - Integrated with single node tasks
- Fault-tolerance
  - Worker and server resilience
- Output streaming
  - Reduces networked filesystem contention

HyperQueue solutions for workflow challenges

- Meta-scheduling
  - Automatic allocation
- Heterogeneous resource scheduling
- Multi-node tasks
  - Integrated with single node tasks
- Fault-tolerance
  - Worker and server resilience
- Output streaming
  - Reduces networked filesystem contention
- Several programming interfaces
  - CLI, TOML workflow files, Python API

HyperQueue solutions for workflow challenges

- Meta-scheduling
  - Automatic allocation
- Heterogeneous resource scheduling
- Multi-node tasks
  - Integrated with single node tasks
- Fault-tolerance
  - Worker and server resilience
- Output streaming
  - Reduces networked filesystem contention
- Several programming interfaces
  - CLI, TOML workflow files, Python API
- Trivial deployment
  - Single binary, no dependencies, no *sudo* needed

Scalability evaluation

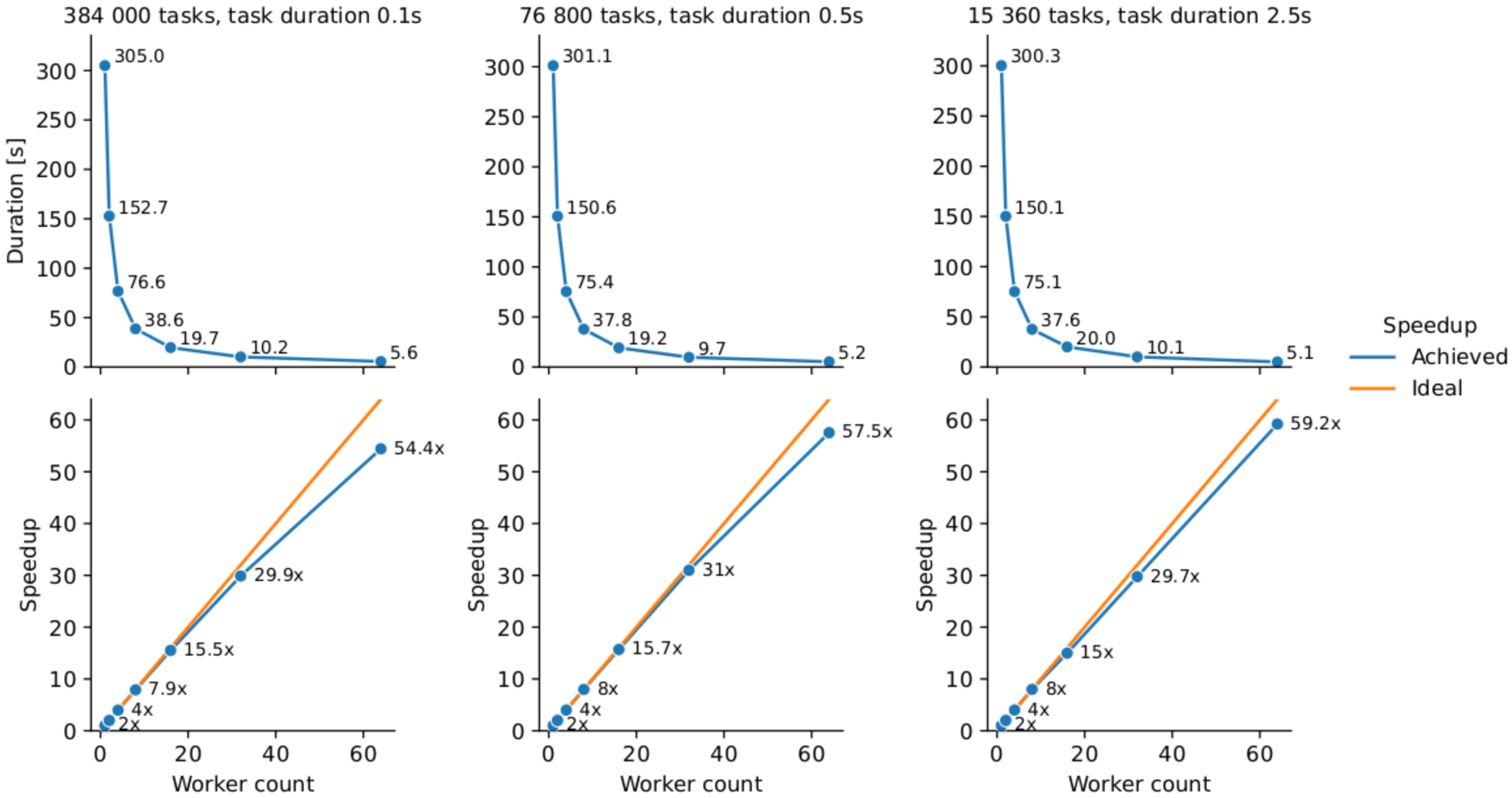


Figure 7.7: Strong scalability of HyperQueue with a fixed target makespan (300 s)

Scalability vs Dask evaluation

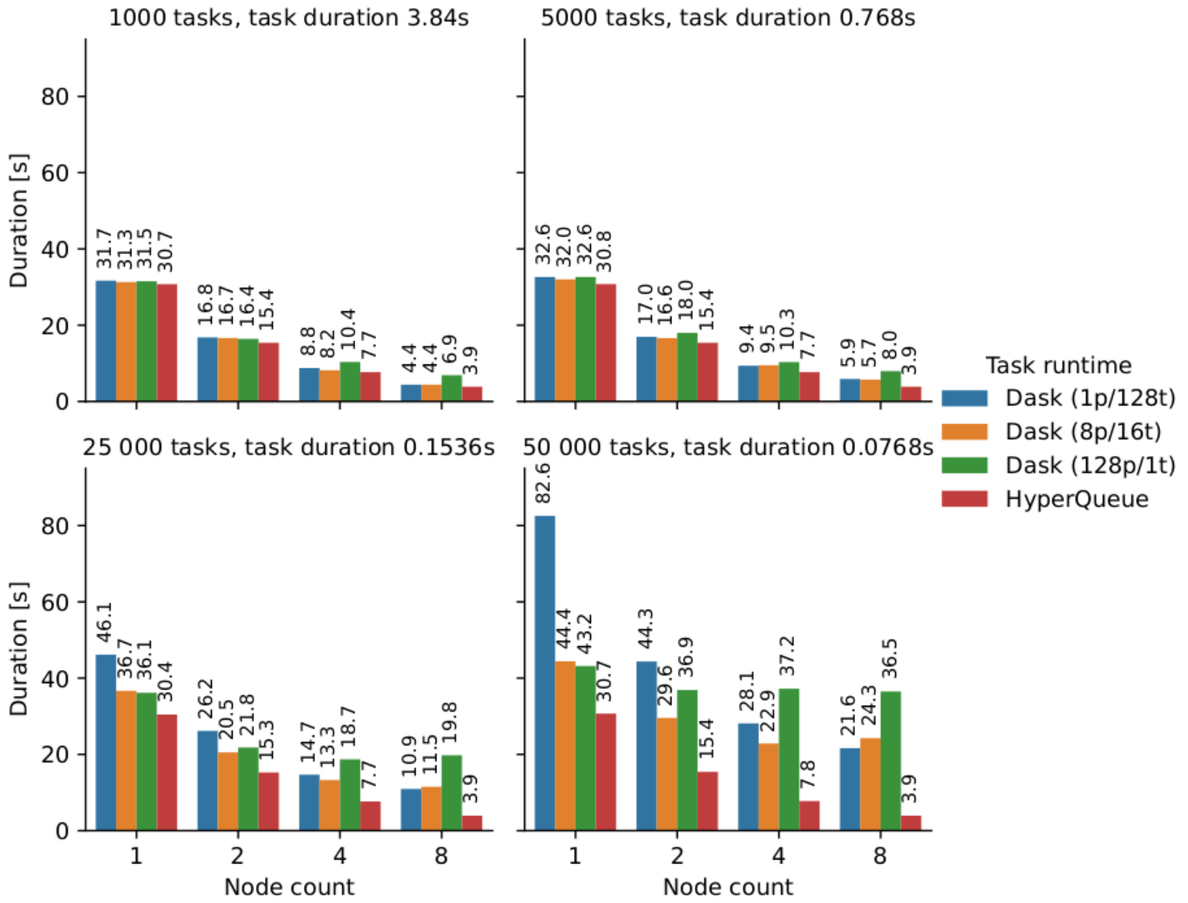


Figure 7.9: Scalability of HyperQueue vs Dask with a fixed target makespan (30 s)

Overhead evaluation

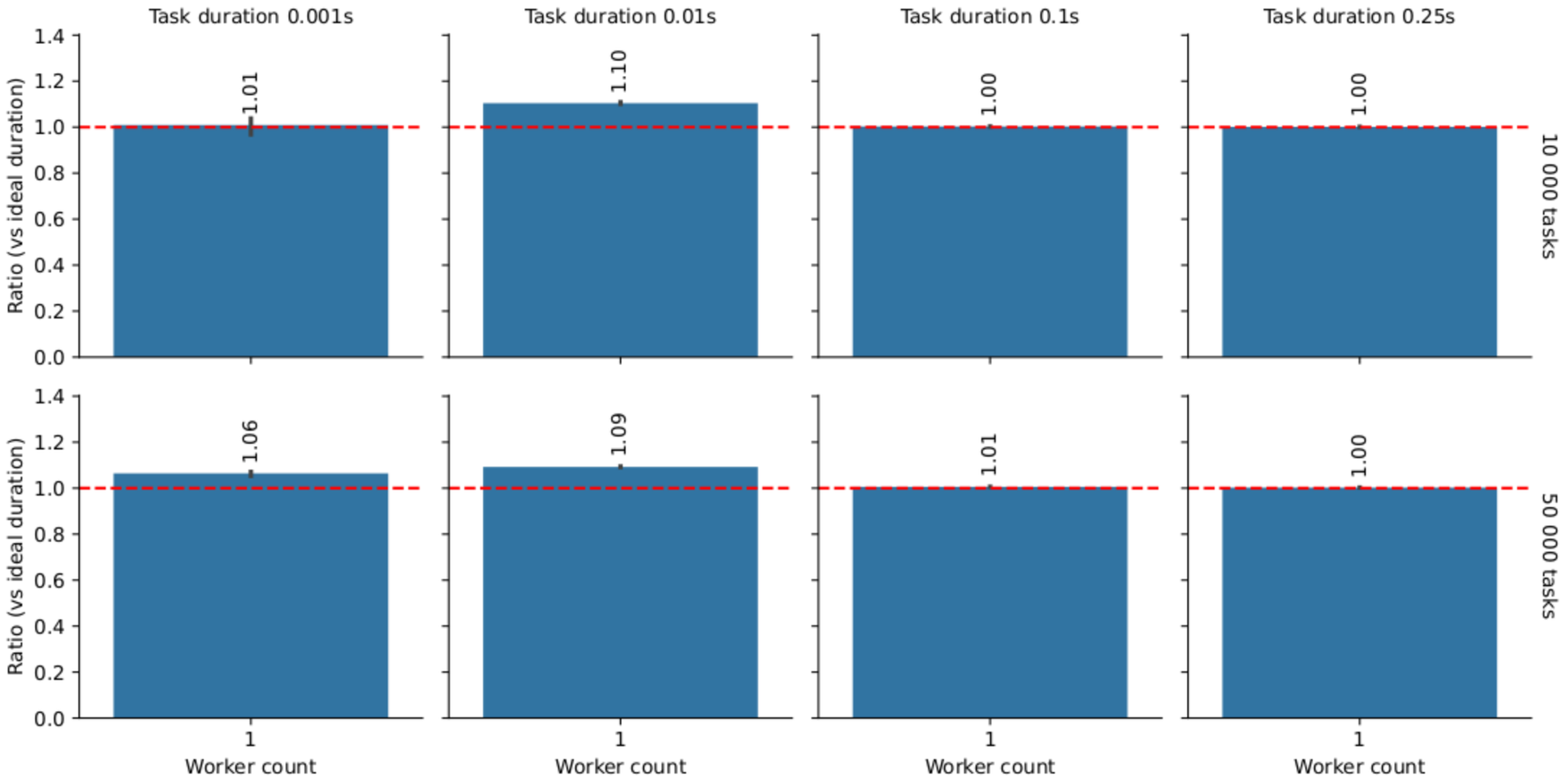


Figure 7.5: Total overhead of HyperQueue (ratio vs manual process execution)

Fractional resources evaluation

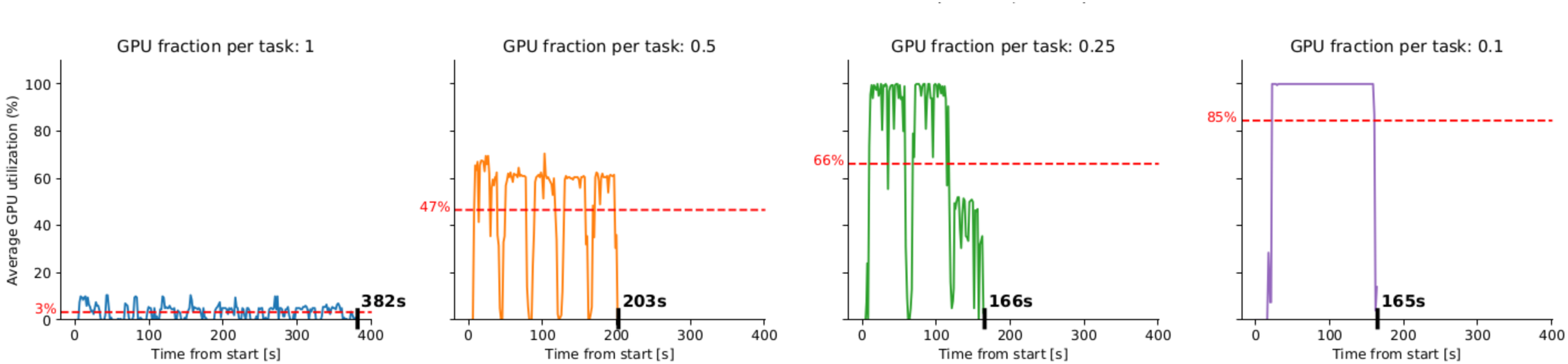


Figure 7.12: GPU hardware utilization improvements with fractional resource requirements

Comparison of HQ and similar tools

Task runtime	Task interfaces				Task dependencies		Fault tolerance		Meta-scheduling			Resources								Data transfers		Deployment
	CLI	Workflow file	Python API	Dynamic task graphs	Explicit	Inferred from files	Worker failure	Server failure	Graph partitioning	Automatic allocation	Generic workers	Basic resources	Custom resources	Non-fungible resources	Related resources	Fractional resources	Resource variants	Multi-node tasks	Directly between tasks	Output streaming	Runtime dependencies
GNU PARALLEL	👍	-	-	-	-	-	◐	-	-	-	-	-	-	-	-	-	-	-	-	👍	Perl
HYPERSHELL	👍	-	👍	-	-	-	●	👍	◐	👍	-	-	-	-	-	-	-	-	-	👍	Python
DASK	-	-	👍	👍	👍	-	●	-	●†	👍†	👍	👍	👍	-	-	-	-	-	👍	👍	Python
RAY	-	-	👍	👍	👍	-	●	👍*	●	-	👍	👍	👍	-	👍	👍	-	-	👍	👍	Python
PARSL	-	-	👍	👍	👍	-	●	-	●	👍	-	👍	-	-	-	-	-	👍	👍	-	Python
PYCOMPSS	👍	-	👍	👍	👍	-	●	👍*	◐	-	-	👍	-	-	👍	-	-	👍	👍	👍	Python, Java
PEGASUS	-	👍	👍	-	👍	👍	●	👍	◐	-	-	👍	-	-	-	-	-	-	-	👍	Python, Java, HTCondor [184]
BALSAM	-	-	👍	👍	👍	-	●	👍	●	👍	👍	👍	-	-	-	-	-	👍	👍	-	Python, PostgreSQL
AUTOSUBMIT	-	👍	-	-	👍	-	●	👍	◐	👍	-	👍	-	-	-	-	-	👍	-	-	Python
FIREWORKS	-	👍	👍	👍	👍	-	◐	👍	●	👍	👍	-	-	-	-	-	-	-	-	-	Python, MongoDB
MERLIN	-	👍	-	-	👍	-	◐	👍	●	-	-	👍	-	-	-	-	-	👍	-	-	Python, RabbitMQ/Redis
SNAKEMAKE	-	👍	👍	-	-	👍	●	-	◐	👍†	-	👍	👍	-	-	-	-	👍	-	-	Python
HYPERQUEUE	👍	👍	👍	👍	👍	-	●	👍	●	👍	👍	👍	👍	👍	👍	👍	👍	👍	-	👍	

👍 supported; - not supported; ● automatic; ◐ manual;
†with an external plugin; *with additional runtime dependencies

HyperQueue impact

- EU projects
  - 

Complex workflows, Python API
  - 

Heterogeneous resources
  - 

Multi-node tasks
  - 

Server resilience, data transfers

HyperQueue impact

- EU projects
  - LIGATE Complex workflows, Python API
  - EVEREST Heterogeneous resources
  - ACROSS Multi-node tasks
  - MAX Server resilience, data transfers
- HPC centers
  - VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER
  - LUMI
  - CINECA
  - CSC

HyperQueue impact

- EU projects
  - LIGATE Complex workflows, Python API
  - EVEREST Heterogeneous resources
  - ACROSS Multi-node tasks
  - MAX Server resilience, data transfers
- HPC centers
  - VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER
  - LUMI
  - CINECA
  - CSC
- Integration in other tools
  - nextflow
  - AiiDA
  - StreamFlow
  - UM-Bridge
  - HEAppE
  - ERT

HyperQueue impact

ARC-CE+HyperQueue based submission system of ATLAS jobs for Karolina HPC

M. Svatoš, J. Chudoba, P. Vokáč

6th Users Conference of IT4Innovations

3.-4.11.2022

Introduction

For several years, the distributed computing of the ATLAS experiment at the LHC (ADC) has been granted opportunistic use of computing resources of the Czech national HPC centre, IT4Innovations. Within the ADC, resources of IT4Innovations are attached to the Czech Tier2. Implementation of HyperQueue (a tool designed to simplify execution of large workflows on HPC clusters, developed by IT4Innovations) into the previously used ARC-CE based submission system allows tweaking of submitted jobs and hopefully helps with handling of pre-emption at IT4Innovations.

Submission system

Harvester, ARC-CE, sshfs, login node, HyperQueue server, ssh, Panda Server, http proxy (squid), http, PBS Pro, Storage, webdav /xroot, Lustre, Czech Tier2, IT4I

- the ARC-CE receives a pilot job, translates the job description into a script that can be run in the batch system, puts necessary files into a folder shared with the HPC via sshfs, and submits the job via ssh connection to the HyperQueue server running on a login node
- the HyperQueue server buffers the jobs and when there are enough of them, it submits jobs into the PBS Pro
- when the PBS job starts, HyperQueue jobs start in it (in sufficient numbers to fill the worker node - if available)
- in each HyperQueue job, pilot starts
- pilot contacts panda server through http proxy (Czech Tier2 squid) to receive payload job (as http is one of few open ports)
- when it receives a payload job, it gets input file from the Czech Tier2 storage via xroot or webdav (in a rucio container)
- then it starts the calculation (in software container)
- when the payload job finishes, it sends outputs to the Czech Tier2 storage via xroot or webdav (in a rucio container)
- when this is finished, pilot will request another payload job (if it can expect that the batch queue setting would allow it to finish)

2021:Migration to HyperQueue

Testing:Efficiency (CPU/Wall) | Testing:Number of cores | Production:Number of cores | Production:Walltlock | Production:Efficiency(CPU/Wall)

- In December 2021, the HyperQueue implementation was tested on Karolina using 16 and 32 cores and then put into production a few days before the year-end

2022: Living with pre-emption

Number of cores used on Karolina and Barbora | Efficiency (CPU/Wall) on Karolina and Barbora | Number of successful and failed jobs on Karolina and Barbora | Number of cores used by all Czech Tier2 resources | CPU consumption of all Czech Tier2 resources

- In 2022, new project which can use only pre-emptive queues started in September.
- Development to tune the submission system continues.
- Contribution of Barbora, which is not using the HyperQueue yet, is not significant in comparison with Karolina.
- Even with pre-emption, the IT4I provides significant resources to Czech Tier2 with high efficiency.

Acknowledgement

This work and computing resources at FZU were co-financed by projects CERN-C (CZ.02.1.01/0.0/0.0/16013/0001404) and CERN-CD (CZ.02.1.01/0.0/0.0/18_046/0016013) from EU funds and MSMT and projects CERN-CZ (LM2018104) and LTT17018.

ATLAS experiment (CERN)

ARC-CE+HyperQueue based submission system of ATLAS jobs for the Karolina HPC

HyperQueue impact

ARC-CE+HyperQueue based submission system of ATLAS jobs for Karolina HPC

M. Svatoš, J. Chudoba, P. Vokáč

6th Users Conference of IT4Innovations

3.-4.11.2022

Introduction

For several years, the distributed computing of the ATLAS experiment at the LHC (ADC) has been granted opportunistic use of computing resources of the Czech national HPC centre, IT4Innovations. Within the ADC, resources of IT4Innovations are attached to the Czech Tier2. Implementation of HyperQueue (a tool designed to simplify execution of large workflows on HPC clusters, developed by IT4Innovations) into the previously used ARC-CE based submission system allows tweaking of submitted jobs and hopefully helps with handling of pre-emption at IT4Innovations.

Submission system

Harvester, ARC-CE, Panda Server, http proxy (squid), Storage, Czech Tier2, sshfs, ssh, http, webdav /xroot, login node, HyperQueue server, PBS Pro, Lustre, IT4I

- the ARC-CE receives a pilot job, translates the job description into a script that can be run in the batch system, puts necessary files into a folder shared with the HPC via sshfs, and submits the job via ssh connection to the HyperQueue server running on a login node
- the HyperQueue server buffers the jobs and when there are enough of them, it submits jobs into the PBS Pro
- when the PBS job starts, HyperQueue jobs start in it (in sufficient numbers to fill the worker node - if available)
- in each HyperQueue job, pilot starts
- pilot contacts panda server through http proxy (Czech Tier2 squid) to receive payload job (as http is one of few open ports)
- when it receives a payload job, it gets input file from the Czech Tier2 storage via xroot or webdav (in a rucio container)
- then it starts the calculation (in software container)
- when the payload job finishes, it sends outputs to the Czech Tier2 storage via xroot or webdav (in a rucio container)
- when this is finished, pilot will request another payload job (if it can expect that the batch queue setting would allow it to finish)

2021:Migration to HyperQueue

Testing:Efficiency (CPU/Wall), Testing:Number of cores, Production:Number of cores, Production:Walltclock, Production:Efficiency(CPU/Wall)

32 cores, 16 cores, 128 cores, 90%

- In December 2021, the HyperQueue implementation was tested on Karolina using 16 and 32 cores and then put into production a few days before the year-end

2022: Living with pre-emption

Number of cores used on Karolina and Barbora, Efficiency (CPU/Wall) on Karolina and Barbora, Number of successful and failed jobs on Karolina and Barbora, Number of cores used by all Czech Tier2 resources, CPU consumption of all Czech Tier2 resources

Karolina, Barbora, GRID, HPC, BOINC, HPC(43%), GRID(56%)

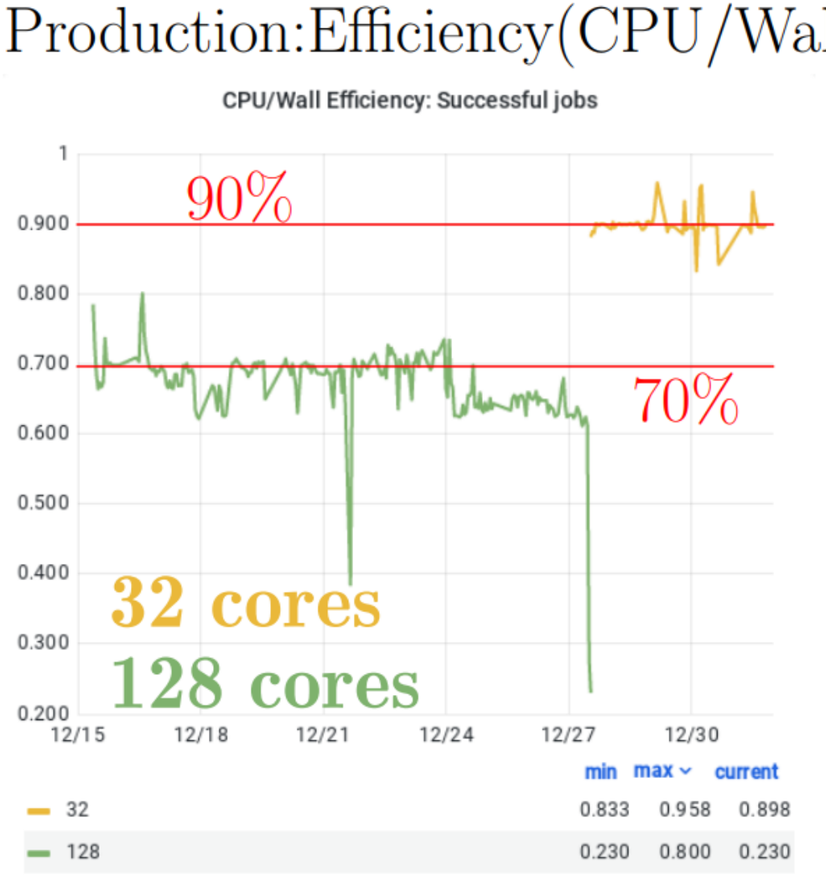
- In 2022, new project which can use only pre-emptive queues started in September.
- Development to tune the submission system continues.
- Contribution of Barbora, which is not using the HyperQueue yet, is not significant in comparison with Karolina.
- Even with pre-emption, the IT4I provides significant resources to Czech Tier2 with high efficiency.

Acknowledgement

This work and computing resources at FZU were co-financed by projects CERN-C (CZ.02.1.01/0.0/0.0/16013/0001404) and CERN-CD (CZ.02.1.01/0.0/0.0/18_046/0016013) from EU funds and MSMT and projects CERN-CZ (LM2018104) and LTT17018.

ATLAS experiment (CERN)

ARC-CE+HyperQueue based submission system of ATLAS jobs for the Karolina HPC



HyperQueue impact

LIGATE Pose selector workflow

HyperQueue impact

LIGATE Pose selector workflow

- 6M+ MD simulations on 4k+ ligands
- 240k GPU hours on LUMI-G cluster
- 2M CPU hours on MeluXina cluster

HyperQueue impact

LIGATE Pose selector workflow

- 6M+ MD simulations on 4k+ ligands
- 240k GPU hours on LUMI-G cluster
- 2M CPU hours on MeluXina cluster

> “
> ...making both workflows some of the largest molecular dynamics campaigns ever performed...
> LIGATE Deliverable D7.1 (lessons learned) ”

Outcome

Outcome

- Integrated solution for workflows on HPC

Outcome

- Integrated solution for workflows on HPC
- Ergonomics
  - Meta-scheduling
  - Fine-grained heterogeneous resource management
  - Multi-node tasks
  - Trivial deployment

Outcome

- Integrated solution for workflows on HPC
- Ergonomics
  - Meta-scheduling
  - Fine-grained heterogeneous resource management
  - Multi-node tasks
  - Trivial deployment
- Efficiency
  - Low overhead per task
  - Millions of tasks, hundreds of nodes

Outcome

- Integrated solution for workflows on HPC
- Ergonomics
  - Meta-scheduling
  - Fine-grained heterogeneous resource management
  - Multi-node tasks
  - Trivial deployment
- Efficiency
  - Low overhead per task
  - Millions of tasks, hundreds of nodes
- Open source
  -  `github.com/it4innovations/hyperqueue`

Thesis objectives fulfillment

1. Identify HPC workflow challenges ✓

Thesis objectives fulfillment

1. Identify HPC workflow challenges ✓
2. Design approaches for overcoming them ✓
   - Meta-scheduling approach
   - Heterogeneous resource management

Thesis objectives fulfillment

1. Identify HPC workflow challenges ✓
2. Design approaches for overcoming them ✓
   - Meta-scheduling approach
   - Heterogeneous resource management
3. Implement them in a task runtime ✓
   - HyperQueue

Thesis objectives fulfillment

1. Identify HPC workflow challenges ✓
2. Design approaches for overcoming them ✓
   - Meta-scheduling approach
   - Heterogeneous resource management
3. Implement them in a task runtime ✓
   - HyperQueue
4. Analyze results on real use-cases ✓
   - LIGATE, CERN

Thank you for your attention

Slides made with https://github.com/spirali/elsie

Publications related to thesis

- **Analysis of workflow schedulers in simulated distributed environments**
  Jakub Beránek, Ada Böhm, Vojtěch Cima
  The Journal of Supercomputing 2022

- **Runtime vs Scheduler: Analyzing Dask's Overheads**
  Ada Böhm, *Jakub Beránek*
  IEEE/ACM Workflows in Support of Large-Scale Science (WORKS) 2020

- **HyperQueue: Efficient and ergonomic task graphs on HPC clusters**
  Jakub Beránek, Ada Böhm, Gianluca Palermo, Jan Martinovič, Branislav Jansík
  SoftwareX 2024

Publications unrelated to thesis

Network-Accelerated Non-Contiguous Memory Transfer
S. Di Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, *J. Beránek*, M. Besta, L. Benini, D. Roweth, T. Hoefler
SC (International Conference for High Performance Computing, Networking, Storage and Analysis) 2019

Streaming Message Interface: High-Performance Distributed Memory Programming on Reconfigurable Hardware
T. De Matteis, J. de Fine Licht, *J. Beránek*, T. Hoefler
SC (International Conference for High Performance Computing, Networking, Storage and Analysis) 2019

A RISC-V in-Network Accelerator for Flexible High-Performance Low-Power Packet Processing
S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, *J. Beránek*, L. Benini, T. Hoefler
ISCA (International Symposium on Computer architecture) 2021

SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory System
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, T. Hoefler
IEEE/ACM MICRO (International Symposium on Microarchitecture) 2021

GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra
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, T. Hoefler
PVLDB 2021

Tunable and Portable Extreme-Scale Drug Discovery Platform at Exascale: The LIGATE Approach
G. Palermo, G. Accordi, D. Gadioli, E. Vitali, C. Silvano, B. Guindani, D. Ardagna, A. 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. Alexander, G. Tauriello, T. Schwede, J. Durairaj, A. Emerson, F. Ficarelli, S. Wingbermühle, E. Lindahl, D. Gregori, E. Sana, S. Coletti, P. Gschwandtner
Proceedings of the 20th ACM International Conference on Computing Frontiers

pyCaverDock: Python implementation of the popular tool for analysis of ligand transport with advanced caching and batch calculation support
O. Vávra, *J. Beránek*, J. Šťourač, M. Šurkovský, J. Filipovič, J. Damborský, J. Martinovič, D. Bednář
Bioinformatics 2023

Haydi: Rapid Prototyping and Combinatorial Objects
Stanislav Böhm, *Jakub Beránek*, Martin Šurkovský
Foundations of Information and Knowledge Systems 2018

Alternative Paths Reordering Using Probabilistic Time-Dependent Routing
M. Golasowski, *J. Beránek*, M. Šurkovský, L. Rapant, D. Szturcová, J. Martinovič, Kateřina Slaninová
Advances in Networked-based Information Systems 2020

A Distributed Environment for Traffic Navigation Systems
J. Martinovič, M. Golasowski, K. Slaninová, *J. Beránek*, M. Šurkovský, L. Rapant, D. Szturcová, R. Cmar
Complex, Intelligent, and Software Intensive Systems 2020

Question 1 (doc. Mgr. Jiří Dvorský, Ph.D.)

> “
> Will it be possible to apply HyperQueue to a quantum computer as well?
> ”

Question 1 (prof. Ismail Hakki Toroslu)

> “
> The thesis identifies key challenges in task scheduling and resource management on heterogeneous supercomputers. What are the most significant bottlenecks and how RSDS or HyperQueue addresses these bottlenecks?
> ”

Question 1 (prof. Ismail Hakki Toroslu)

> “
> The thesis identifies key challenges in task scheduling and resource management on heterogeneous supercomputers. What are the most significant bottlenecks and how RSDS or HyperQueue addresses these bottlenecks?
> ”

- RSDS: runtime optimizations, scalability

Question 1 (prof. Ismail Hakki Toroslu)

> “
> The thesis identifies key challenges in task scheduling and resource management on heterogeneous supercomputers. What are the most significant bottlenecks and how RSDS or HyperQueue addresses these bottlenecks?
> ”

- RSDS: runtime optimizations, scalability
- HQ: meta-scheduling, heterogeneous resource management

Question 2 (prof. Ismail Hakki Toroslu)

> “
> It is shown that simple scheduling heuristics like work-stealing can compete with more complex algorithms. Under what conditions would a more complex algorithm be necessary?
> ”

Question 2 (prof. Ismail Hakki Toroslu)

> “
> It is shown that simple scheduling heuristics like work-stealing can compete with more complex algorithms. Under what conditions would a more complex algorithm be necessary?
> ”

- Specialized use-cases

Question 2 (prof. Ismail Hakki Toroslu)

> “
> It is shown that simple scheduling heuristics like work-stealing can compete with more complex algorithms. Under what conditions would a more complex algorithm be necessary?
> ”

- Specialized use-cases
- Memory consumption limits

Question 2 (prof. Ismail Hakki Toroslu)

> “
> It is shown that simple scheduling heuristics like work-stealing can compete with more complex algorithms. Under what conditions would a more complex algorithm be necessary?
> ”

- Specialized use-cases
- Memory consumption limits
- Latency (not throughput) optimized scheduling

Question 3 (prof. Ismail Hakki Toroslu)

> “
> Thesis mentions that Python runtime overhead is a critical bottleneck in Dask. How Rust-based server (RSDS) reduces this overhead, and why Rust was chosen for this task over other languages?
>
> ”

Question 3 (prof. Ismail Hakki Toroslu)

> “
> Thesis mentions that Python runtime overhead is a critical bottleneck in Dask. How Rust-based server (RSDS) reduces this overhead, and why Rust was chosen for this task over other languages?
> ”

- Compact task storage, compact message format

Question 3 (prof. Ismail Hakki Toroslu)

> “
> Thesis mentions that Python runtime overhead is a critical bottleneck in Dask. How Rust-based server (RSDS) reduces this overhead, and why Rust was chosen for this task over other languages?
> ”

- Compact task storage, compact message format
- Compiled to native code, small runtime, no GIL

Question 3 (prof. Ismail Hakki Toroslu)

> “
> Thesis mentions that Python runtime overhead is a critical bottleneck in Dask. How Rust-based server (RSDS) reduces this overhead, and why Rust was chosen for this task over other languages?
> ”

- Compact task storage, compact message format
- Compiled to native code, small runtime, no GIL
- Memory safety, data race safety

Question 4 (prof. Ismail Hakki Toroslu)

> “
> HyperQueue has already been adopted by several European supercomputing centers. Is there any feedback that could be used to improve HyperQueue?
> ”

Question 4 (prof. Ismail Hakki Toroslu)

> “
> HyperQueue has already been adopted by several European supercomputing centers. Is there any feedback that could be used to improve HyperQueue?
> ”

- Missing features (data transfers, extended Python API)

Question 4 (prof. Ismail Hakki Toroslu)

> HyperQueue has already been adopted by several European supercomputing centers. Is there any feedback that could be used to improve HyperQueue?

- Missing features (data transfers, extended Python API)
- More transparent automatic allocation

Question 5 (prof. Ismail Hakki Toroslu)

> “
> Estee is designed to benchmark task schedulers and help prototype new algorithms. What future extensions or improvements can be done on Estee, especially from HPC point of view?
> ”

Question 5 (prof. Ismail Hakki Toroslu)

> “
> Estee is designed to benchmark task schedulers and help prototype new algorithms. What future extensions or improvements can be done on Estee, especially from HPC point of view?
> ”

- Improve simulation throughput (e.g. Rust core + Python API)

Question 5 (prof. Ismail Hakki Toroslu)

> “
> Estee is designed to benchmark task schedulers and help prototype new algorithms. What future extensions or improvements can be done on Estee, especially from HPC point of view?
> ”

- Improve simulation throughput (e.g. Rust core + Python API)
- Heterogeneous resource management

Question 1 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Does the support of only acyclic task graphs may impose less efficient solutions in cases where loops involved regions of tasks? What kind of advantages and disadvantages do you see in using acyclic task graphs? Do the iterative task graphs transformed into acyclic task graphs (such as L-DAG), or each iteration as a separate task, fully solve the problem without disadvantages?
> ”

Question 1 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Does the support of only acyclic task graphs may impose less efficient solutions in cases where loops involved regions of tasks? What kind of advantages and disadvantages do you see in using acyclic task graphs? Do the iterative task graphs transformed into acyclic task graphs (such as L-DAG), or each iteration as a separate task, fully solve the problem without disadvantages?
> ”

- Iterations complicate the programming model

Question 1 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Does the support of only acyclic task graphs may impose less efficient solutions in cases where loops involved regions of tasks? What kind of advantages and disadvantages do you see in using acyclic task graphs? Do the iterative task graphs transformed into acyclic task graphs (such as L-DAG), or each iteration as a separate task, fully solve the problem without disadvantages?
> ”

- Iterations complicate the programming model
- Static loop count => unrolling

Question 1 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Does the support of only acyclic task graphs may impose less efficient solutions in cases where loops involved regions of tasks? What kind of advantages and disadvantages do you see in using acyclic task graphs? Do the iterative task graphs transformed into acyclic task graphs (such as L-DAG), or each iteration as a separate task, fully solve the problem without disadvantages?
> ”

- Iterations complicate the programming model
- Static loop count => unrolling
- Dynamic loop count => dynamic tasks

Question 2 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Based on your experience, is there a list of recommendations to guide developers/users w.r.t. task granularity? Is the control of the task graph granularity, based on the invocation of functions (e.g., in Python), a good approach?
> ”

Question 2 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Based on your experience, is there a list of recommendations to guide developers/users w.r.t. task granularity? Is the control of the task graph granularity, based on the invocation of functions (e.g., in Python), a good approach?
> ”

- More granularity => better load balancing, more overhead

Question 2 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Based on your experience, is there a list of recommendations to guide developers/users w.r.t. task granularity? Is the control of the task graph granularity, based on the invocation of functions (e.g., in Python), a good approach?
> ”

- More granularity => better load balancing, more overhead
- Task runtimes should adopt to the granularity required by its users

Question 3 (assoc. prof. João Manuel Paiva Cardoso)

> “
> What are the implications of not considering pipelining execution of tasks? How could this restriction be solved?
> ”

Question 3 (assoc. prof. João Manuel Paiva Cardoso)

> “
> What are the implications of not considering pipelining execution of tasks? How could this restriction be solved?
> ”

- Data transfers between tasks (inputs/outputs => first-class concept)

Question 3 (assoc. prof. João Manuel Paiva Cardoso)

> “
> What are the implications of not considering pipelining execution of tasks? How could this restriction be solved?
> ”

- Data transfers between tasks (inputs/outputs => first-class concept)
- Complicates task completion semantics (fault tolerance)

Question 3 (assoc. prof. João Manuel Paiva Cardoso)

> “What are the implications of not considering pipelining execution of tasks? How could this restriction be solved?”

- Data transfers between tasks (inputs/outputs => first-class concept)
- Complicates task completion semantics (fault tolerance)
- Potential future work for HyperQueue

Question 4 (assoc. prof. João Manuel Paiva Cardoso)

> “
> How does DASK support symbolic task graph representations? Should they need to be rematerialized, or can the runtime system directly use the symbolic representation to schedule and instantiate them?
> ”

Question 4 (assoc. prof. João Manuel Paiva Cardoso)

> “How does DASK support symbolic task graph representations? Should they need to be rematerialized, or can the runtime system directly use the symbolic representation to schedule and instantiate them?”

- Task graphs are rematerialized on the Dask server

Question 4 (assoc. prof. João Manuel Paiva Cardoso)

> “
> How does DASK support symbolic task graph representations? Should they need to be rematerialized, or can the runtime system directly use the symbolic representation to schedule and instantiate them?
> ”

- Task graphs are rematerialized on the Dask server
- Incremental rematerialization

Question 4 (assoc. prof. João Manuel Paiva Cardoso)

> “
> How does DASK support symbolic task graph representations? Should they need to be rematerialized, or can the runtime system directly use the symbolic representation to schedule and instantiate them?
> ”

- Task graphs are rematerialized on the Dask server
- Incremental rematerialization
- Experimental research: represent task graphs with finite automata

Question 5 (assoc. prof. João Manuel Paiva Cardoso)

> “
> How does the information for each task input into the runtime system,
> e.g., the number of nodes needed? Is the information in the form of annotations?
> Is it in the task graph representation?
>
> ”

Question 5 (assoc. prof. João Manuel Paiva Cardoso)

> “
> How does the information for each task input into the runtime system,
> e.g., the number of nodes needed? Is the information in the form of annotations?
> Is it in the task graph representation?
>
> ”

- Dependency/resource information is embedded in the DAG

Question 5 (assoc. prof. João Manuel Paiva Cardoso)

> “
> How does the information for each task input into the runtime system,
> e.g., the number of nodes needed? Is the information in the form of annotations?
> Is it in the task graph representation?
> ”

- Dependency/resource information is embedded in the DAG
- Configuration depends on the interface (Python, CLI, TOML)

Question 5 (assoc. prof. João Manuel Paiva Cardoso)

> “
> How does the information for each task input into the runtime system,
> e.g., the number of nodes needed? Is the information in the form of annotations?
> Is it in the task graph representation?
>
> ”

- Dependency/resource information is embedded in the DAG
- Configuration depends on the interface (Python, CLI, TOML)
- HQ uses structural sharing for shared requirements

Question 6 (assoc. prof. João Manuel Paiva Cardoso)

> “
> What kind of fault-tolerance schemes are included?
> How can other fault-tolerance schemes be included?
> ”

Question 6 (assoc. prof. João Manuel Paiva Cardoso)

> “What kind of fault-tolerance schemes are included?
> How can other fault-tolerance schemes be included?”

- Tasks are resilient to worker failures (task instances)

Question 6 (assoc. prof. João Manuel Paiva Cardoso)

> “What kind of fault-tolerance schemes are included?
> How can other fault-tolerance schemes be included?”

- Tasks are resilient to worker failures (task instances)
- Tasks are resilient to server failures (journaling)

Question 6 (assoc. prof. João Manuel Paiva Cardoso)

> “What kind of fault-tolerance schemes are included?
> How can other fault-tolerance schemes be included?”

- Tasks are resilient to worker failures (task instances)
- Tasks are resilient to server failures (journaling)
- Potential future work: task checkpointing

Question 7 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The example in Fig. 3.2 seems to not consider the different execution time of tasks according to the worker. When there are differences, for instance because of the use of different computing units (CPU vs GPU), how does the scheduling decide? What kind of possibilities do you see if we also consider variants of the tasks, using the same CPU, but with different energy/execution time trade-offs?
> ”

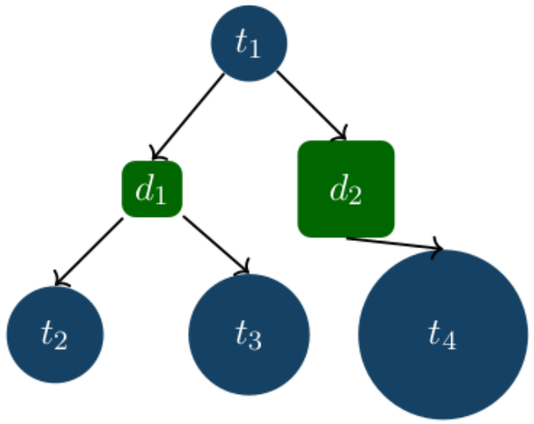
Question 7 (assoc. prof. João Manuel Paiva Cardoso)

“

The example
according to t
use of differer
What kind of p
using the sam

n time of tasks
cause of the
uling decide?
he tasks,
-offs?

”



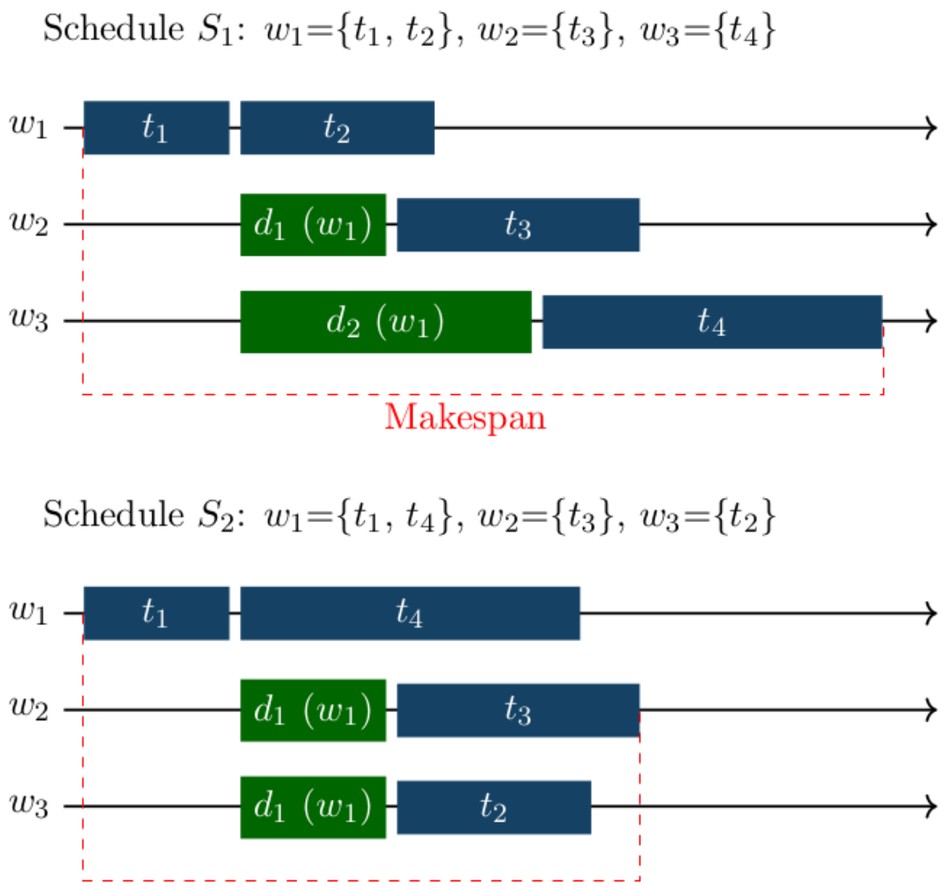


Figure 3.2: Task graph executed with two different schedules

Question 8 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The use of checkpointing, although convenient, usually introduces a large overhead. Is this overhead automatically managed or it needs to be explicitly managed by the user?
>
> ”

Question 8 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The use of checkpointing, although convenient, usually introduces a large overhead. Is this overhead automatically managed or it needs to be explicitly managed by the user?
> ”

- HQ doesn't perform task checkpointing

Question 8 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The use of checkpointing, although convenient, usually introduces a large overhead. Is this overhead automatically managed or it needs to be explicitly managed by the user?
> ”

- HQ doesn't perform task checkpointing
- HQ optionally stores a (GZIP-compressed) event journal on disk

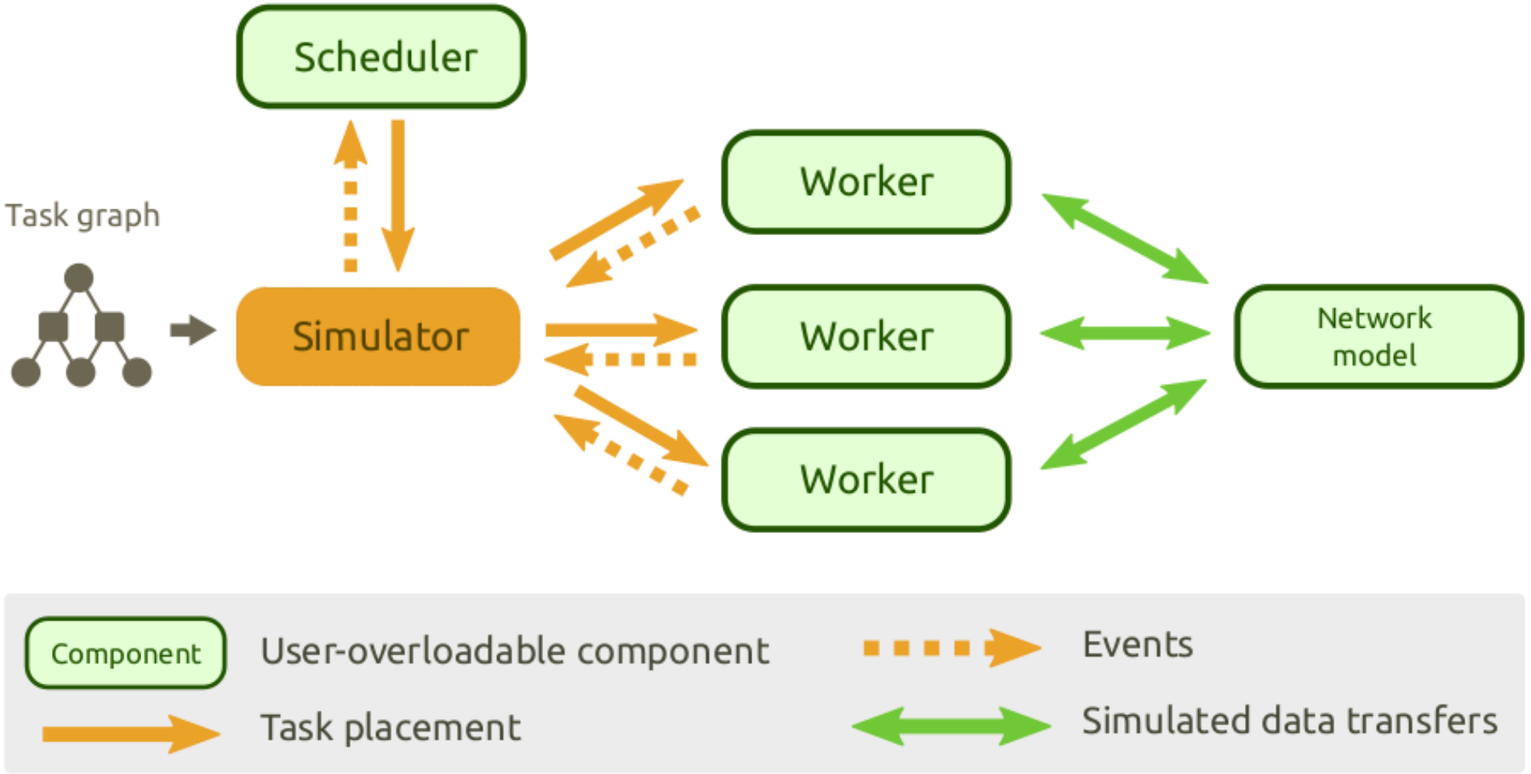
Question 9 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The ESTEE architecture shown in Fig. 5.1 in unclear in terms of some of the inputs to the framework. E.g., allocated resources, target clustering configuration, etc. Could you explain in more detail how the inputs are described and what kind of inputs are expected? Also, in listing 5.1 there is no mention of target cluster, #CPUs, connections, etc. Could you explain more about how that information is input? Could you explain about the ESTEE simple “communication network model”.
> ”

Question 9 (assoc. prof. João Manuel Paiva Cardoso)

“

The ESTEE arc
to the framew
Could you exp
inputs are exp
connections, e
Could you exp



of the inputs
ration, etc.
hat kind of
cluster, #CPUs,
n is input?
model”.

”

Figure 5.1: ESTEE architecture

Question 9 (assoc. prof. João Manuel Paiva Cardoso)

“

```
# 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)
```

”

Question 10 (assoc. prof. João Manuel Paiva Cardoso)

> “
> In terms of the scheduler, does the runtime system use updated execution times measured at runtime? And if so, how does it deal with that?
> ”

Question 10 (assoc. prof. João Manuel Paiva Cardoso)

> “In terms of the scheduler, does the runtime system use updated execution times measured at runtime? And if so, how does it deal with that?”

- ESTEE does not do that, it is emulated by the *mean* imode

Question 10 (assoc. prof. João Manuel Paiva Cardoso)

> “
> In terms of the scheduler, does the runtime system use updated execution times measured at runtime? And if so, how does it deal with that?
> ”

- ESTEE does not do that, it is emulated by the *mean* imode
- Users decide task durations, so measuring them is not so important

Question 11 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Do the schedulers and the evaluation in subsection 5.2 deal with goals to use fewer CPUs as much as possible (part of the work selection strategy)? If so, how does it work?
> ”

Question 12 (assoc. prof. João Manuel Paiva Cardoso)

> “
> When considering different task graphs and the impact of them on the gap between the two network models in subsection 5.2, do you think that understanding the shape and main characteristics of the task graphs would help to improve scheduling decisions?
> ”

Question 12 (assoc. prof. João Manuel Paiva Cardoso)

> “When considering different task graphs and the impact of them on the gap between the two network models in subsection 5.2, do you think that understanding the shape and main characteristics of the task graphs would help to improve scheduling decisions?”

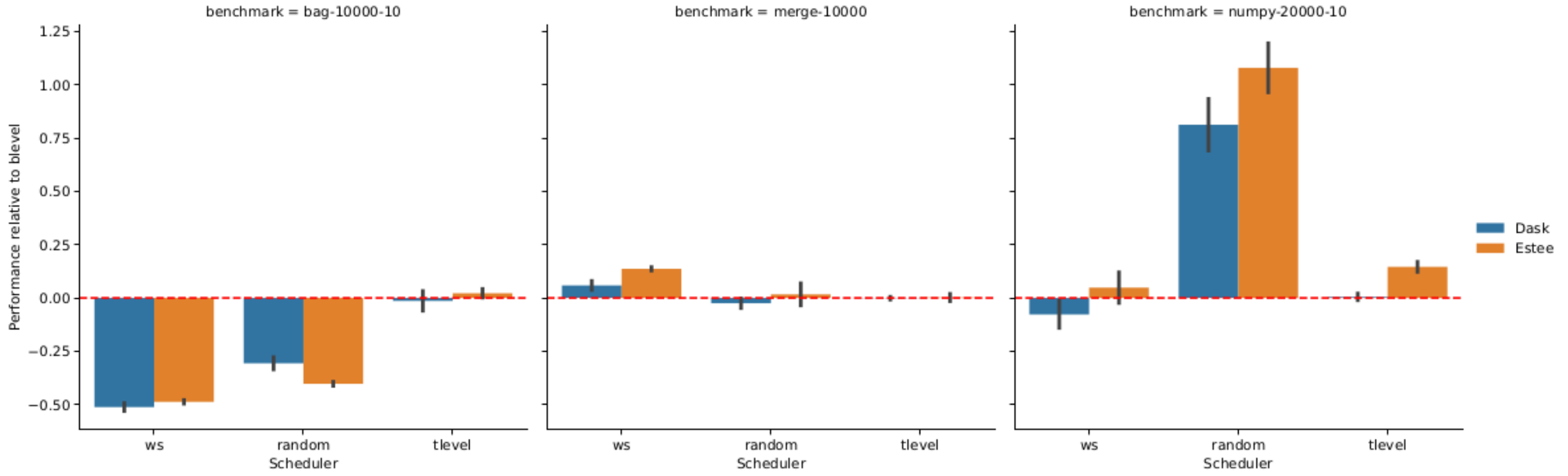
- Scheduling algorithms are heavily based on heuristics

Question 13 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Trends in terms of scheduler performance relative to b-level (in DASK and ESTEE, page 75) with different cluster configurations? Any thoughts about the effect of the task graph shapes/patterns on the accuracy of relative performance?
> ”

Question 13 (assoc. prof. João Manuel Paiva Cardoso)

“
Tre
pa
of



EE,

”

Figure 5.7: Scheduler performance relative to *blevel* in DASK and ESTEE

Question 14 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Any task graph pattern/shape would be able to indicate when a scheduler is better than others and thus could be the one used? Use of the best for a particular task graph and cluster configuration.
> ”

Question 14 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Any task graph pattern/shape would be able to indicate when a scheduler is better than others and thus could be the one used? Use of the best for a particular task graph and cluster configuration.
> ”

- Graph shape *can* be a heuristic for selecting a scheduler

Question 15 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Any thoughts based on the experiments about the MSD (minimal scheduling delay) impact according to the scheduler and task graph? Long MSD implies less decisions of the scheduler but more knowledge (arrival of more task events).
> Does MSD imply similar effects on different information modes?
> ”

Question 15 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Any thoughts based on the experiments about the MSD (minimal scheduling delay) impact according to the scheduler and task graph? Long MSD implies less decisions of the scheduler but more knowledge (arrival of more task events). Does MSD imply similar effects on different information modes?
> ”

- Configuring MSD is required to improve runtime performance

Question 15 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Any thoughts based on the experiments about the MSD (minimal scheduling delay) impact according to the scheduler and task graph? Long MSD implies less decisions of the scheduler but more knowledge (arrival of more task events). Does MSD imply similar effects on different information modes?
> ”

- Configuring MSD is required to improve runtime performance
- Unclear how to predict its effect on scheduling (heuristics)

Question 16 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The task runtime optimization evaluations would benefit from the knowledge of optimal scheduling. This could be very important to understand the potential for further improvements in terms of the actual results with heuristics.
> Do you see an experimental way to know those optimal results?
> ”

Question 16 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The task runtime optimization evaluations would benefit from the knowledge of optimal scheduling. This could be very important to understand the potential for further improvements in terms of the actual results with heuristics.
> Do you see an experimental way to know those optimal results?
> ”

- Complex benchmarks generated from real-world numpy/pandas code

Question 16 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The task runtime optimization evaluations would benefit from the knowledge of optimal scheduling. This could be very important to understand the potential for further improvements in terms of the actual results with heuristics.
> Do you see an experimental way to know those optimal results?
> ”

- Complex benchmarks generated from real-world numpy/pandas code
- NP-hard to get optimal scheduling (brute-force)

Question 17 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The limitation of tasks in pure Python and on the constraint to workers behave in a single-threaded fashion, … may have a high overhead. Any thoughts on that overhead and when it is adequate to circumvent the limitation? Any thoughts about the main conclusions regarding I/O-bound vs compute-bound benchmarks?
>
> ”

Question 17 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The limitation of tasks in pure Python and on the constraint to workers behave in a single-threaded fashion, … may have a high overhead. Any thoughts on that overhead and when it is adequate to circumvent the limitation? Any thoughts about the main conclusions regarding I/O-bound vs compute-bound benchmarks?
>
> ”

- Each worker introduces scheduling and communication overhead

Question 17 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The limitation of tasks in pure Python and on the constraint to workers behave in a single-threaded fashion, … may have a high overhead. Any thoughts on that overhead and when it is adequate to circumvent the limitation? Any thoughts about the main conclusions regarding I/O-bound vs compute-bound benchmarks?
>
> ”

- Each worker introduces scheduling and communication overhead
- 'Time spent in Python' vs 'Time spent in C/C++ or I/O or other program'

Question 17 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The limitation of tasks in pure Python and on the constraint to workers behave in a single-threaded fashion, … may have a high overhead. Any thoughts on that overhead and when it is adequate to circumvent the limitation? Any thoughts about the main conclusions regarding I/O-bound vs compute-bound benchmarks?
>
> ”

- Each worker introduces scheduling and communication overhead
- 'Time spent in Python' vs 'Time spent in C/C++ or I/O or other program'
- Python 3.13+ has support for running without GIL

Question 18 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Fig. 6.2 and 6.3 (page 86) show the overhead per task when increasing the number of tasks and using two benchmarks (“merge” and “merge-25000”). Are the results similar to other benchmarks? Why the use of these benchmarks and the ones in Fig. 6.4 and 6.5 (page 87) for studying the scaling of DASK and the effect of GIL?
> ”

Question

Cardoso)

“

Fig. 6.2 and 6.3
of tasks and u
similar to othe
Fig. 6.4 and 6.5

ng the number
e the results
the ones in
fect of GIL?

”

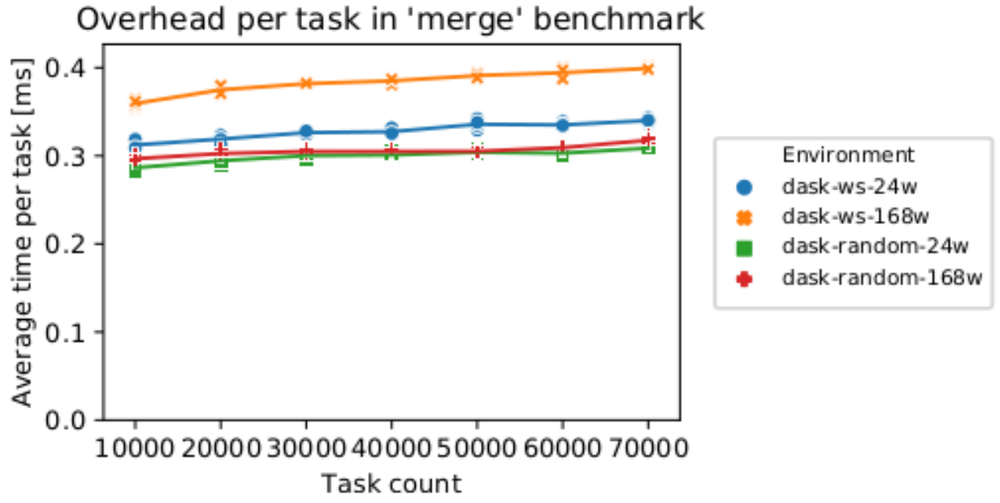


Figure 6.2: Overhead per task in DASK with an increasing number of tasks.

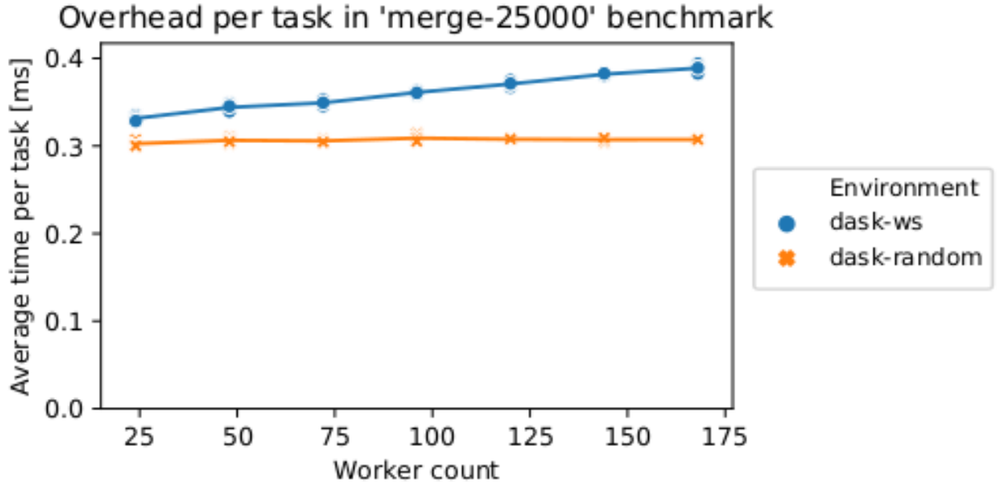
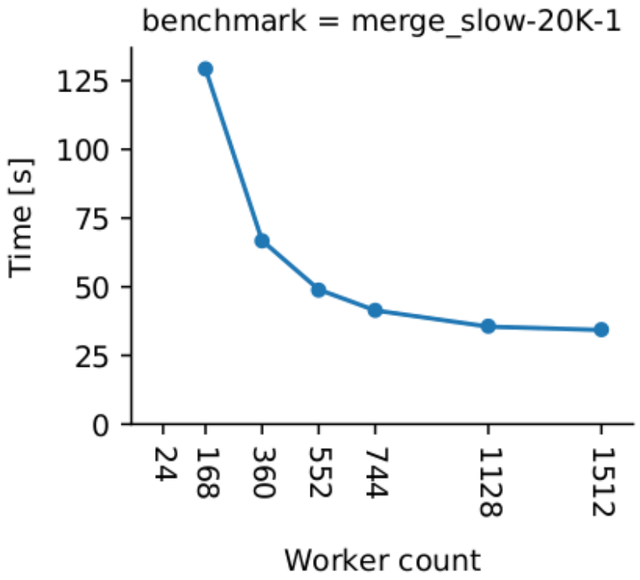
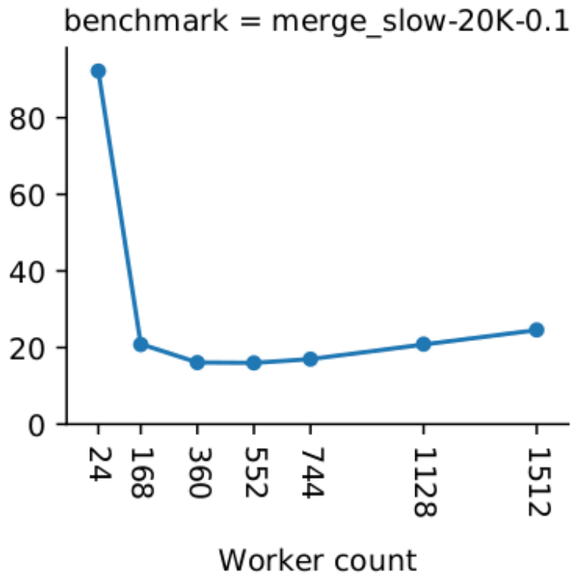


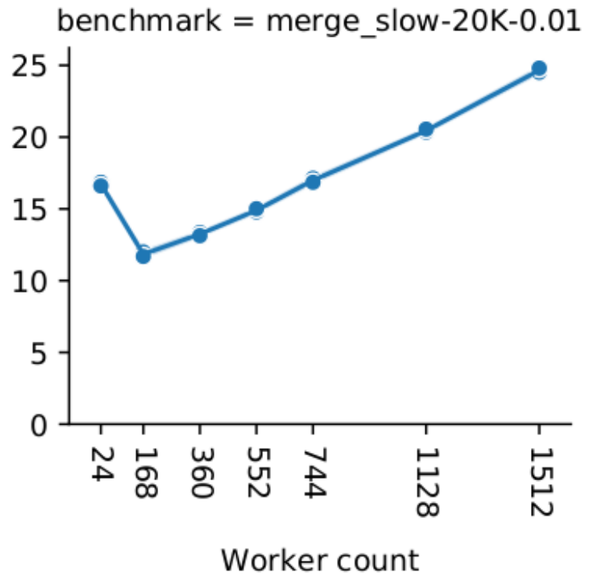
Figure 6.3: Overhead per task in DASK with an increasing number of workers.

Question 18 (assoc. prof. João Manuel Paiva Cardoso)

“Fig
of t
sim
Fig







ber
ts
n
”

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

Question 18 (assoc. prof. João Manuel Paiva Cardoso)

“
Fig
of t
sim
Fig
ber
ts
n
”

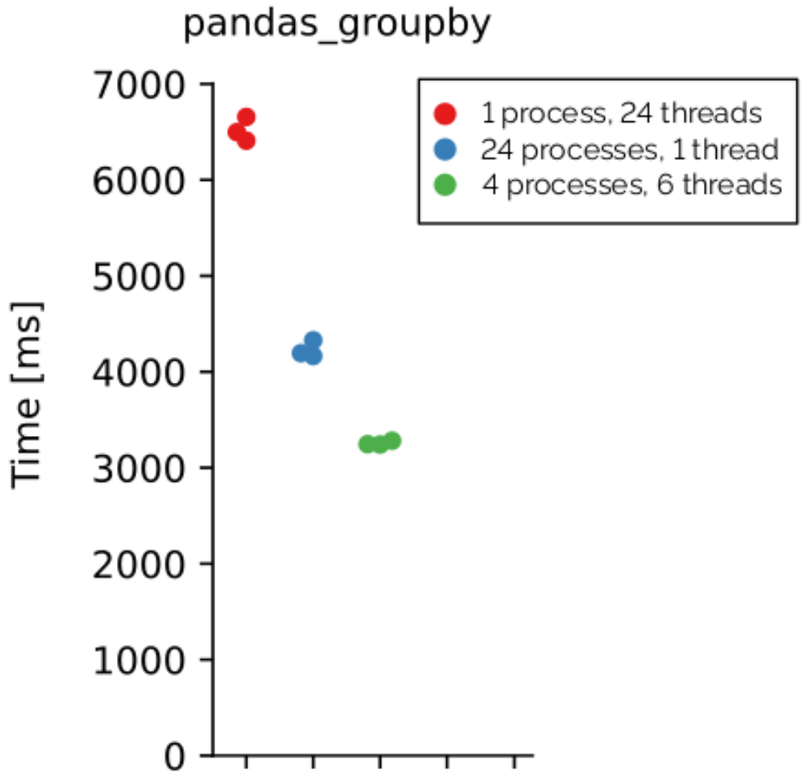


Figure 6.5: Effect of GIL on the performance of the *pandas_groupby* benchmark

Question 19 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The manual tuning of worker configuration (impact of worker per core and the non-optimal scalability of DASK) for some workflows seems very important to improve performance. Any idea if this can be statically determined?
> ”

Question 19 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The manual tuning of worker configuration (impact of worker per core and the non-optimal scalability of DASK) for some workflows seems very important to improve performance. Any idea if this can be statically determined?
> ”

- Very difficult to determine, can differ task by task

Question 19 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The manual tuning of worker configuration (impact of worker per core and the non-optimal scalability of DASK) for some workflows seems very important to improve performance. Any idea if this can be statically determined?
> ”

- Very difficult to determine, can differ task by task
- Affects the configuration of the cluster

Question 20 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Any thoughts about providing schedulers the worker network connections? Do you see a relevant impact on RSDS?
> ”

Question 21 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Since RSDS does not implement all DASK message types, do you see in future plans their integration to RSDS, or are there technical challenges that do not justify their integration?
> ”

Question 21 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Since RSDS does not implement all DASK message types, do you see in future plans their integration to RSDS, or are there technical challenges that do not justify their integration?
> ”

- Some messages are Python specific

Question 21 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Since RSDS does not implement all DASK message types, do you see in future plans their integration to RSDS, or are there technical challenges that do not justify their integration?
> ”

- Some messages are Python specific
- 'Run this Python code on the server'

Question 22 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Why do the schedulers (ws and random) integrated in RSDS do not include a list-scheduling or an ALAP scheduling scheme?
> ”

Question 22 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Why do the schedulers (ws and random) integrated in RSDS do not include a list-scheduling or an ALAP scheduling scheme?
> ”

- Work-stealing is list scheduling (B-level) + balancing between workers

Question 22 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Why do the schedulers (ws and random) integrated in RSDS do not include a list-scheduling or an ALAP scheduling scheme?
> ”

- Work-stealing is list scheduling (B-level) + balancing between workers
- We focused on runtime optimization

Question 23 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The conclusion that the improved performance of RSDS with ws is caused by better runtime efficiency and not by better schedules seems to result from the large overhead of the DASK implementation. (page 93)
> Could you comment on this?
> ”

Question 23 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The conclusion that the improved performance of RSDS with ws is caused by better runtime efficiency and not by better schedules seems to result from the large overhead of the DASK implementation. (page 93)
> Could you comment on this?
> ”

- RSDS/ws vs Dask/ws: 1.66x

Question 23 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The conclusion that the improved performance of RSDS with ws is caused by better runtime efficiency and not by better schedules seems to result from the large overhead of the DASK implementation. (page 93)
> Could you comment on this?
> ”

- RSDS/ws vs Dask/ws: 1.66x
- Dask/random vs Dask/ws: 0.95x

Question 23 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The conclusion that the improved performance of RSDS with ws is caused by better runtime efficiency and not by better schedules seems to result from the large overhead of the DASK implementation. (page 93)
> Could you comment on this?
> ”

- RSDS/ws vs Dask/ws: 1.66x
- Dask/random vs Dask/ws: 0.95x
- RSDS/random vs Dask/ws: 1.41x

Question 24 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The results presented in Fig. 6.11, page 96, seem to indicate a greater dependence of DASK to #workers (its “performance is reduced significantly with each additional worker”) than RSDS. Could you elaborate on that? Could part of the overhead reduction of RSDS over DASK, besides the implementation used, be because of the much simpler ws scheduling scheme used by RSDS when compared to DASK?
>
> ”

Question 24 (assoc. prof. João Manuel Paiva Cardoso)

“

The results pr
of DASK to #w
additional wor
the overhead
be because of
compared to D

r dependence
ach
art of
ation used,
DS when

”

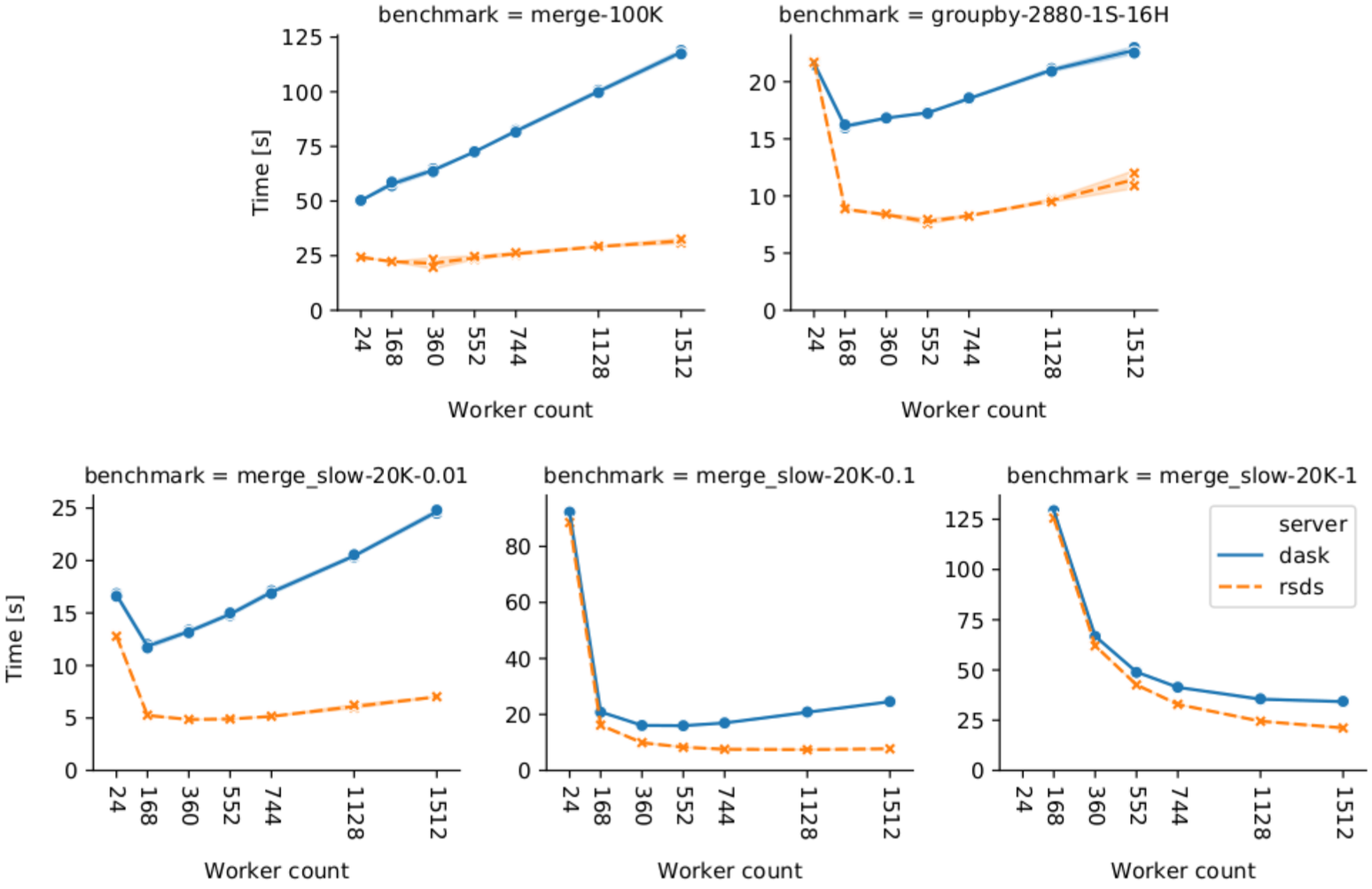


Figure 6.11: Strong scaling of RSDS vs Dask on selected task graphs

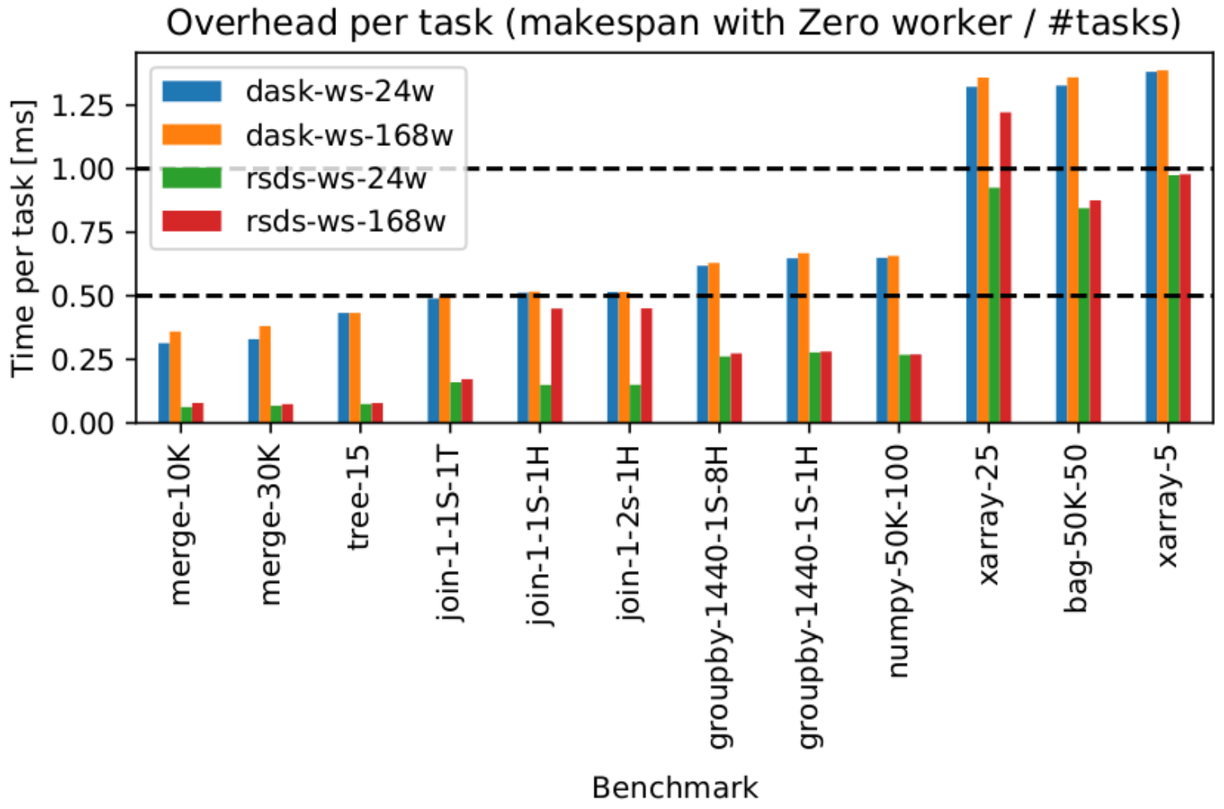
Question 25 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Do the overheads of DASK and RSDS for each task depend on the number of workers and tasks in the task graph? What is the reason for the significant overhead increase in RSDS with some of the benchmarks (Fig. 6.14, page 98)?
> ”

Question 25 (assoc. prof. João Manuel Paiva Cardoso)

“

Do the over
of workers a
overhead ir



number
significant
4, page 98)?

”

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

Question 26 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The thesis mentions that task graphs are represented in a compressed format, but that format is not fully described. Could you elaborate on this?
> ”

Question 26 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The thesis mentions that task graphs are represented in a compressed format, but that format is not fully described. Could you elaborate on this?
> ”

- Representation of a task x number of tasks

Question 26 (assoc. prof. João Manuel Paiva Cardoso)

> “The thesis mentions that task graphs are represented in a compressed format, but that format is not fully described. Could you elaborate on this?”

- Representation of a task x number of tasks
- Currently used only for task arrays (DAGs without dependencies)

Question 26 (assoc. prof. João Manuel Paiva Cardoso)

> “
> The thesis mentions that task graphs are represented in a compressed format, but that format is not fully described. Could you elaborate on this?
> ”

- Representation of a task x number of tasks
- Currently used only for task arrays (DAGs without dependencies)
- Could be extended to support also task dependencies (Dask does this)

Question 27 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Why the fractional resource requirements in HYPERQUEUE are not defined as integer percentages and need to use fixed-point representations?
> ”

Question 27 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Why the fractional resource requirements in HYPERQUEUE are not defined as integer percentages and need to use fixed-point representations?
> ”

- Fixed-point arithmetic is essentially integer percentages

Question 27 (assoc. prof. João Manuel Paiva Cardoso)

> “
> Why the fractional resource requirements in HYPERQUEUE are not defined as integer percentages and need to use fixed-point representations?
> ”

- Fixed-point arithmetic is essentially integer percentages
- Floating point operations have precision problems

Question 28 (assoc. prof. João Manuel Paiva Cardoso)

> “
> It is interesting that HYPERQUEUE showed unnecessary tuning on the number of used nodes and workers, and number of tasks and task durations, as opposite of DASK has shown. What is the main reason for this?
> ”

Question 28 (assoc. prof. João Manuel Paiva Cardoso)

> “
> It is interesting that HYPERQUEUE showed unnecessary tuning on the number of used nodes and workers, and number of tasks and task durations, as opposite of DASK has shown. What is the main reason for this?
> ”

- No GIL

Question 28 (assoc. prof. João Manuel Paiva Cardoso)

> “
> It is interesting that HYPERQUEUE showed unnecessary tuning on the number of used nodes and workers, and number of tasks and task durations, as opposite of DASK has shown. What is the main reason for this?
> ”

- No GIL
- Single node = single worker, which manages all node resources

Question 29 (assoc. prof. João Manuel Paiva Cardoso)

> “
> How do you see the possible support of HYPERQUEUE to data transfers directly between tasks? And to streaming and task pipelining support?
> ”

Question 29 (assoc. prof. João Manuel Paiva Cardoso)

> “How do you see the possible support of HYPERQUEUE to data transfers directly between tasks? And to streaming and task pipelining support?”

- This is currently work-in-progress

Question 29 (assoc. prof. João Manuel Paiva Cardoso)

> “How do you see the possible support of HYPERQUEUE to data transfers directly between tasks? And to streaming and task pipelining support?”

- This is currently work-in-progress
- Need to figure out task completion semantics and fault tolerance