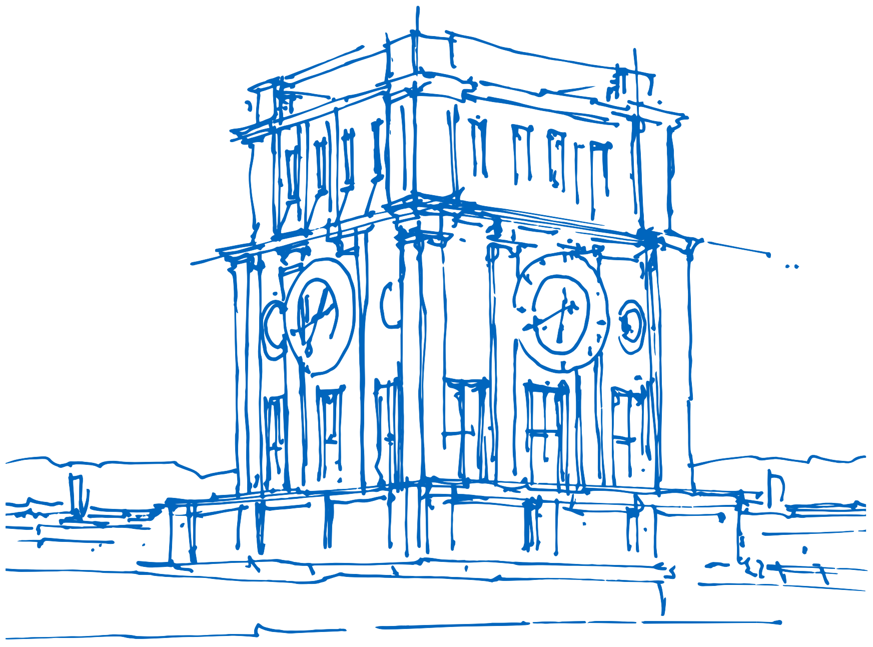
Informatics 10 - Chair of Computer Architecture and Parallel Systems
TUM School of Computation, Information and Technology
Technical University of Munich

Seminar: Scheduling – Modern Problems in a Seemingly Solved Discipline

Seminar Proceedings – Sommersemester 2022 (IN0014,IN2107)

Dr. Matthias Maiterth
Dr. Eishi Arima
Dr. Isaias Compres

Head of Chair: Prof. Dr. Martin Schulz



Seminar: Scheduling – Modern Problems in a Seemingly Solved Discipline

Seminar Proceedings – Sommersemester 2022 (IN0014,IN2107)

Dr. Matthias Maiterth
Dr. Eishi Arima
Dr. Isaias Compres

Head of Chair: Prof. Dr. Martin Schulz

Preface

"And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.""
Linus Torvalds, 2001[1]

In high performance computing (HPC), scheduling is a central aspect for efficient processing of applications and user workloads. This seminar looks at traditional aspects of job or batch scheduling and analyizes modern takes and solutions for this long standing (and seeminlgy solved) problem.
Aspects discussed in the seminar are:

- Traditional scheduling — FCFS, SJF, backfilling
- Job and batch schedulers in HPC
- Usage and scheduling of containers in HPC
- Workflow scheduling
- Additional resources: Energy as resource, quantum and other domain specific resources
- Simulators for schedulers
- Task and process scheduling

The students selected materials for the presentation, provided a report, and gave a presentation.

This proceeding represents the submitted seminar papers of the students.

[1] https://lkml.org/lkml/2001/12/15/32

Table of Contents

Metrics and Fundamentals of Scheduling

Ece Eroğlu
Bachelor's Degree Program (Computer Science)
Technische Universität München
ge59gib@tum.de

***Abstract*—Scheduling is the action of distributing the available computing resources that a system has to the tasks that require them. There are different scheduling types like process-, I/O, GPU-, and (parallel) job scheduling. This paper focuses on parallel job scheduling. There are a lot of algorithms and strategies available for parallel job scheduling, but depending on what the main goal that is aimed to be achieved for the scheduler is and what the conditions of the environment are, there are different aspects and metrics that one should keep in mind and base the decision-making on. This paper explains the fundamentals of parallel job scheduling and the algorithms that are available, different concepts for the scheduler as a goal, like cost reduction (in this paper by means of time) and fairness, and what kinds of metrics to consider when picking strategies according to this goal. Other than that, different scheduling strategies are compared to each other with the help of visual graphs, so that one can see how they do in different aspects.**

***Index Terms*—metrics, algorithms, time, fairness**

I. Introduction

In standard computers (uniprocessor systems) the tasks are performed one by one, so a task is processed when the one before is done. However in high performance computing (HPC), all the available compute nodes of the system are used to process as many tasks as possible in one. It is mainly used for the automation of large, non-interactive batch jobs. Today, HPC made it possible running huge numbers of jobs with such high speeds that was probably even hard to imagine just a few decades ago. We can see how much development has been made throughout the history of batch processing, from punch-card programmed computers to the supercomputers of today. Different strategies of parallel job scheduling can be used in these systems while keeping in mind the different goals that are prioritized.

The second section of this paper explains how parallel job scheduling works. The third section at first introduces cost metrics to consider while determining the costs of the scheduling which helps to optimize the system by reducing the time jobs require to be scheduled, then there is an example of how to use these metrics in an example scenario, and then some different metrics to determine the fairness of a scheduler. In the fourth section one can see some algorithms that are used in parallel job scheduling and how they compare to each other time- and fairness-wise. In the 5th section the concept of gang scheduling is elaborated on, which can be an alternative for other parallel job scheduling strategies. After that there is a discussion section where some concepts that have been mentioned before are evaluated, and lastly one can find a summary of the presented information and an outlook of the paper.

II. Parallel Job Scheduling

In parallel job scheduling, the system is made up of partitions with various numbers of processors. Several queues of jobs are created, each corresponding to a different combination of job characteristics, and they can have different priorities. For example, one queue might correspond to jobs that need around 32 processors, and are expected to run for a maximum of 15 minutes. Each partition gets associated with one or multiple queues, and its processors serve as a pool for these queues for the jobs to be executed on. When some processors in the partition are free, the associated queues are searched in order of priority for one that is not empty. The first job that is found that fits in the available processors is given the resources and it runs until it completes. Jobs are processed in first come first serve order within each queue. [1]

III. Metrics

A. Cost Metrics

When making strategic decisions while scheduling it is important to consider how much the whole process would cost. Costs of a scheduler can normally be observed in different categories like energy, money etc. but this paper will focus on the time aspect and introduce cost metrics that aim to reduce the time jobs require to be scheduled. There are some metrics to help determine these costs. Before looking at the metrics one should be familiar with these notations:

- t_i = completion time of job i
- s_i = release time of job i
- w_i = weight of job i
- d_i = deadline of job i
- τ = set of the scheduled jobs

The completion time is the time when the system completes working on this job. The release time is the earliest time the system can start working on job i. The weight of a job is a way of giving jobs different levels of priorities, and the deadline is the time in which a job should be done with its execution. Now, keeping in mind the notations from before, the actual cost metrics can be seen below:

- $\max_{i \in \tau} t_i$ = makespan (throughput)
- $|i \in \tau | t_i > d_i|$ = deadline misses

- $\sum_{i\in\tau} w_i t_i =$ weighted completion time
- $\sum_{i\in\tau} w_i(t_i - s_i) =$ weighted flow (response) time
- $\sum_{i\in\tau} w_i \max(0,(t_i - d_i)) =$ weighted tardiness

To have a good scheduling strategy, using different metrics for different times of the day might be necessary in most systems as the conditions can differ. For example, in daytime there are more jobs submitted by users and they mostly wait for the completion of their jobs in the daytime, so aiming to reduce the weighted completion/response time of the jobs by prioritizing the shorter ones would be a fitting strategy so that the user satisfaction is maximized and costs are reduced. On the other hand at night, when the jobs being submitted are less, considering makespan and processing longer jobs is better for maximizing the utilization of the system and reducing the response times of the jobs in total, as this way only a fewer number of long jobs would be delayed instead of a bigger number of small jobs. In real time systems where jobs have certain deadlines it is necessary to pay attention to the deadline misses and weighted tardiness. [1]

B. *Example Using Cost Metrics*

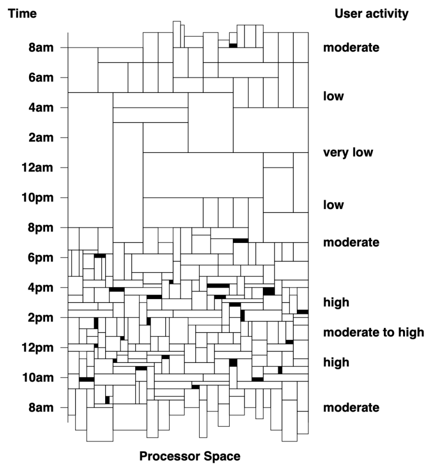


Fig. 1. Example of a processor space

Let us say a user submits a 3 hour long job at 9am. They would normally expect the job to be done around lunchtime. If it is completed later it could cause user dissatisfaction and increase costs by means of tardiness scheduling. However it is still better in the long run if the release time of their job is postponed to the evening, because as it can be seen in figure 1, the user activity is much higher during the day compared to the evening. That means it is a good strategy to prioritize shorter jobs so that the shortest weighted response time is achieved, and postpone the longer batch jobs, like which the user submitted, to the night where user activity is less. Thus the costs are reduced by getting as much jobs done as possible with the best use of the time and the available processors. [1]

C. *Fairness Metrics*

When choosing a strategy in scheduling, reducing the costs is the main goal most of the time. But only considering that can be misleading in some cases due to selective job starvation. A good scheduler should be efficient but also fair to the jobs. The concept of fairness is about distributing the resources of a system amongst the jobs as equally as possible. There are different metrics to consider for fairness. First metric that can be used is dispersion, which is higher when the scheduler favors some jobs over the others. It can be measured with the standard deviation, variance or the coefficient of variation of the waiting time a job in the queue has to the average waiting time. Another metric is the fair start time, which states a job is not treated fairly if it is delayed by other jobs later in the queue. For example, there is a job J_i in a queue. The actual start time t_i is the time J_i starts being processed. The fair start time f_i is the time J_i would start processing if it was the only job in the queue. If $t_i > f_i$, then J_i is considered to be treated unfairly. Another metric is the resource allocation queuing fairness measure $(RAQFM)$, in which all jobs are allowed to have an equal part of the resources available. If there are $N(t)$ jobs available at a time t, then each job can have $\frac{1}{N(t)}$ of these resources. [4]

IV. Algorithms

There are various algorithms available in practice for parallel job scheduling. First there is first come first serve $(FCFS)$ [2], which is about assigning resources to the jobs in the order that they arrive in the queue. The jobs state the number of processors they need and if there are enough processors available to run the job at the beginning, the processors are assigned to it and the job starts running. This is the most naive approach and not as performant as some of the other algorithms available. Another option is shortest job first (SJF) [3], which always assigns the resources to the job with the shortest execution time in the queue. It is time-wise more optimized than FCFS as it reduces the response and waiting times of the jobs. But it could also cause longer jobs to starve, hence it is not really a fair strategy. It can especially be found in systems with time slicing or gang scheduling, but the details of that can be found in the 5th section.

Another algorithm is backfilling, which is one of the most commonly-used strategies in practice. Backfilling is an optimization to the FCFS order. In FCFS if the system does not have enough resources for the job at the head of the queue to run, then it waits until some processors are freed and then the first job starts running. But in backfilling, while the first job is waiting, other jobs in the back of the queue can be scheduled instead if they require less processors and there is currently enough processors available for them to run, especially if the first job in the queue is not delayed because of this action. This way the processors that would normally remain idle are used, which means it increases system utilization and reduces the time spent in total. [2]

A. Comparison

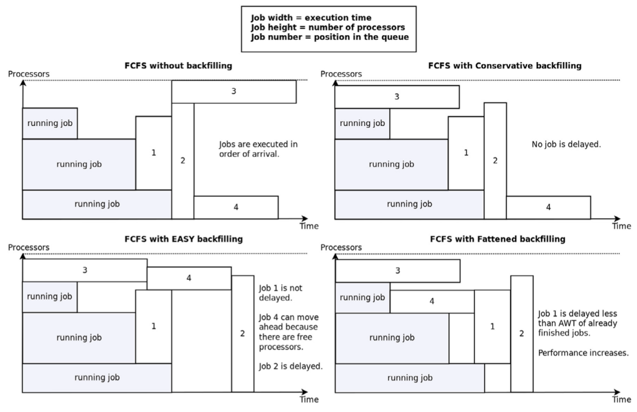

Job width = execution time
Job height = number of processors
Job number = position in the queue
FCFS without backfilling
FCFS with Conservative backfilling
FCFS with EASY backfilling
FCFS with Fattened backfilling
Processors
Time
running job
Jobs are executed in order of arrival.
No job is delayed.
Job 1 is not delayed.
Job 4 can move ahead because there are free processors.
Job 2 is delayed.
Job 1 is delayed less than AWT of already finished jobs.
Performance increases.

Fig. 2. FCFS and backfilling algorithms

Backfilling itself can also be divided into different types. The first algorithm is conservative backfilling, which moves the shorter jobs forward only if none of the jobs before it are delayed because of this action. In the second visual of figure 2, one can see how conservative backfilling works compared to pure FCFS. In FCFS, as there are not enough processors available for the first job, the system just waits until there is enough and then starts with the first job. But in conservative backfilling, the third job in the queue is scheduled ahead of the first one, as the system has enough processors for it and this action does not affect the waiting times of the jobs in the queue before the third one. Secondly, there is EASY Backfilling, which allows shorter jobs to move forward if they do not delay the first job of the queue, but the rest of the jobs may be delayed, so it is more flexible than conservative. One can see how it works in the third visual of figure 2. The third job is scheduled first just like in conservative, but then after the first job, the fourth job is scheduled ahead of the second one even though it delays it. But the first job is not delayed and that is the only requirement. Then there is fattened backfilling, which is actually just a proposed algorithm believed to be more efficient than the algorithms stated before. Its requirement is not delaying the first job in line for more than the average waiting time (AWT) of the already finished jobs. AWT is another possible cost metric which is the average amount of time the jobs wait in the queue before they are assigned to processors. In the last visual of figure 2, one can see that just like the other algorithms, the third job is scheduled first again. But now also the fourth job is scheduled before the first one, as the delay is not a problem anymore as long as it is shorter than the AWT. From the visuals these algorithms can be compared by the amount of time they require as FCFS $>$ Conservative $>$ EASY $>$ Fattened. In other words, fattened backfilling is the most cost-efficient one time-wise while FCFS is the least. [3]

Keeping the aforementioned fairness metrics in mind, comparison of different algorithms according to both the time the jobs need and fairness is now possible, and one can see how time and fairness relate to each other. In figures 3 and 4, an experimental result of comparison between three algorithms, FCFS, Fit Processors First Served $(FPFS)$ and Greedy can be seen. FPFS is an algorithm in which jobs are queued like FCFS, but if the system does not have enough processors for the job in the start of the queue, then the first job later in the queue that does not require more than the available ones is scheduled instead. But, so that the first job in the queue does not starve, jobs later in the queue can jump over the first job for only a limited number of *maxJumps* times. Greedy is also similar to FPFS, but each job has a priority indicator. When the first job cannot be processed, the highest priority job from the first *depth* number of jobs in the queue that does not require more than the available processors is scheduled, and just like FPFS, there is a limited number of *maxJumps*. These algorithms mentioned have been compared by both the time jobs need and fairness and the results can be seen in figures 3 and 4. For FPFS and Greedy there are different versions with different parameters, $FPFS(x)$ with $x =$ *maxJumps*, and $Greedy(x, y)$ with with $x =$ *maxJumps* and $y =$ *depth*. For time measurement, the average response time (ART) metric is used, which is like weighted response time but the per job value instead of the sum. For fairness, the dispersion metric with the parameters of standard deviation is used in the second graph, and coefficient of variation in the first one. In both graphs, it can be seen that Greedy and FPFS are more efficient than FCFS as the jobs have shorter ARTs there. But when it comes to fairness, the graphs have different results for the different parameters of Greedy and FPFS for standard deviation and coefficient of variation. Even with those differences, we can see that FCFS is fairer compared to Greedy and FPFS, which makes sense as in FCFS all jobs are handled in the order they arrive but in Greedy and FPFS some jobs in the back of the queue can delay the first job for a bit even if not more than *maxJumps* times. [4]

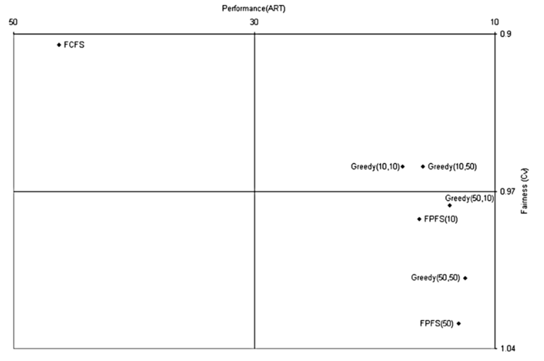


Fig. 3. comparison with metric coefficient of variation

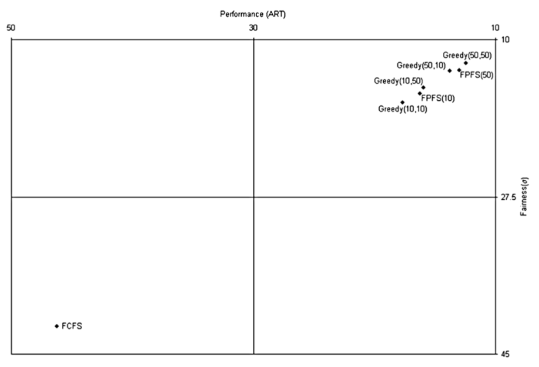


Fig. 4. comparison with metric standard deviation

V. Gang Scheduling

Besides the algorithms mentioned above, different types of scheduling algorithms based on time slicing and space slicing have been proposed [1]. In space slicing, each partition in the system is allocated to a single job, so the jobs share the processor space available like the name suggests and in time slicing, multiple jobs run on one partition with their own time slices, so they share the CPU-time. Instead of these two, in practice mostly a combination of these strategies, which is gang scheduling, is used more often. Other than batch processing strategies like backfilling, gang scheduling is the main alternative. In gang scheduling, jobs are preempted and rescheduled as a unit, so the conditions of a dedicated machine are created, where the threads of a job are scheduled together and they share the resources in a partition [2].

In time slicing and gang scheduling, preemption is used often for reducing the waiting/response times by approaching the SJF strategy [1]. It is helpful especially in cases where runtimes are unknown [2] or there is high variability when it comes to job lengths [1], so that short jobs do not have to wait in the queue for longer jobs. Another reason for using preemption is allowing computation and I/O to work together [1]. In gang scheduling, if jobs that are CPU bound and I/O bound are paired together in a partition, then while some jobs perform I/O the rest can keep computing, thus the resource utilization can be improved and processors that would remain idle in I/O time are made use of [2]. Besides that, preemption is also helpful for dividing the resources amongst the competing jobs, and also can be used to reduce fragmentation, as it is not always necessary anymore to gather idle processors a long job needs all at once [1].

VI. Discussion

So far the two aspects that can be the main goal of a scheduler, cost reduction by means of time, and fairness were explained. Even though fairness has been elaborated on here, in practice the main goal is mostly concentrated on the costs. That is the way the maximal efficiency can be achieved. Besides that, cost metrics are more objective as they give more certain results. On the contrary, in fairness the measures do not always give consistent results and it is a more relative aspect. Different ways of measurements can give counter-intuitive results. The metrics can falsely express unfairness even when there is no actual discrimination or starvation, and vice versa. In the 4th section where algorithms were compared for both the response time of the jobs and fairness of the algorithms, it was seen that measuring fairness is actually harder than time and can vary in different situations, just like how the metrics of standard deviation and coefficient of variation gave different results for dispersion. Also the two aspects are one can say inversely correlated, so when an algorithm is fairer, then the jobs would most likely need on average longer times before completion. That is why aiming to reduce the time required is paid attention to more in practice, as it is important to keep the costs low and user satisfaction high. [4]

Besides that, out of the algorithms that were mentioned in the paper, backfilling strategies are more commonly used for batch processing. We have already seen that they perform better compared to other algorithms. Gang scheduling is an alternative to the batch processing strategies, but it is actually used more in fast personal computers than in HPC. As gang scheduling requires the context switches to be synchronized across the nodes of the machine, the implementation of it on large machines can be expensive. But recent developments in experimental systems show that the overheads caused by synchronization and coordination can be reduced. [2]

VII. Conclusion

In this paper, the current algorithms and strategies used in practice for parallel job scheduling like FCFS, SJF, Backfilling, Gang Scheduling etc. have been explained and they have been evaluated when it comes to different aspects like time and fairness with the metrics that were introduced. It was observed how different algorithms do better in different scenarios and how time optimization is a more focused on concept than fairness in practice, as it is aimed to keep the costs low and increase the efficiency as much as possible. Besides the current popular strategies, some other proposed ideas have been elaborated on, like for example fattened backfilling, which shows that even though a lot of advancement has been made in the parallel job scheduling area throughout history, it is still actively researched as improvement in different aspects is still possible.

References

[1] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik and Parkson Wong, ”Theory and Practice in Parallel Job Scheduling”, pp.2–15

[2] Dror Feitelson, Larry Rudolph, Uwe Schwiegelshohn, ”Job Scheduling Strategies for Parallel Processing”, Springer-Verlag Berlin Heidelberg 2005, pp.2–11

[3] César Gómez-Martín, Miguel A. Vega-Rodríguez, José-Luis González-Sánchez, ”Fattened Backfilling”, J. Parallel Distrib. Comput. 97 (2016) pp.69–71

[4] J. Ngubiri, M. van Vliet, ”Characteristics of Fairness Metrics and their Effect on Perceived Scheduler Effectiveness”, International Journal of Computers and Applications, 32:2, pp. 188–193

GPU Scheduling (Warps and Compute Units)

Hopperdietzel Stephan
Bachelor's Degree Program (Computer Science)
Technische Universität München
ge25zot@mytum.de

Abstract—**General Purpose GPU computing became more relevant in recenent years due to superior multithreading capabilities compared to CPUs. As they were not initialy designed for general purpose applications there is a lot of optimization still left for this use case. To make better use of the existing hardware pipeline the scheduler can be improved.**
This paper presents optimizations to the scheduler showing there is a lot of performance still to be gained within GPUs for scientific computing. There are 5 proposed solutions presented, which deal with improving the utilization of existing hardware and preventing stalls inside the pipeline. Simulations show that the improvements have a performance gain between 8-41% on average and can double for specific applications.

I. Introduction

Using the GPU as a general purpose accelerator has become more popular in recent years. GPUs provide increased performance, energie effienency and cost compared to CPUs in certain tasks. [1]
This is due to the increased area dedicated to computation on the chip compared to the area designated to control, as GPUs are designed for Single Instruction Multiple Data/Thread use in each core. x86 CPUs can do SIMD with some extensions (e.g. AVX), but those are not prioritized in CPUs and still have a much smaller SIMD width to GPUs resulting in less data per instruction.
GPUs can execute many threads of the same program concurrently. When a high number of threads execute the same code a GPU is utilized best. This includes applications such as fluid simulation, databases, climate prediction and many more, which utilize APIs like OpenCL, CUDA or CTA to execute C code on GPUs. [2]

II. Background

A. Pipeline

GPUS were designed to compute and display the graphics of a computers, which incldues computing shaders, a heavily multithreaded workload. Therefore GPU architecture is vastly different to CPUs. [2]
Instead of multiple cores like a CPU GPUs consists of multipe *Streamline Multiprocessors* (SM/Compute Unit). Each contains multiple ALUs that execute the same instruction concurrently. To use this, threads of the same program are grouped together and get executed in *lockstep*, meaning they share a common program counter and go throught the same fetch-decode-execute cycle but with their own data in diffrent ALUs at the same time.
The fetch-decode-execute cycle is slow and can be parallized, might leading to one instruction being loaded and others being decoded, computed or waiting for memory all at the same time. This is called a pipeline. “Fig. 1”
Starting at the top left is the *warp list* storing the warps contained in the SM. A *warp* is the aformentioned group of threads. The number of threads in each warps is identical to the number of ALUs in a SM so that it can be executed in one cycle. This is typically 32/64 threads inside a warp depending on the model and manufacturer. The *warp list* stores id, active mask and program counter for all warps in the SM. The *active mask* describes which ALUs are currently occupied in the warp, meaning if the nth bit is 0 the ALUN is not used when the warp gets executed.
To begin a cycle the scheduler picks an available warp in a round robin way, ensuring all warps have the same priority and progress at the same rate. In the pipeline the instruction of the warp is fetched and decoded next.
Following this the registers of the threads are loaded and put into the lanes for execution. These are stored in a *Register File* where each row corresponds to a warp and the cells are all registers of a thread. In the lane they get processed by ALUs. During the execution threads can access memory. This can be main or private memory. Latter is further storage for data associated with a specific thread.
After calculation registers are wrote back into the register file, the program counter and active mask are updated in the warp list and the warp can be scheduled again.

B. Branching

The concept of running multiple threads is very effective, but does not take diverging control flow into account. At conditional jumps threads inside a warp may choose different paths in the control flow, resulting in mutliple different program counters for the warp.
To solve this a *divergence stack* is used “Fig. 2”. When reaching a junction in the control flow (in the example A) the scheduler chooses one of the pathes to execute first (in this case B) and creates two entries in the divergence stack. First a *Join Entry* which represents where the pathes meet each other again. In the example the control flow merges at D so it is put as the *Recovery Program Counter* (Rec PC). Since all threads started at A the active mask of the join entry is all 1 and the program counter to execute again is D. The other entry to be pushed onto the stack is the *Divergence Entry*. It represents

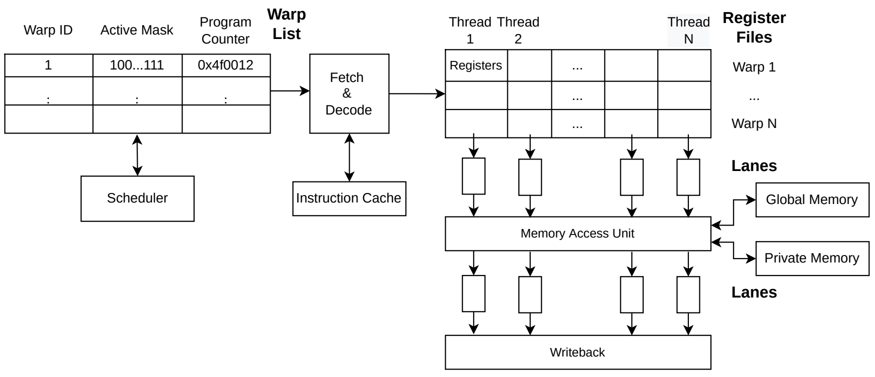


Fig. 1. GPU Pipeline

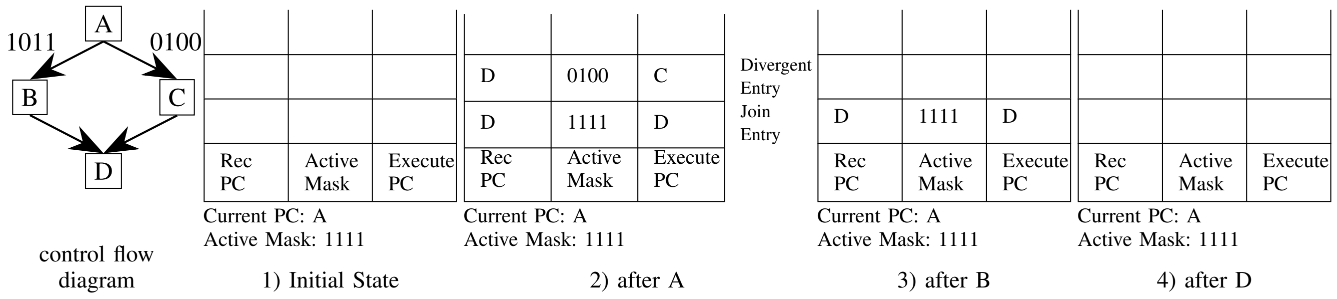


Fig. 2. Divergence Stack

the start of path that has not been taken in the control flow (in this case C). As it merges at D the Rec PC is set to D, the active mask is set to the threads which take the path and the programm counter is C as it is the start of the path.
After pushing the entries execution on the chosen branch start (state 2).
When reaching D the scheduler sees that it has entries on the divergence stack that meet there, so it pops the divergence entry of the stack and starts executing it(state 3).
After all diverging paths are taken the join entry is popped of the stack and execution continues (state 4).
If additional branching occurs inside a branch the mechanism is the same and the stack grows, with the exception that the join entry might not consist of all 1s as an active mask. As seen in the example braching leaves many lanes doing nothing.

III. Improvements

In this section varying improvements to the baseline GPU scheduler are presented. They can be sperated by the problem they solve. Large Warps, Simultaneous Branch Interweaving (SBI) and Simultaneous Warp Interweaving (SWI) try to make the most use of the existing lanes in the hardware. Two-Level Scheduling and Memory Divergence Correction (MeDiC) try to minimize the effects of stalls in the pipeline.

A. Large Warps

When the flow of a programm splits up, the active mask gets less populated, which means that some lanes of our SM are not used. To reduce the impact of branching the researchers propese the *large warp microarchitecture*. The large warp replaces the existing warps in the pipeline. They are composed $k \times n$ threads where n is the number of lanes in the SM. The amount of threads is kept the same in each SM meaning that each SM has fewer, but larger warps now.
These large warps can no longer be executed in a single step in a pipeline, since they have more threads than the SM has lanes. So the execution of a large warps takes multiple cylces and only after all threads, that are marked for execution in the active mask, are executed the programm counter and other control structures are updated and the warp is considered for scheduling again.
To make efficient use of the hardware the active mask is organized as matrix size with k rows and n columns. Latter represent the existing lanes of the hardware.
When choosing which threads to execute in each cycle the scheduler takes the first not already executed thread in each column, meaning the threads to be executed in the pipeline in a step can come from different rows. The register file has to be extended allowing registers from different rows to load at the same time.
Overall the lanes are now used more often and instead of executing the fixed amount of k warps potentialy less cycles were needed to execute the large warp with the same amount of threads. An example for this can be seen in “Fig. 3”, where each color represent threads executed at the same time. The normal warps take 8 cycles, while large warps only takes 5.

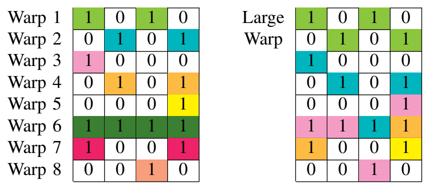


Fig. 3. normal/large warp

[3]

B. Simultaneous Branch Interweaving

SBI tries to get around the same problem of low ALU utilization due to branching by introducing a second instruction loader.
When a program encouters a point of divergence the threads inside it chooses one path to execute first. If it is not a simple if-then, but a if-then-else two or more program counters are now available for the warp to advance. As explained in II-B branching the baseline would now pick one path and execute it until reaching the join entry before exectung the other one. SBI on the other hand executes can two branches at once. It introduceds a second scheduler that loads an instruction from a different branch of the same warp than the first scheduler. The active mask of this second instruction is not allowed to interfere with the active mask of the first instruction. The ALUs now need a multiplexer that decide if it is active and if so which of the two instructions it has to execute.
Finding a second instruction to load is fairly trivial, as threads inside a warp are mutualy exclusive in one branch, resulting in active masks of different concurrent branches that do not overlap. Due to being at two points in the program, a simple divergence stack can not be used anymore. Instead a heap is used to keep track of when threads have to join and which two branches can be executed.
SBI decreases the times a lane is inactive and can reduce the number of cycles needed. It works best when the concurrent branches in the control flow have similar execution time. [4]

C. Simultaneous Warp Interweaving

SWI is similar to the idea of SBI, but instead of improving when threads have balanced workloads in different paths it improves throughput when the workloads are unblanced.
As with SBI there is a second scheduler that picks a instruction with a non overlapping active mask to the first instruction. The second instruction can be any current instruction from another warp inside the same SM.
There is now a need to load the register files from two warps in each step and decide on which register has to be put into each lane. This again is done by introducing a multiplexer. The same multiplexer as in SBI for deciding which ALU executes which instruction is also required.
As the scheduler takes longer to look for a second instruction it trade scheduler latency for better ALU utilization increasing performance overall. [4]

D. Two-Level Warp Scheduling

Inside a SM warps are executed using Round Robin. Subsequently warps tend to progress at the same rate through a given program. This is great for locality meaning caches are used effectivly. But in the same way they tend to reach long latency operations at the same time, resulting in no threads being available for scheduling.
To avoid this the warps are combined into mutliple groups. The warps inside them are still scheduled using round robin. Groups get different priority, where as long as the highest priority one has warps to schedule they get executed. Once all warps in a group stall, the groups themselves get reprioritized in a roub robin way. This is the second level of scheduling.
As a result the groups progress at different rates, so they do not reach long latency operations at the same time. When the last group reaches a long latency operation the first is likely to be finished already meaning there are always warps to be executed at any given moment. The cache locality is still maintained as inside a group the threads progress at the same speed.
[3]

E. Memory Divergence Correction

If a instruction inside a warp accesses memory it needs to wait until memory requests of all threads finished loading before continuing to execute. This means if a single thread inside the warp has a cache miss all other threads have to wait, even if all other request were cache hits.
Research has made two observations on this problem. First not all warps behave the same. Some of them have most of their request taken care of by cache (mostly-hit Warp), while others do not utilize the cache well and have most of their request miss cache (mostly-miss Warp). Second a warp tends to keep its cache Hit/Miss ratio over long periods. Combined these two observations can be exploited to improve overall performance.
The goal is to convert mostly-hit warps into all-hit warps and mostly-miss warps into all-miss warps.
This speeds up the exectution as threads inside mostly-hit warps then do not have to wait for long memory accesses and can progress faster while the cache hits inside the mostly-miss do not increase performance of its execution as it needed to wait for memory anyway.
To identify the warp type the cache hit ratio is sampled for each warp. After sampling the type of the warp, its number of cache misses and number of cache access are stored. This is done periodically to account for long term drifts.
To reach the goal of converting mostly-hit into all-hit warps modifications to the cache and memory scheduler are proposed.
As all-miss warps do not benefit from cache accesses they can bypass the cache entirely. This frees up space inside the cache which can be filled with data for mostly/all-hit warps which benefit from it. Addionaly due to decreased number of cache request the queuing delays are shorter, decreasing latency of the cache.

For a second measure keeping data from all/mostly-hit warps in the cache is more efficient than keeping data from balanced or mostly-miss warps. Therefore the insertion policy of the cache is changed. Cache blocks get associated with the type of warp that requested it. By inserting cache blocks in diffrent positions in the least recently used queue, when evicting cache blocks from all/mostly-hit warps are kept longer and others are evicted earlier.
Having more cache space available and keeping the data longer in it increases the chances of mostly-hit warps becoming all-hit warps, but does not guarantee it. For the case a mostly-hit warps still has a cache miss the memory scheduler is modified. Memory request are tagged with a bit that indicated if it comes from an mostly-hit warp. If it does so it gets higher priority in the request queue of the memory controller and therefore returns faster.
[5]

IV. Discussion

A. Hardware Cost & Performance Improvements

All of the improvements explained need additional or modified hardware to work. Since producing actual hardware is expensive and complex performance results are simulated and the hardware costs estimated.
Large Warps hardware cost is mostly the modified register file which is estimated to grow 11-18% in size. This corresponds to 2.5% of overall GPU size. Large warps excell in branch-intensive applications, increasing the performance by up to 50% in bucket sort. Over a mixed set of benchmarks large warps increase performance by 7.9%. [3]
SBI primary hardware cost comes from the second scheduler and the divergence heap, corresponding to a 3% increase in GPU size. SBI works best on irregular applications . These applications access memory locations and choose their path inside the control flow based on their data. [1] Performance for these programs is increase by 41% while the average performance increase is only 15% on regular applications. [4]
SWI has similar requirements in hardware changes as SBI. It needs a slightly modified second scheduler and the possiblity to load from a two rows in the register file, but does not need the divergence heap. This results in a 2.9% increase in size of the GPU. SBI benefits the most in irregular programs increasing performance by around 33% in these workoads. On regular apllications the performance increase is simulated at around 25% over baseline. [4]
Two-Level warp scheduling hardware cost is insignificant, only requiring minor modifcations to the scheduler. In benchmarks that suffer from long latency operations and associated stall Two Level scheduling works best. It can increase performance for example by around 50% in matrix multiplication. Overall the increase is more modest with 9.9% over a broad variety of benchmarks. [3]
Hardware cost of MeDiC is relativly low. It only needs space to store the metadata for each warp and some bits to store the warptype for each cache line. This is less than 1% of the size of the cache. In benchmarks MeDiC provides siginificant performance improvements over nearly all types of applications. In Breath-First-Search it more than doubles the performance. Overall it is up by around 41.5% over baseline. [5]

B. Combining Improvements

While the improvements work alone, some of them can be combined together to get even better results.
As improvements that tackle stalls do not interfere with ones improving overall utilization of GPU lanes, they can be combined trivialy. For example combining two-level scheduling with large warps preserve indivual improvements and therefore yield a 19.1% performance increase over baseline. [3]
SBI and SWI require very similar hardware, meaning they can be combined together without much overhead (3.7% increase in GPUS size). Since they both try optimizing the same hardware component they do not scale as well toghether as the previous example, but as they have there highest performance gains in different types of applications it is still worth to combine them. [4].

V. Conclusion

The baseline Round Robin Scheduler does not utilize the hardware well for general purpose computing on a gpu.
The two main problems of the base scheduler can be solved in different ways. Underutilizing lanes due to branching requires a lot of changes to the hardware. Optimizations for this yield fastly different performance gains based on the type of program. Stalls in the pipeline due to long latency operations or cache misses can be solved without large changes in hardware and optimize different kind of programs.
Modern GPUs are a lot more complex than shown here as baseline. They have multiple execution units of different types inside one SM. This means that the scheduler has to be more complex by design. Furthermore it is likely, that some of the optimizations presented are implemented in recent GPUs, but as manufactures closly guard the internal workings of their GPU this can not be verified.

References

[1] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, 2012, pp. 141–151.
[2] E. Wu and Y. Liu, "Emerging technology about gpgpu," in *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*, 2008, pp. 618–622.
[3] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 308–317.
[4] N. Brunie, C. Collange, and G. Diamos, "Simultaneous Branch and Warp Interweaving for Sustained GPU Performance," in *39th Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, United States, Jun. 2012, pp. 49 – 60. [Online]. Available: https://hal-ens-lyon.archives-ouvertes.fr/ensl-00649650
[5] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting inter-warp heterogeneity to improve gpgpu performance," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, ser. PACT '15. USA: IEEE Computer Society, 2015, p. 25–38. [Online]. Available: https://doi.org/10.1109/PACT.2015.38

Process Scheduling (OS)

Mario Delic
Bachelor's Degree Program (Computer Science)
Technische Universität München
m.delic@tum.de

Abstract—**Operating System schedulers have steadily improved over the past years from moderately effective scheduler with simple heuristics to nowadays strong, general purpose schedulers with variable features and extensibility. However, most of them cannot handle the architectural diversity without sacrificing at least some performance. This review aims to provide an overview of process scheduling in operating systems and recent research and improvements for task and architecture specific scheduling challenges. OS-scheduler extensions and the methods they use for dealing with a specific issue will be presented, particularly regarding optimization of memory usage and latency.**

I. Introduction

Although computers have been shrinking in size constantly, their complexity has been steadily growing. When computer sciences were at their start, there was effectively no real scheduling; a machine would work through it's instructions in a non-preemptive first come first served manner. This approach worked just fine on simple old machines designed to process one request at a time from start to finish, one after another; and it still does work perfectly for systems that function in a similar way or want to maximize throughput or minimize scheduling overhead from memory operation and CPU-idling. Today, with the diversification of computer architecture and use cases and scenarios, an operating systems scheduler has to adapt accordingly to avoid both inefficient performance in terms of time and resource utilization as well as missing the expectations of the systems (meeting deadline, avoiding starvation, etc.). And since the CPU scheduler executes an enormous amount of times in even a single second, its performance is a very relevant part of the system in general[6][7].
One prominent CPU scheduler is the current Linux Scheduler, it has experienced frequent updates and changes up until the year 2007/Kernel version 2.6 when the Completely Fair Scheduler (CFS) got introduced as the vanilla scheduler. From that point onwards, the CFS scheduler has remained the baseline scheduler in the Kernel. It offers multiple default scheduling policies and classes, trough which the behavior could be tweaked towards a better suitability for specific tasks. Additionally, the implementation supports the possibility of further extending the scheduler if needed, for example for architecture specific scheduling[7]. The research that will be presented in this document is extending the Linux Scheduler or using it as a frequent reference.
The classic scheduling heuristics and algorithms (such as Shortest Job First, Round Robin scheduling etc.) are well known, and the issue of (fairly) assigning the processes and thread in a run queue is seemingly solved. In contrast, cache and memory efficiency and varying hardware architectures are some of the new modern problems which OS-schedulers have to face. These have been the topic of a lot of recent research. This review will present the core problems addressed in selected relevant papers regarding OS scheduling with respect to hardware structure and the effects on and memory and CPU performance. Section II introduces relevant metrics and problems. Section III and IV will highlight some key heuristics and algorithms used to make decisive decisions and to optimize the behavior of the new schedulers will be highlighted. Finally, Section V evaluates and discusses the measured results.

II. Challenges

Optimizing an OS scheduler in a general manner is a hard task to do, especially with the aforementioned challenges. Instead, a few key criteria are selected which are then further examined in order to create a smaller extension which is able to better handle problems in that domain.

Core assignment and speedup factor: some processes and execution threads will receive a higher speedup on a certain core compared to other threads[1]. Scheduling threads with respect to their speedup can yield improvements, but factors such as priority, critical and blocking threads, and potential memory migration need to be taken into account in order to maximize efficiency[1][4].

Memory access: scheduling processes for execution requires their data to be loaded from memory, and upon pausing, their state to be written back to memory. This can create overhead due to additional memory operations[5]. Additionally, during execution, processes and their threads will most likely access other memory locations. This can result in the processes competing for memory and cache, and with oblivious scheduling, leads to avoidable cache misses and memory accesses which waste time and energy. If a process were to be migrated or rescheduled as a result of some policy, that migration might end up very costly, which is why memory in particular has to be handled carefully[2].

Hardware structure: different cores or interconnects between CPU units require consideration. While the Linux Scheduler already takes into account core size and interconnect distance, it's usually oblivious to any other detail for these issues[1][3]. Therefore placement of the process or thread matters, and a heuristic for searching out threads worth moving is needed for such systems. Additionally, the overhead from a possible memory migration to another node impacts the decisionmaking here as well[3].

Extension overhead: naturally, an extension it will incur overhead, as the it also needs to execute constantly. Even if the scheduling algorithm itself proves to be negligible[1], the data collection for these algorithms, as well as memory operations done due to scheduling decisions by the extension may not be. Acquiring and organizing data, which is needed for making adequate scheduling decisions, during runtime can in fact be quite demanding. These need to be kept low enough for the extension to have significant merit[2][3].

Combining all of these and possibly more problems in one effective generalist solution should prove quite difficult, which is why research mostly extends schedulers by only one module or use case. In the context of OS and process scheduling, selecting the process or thread to run is not the only relevant decision, as memory usage is highly intertwined with the resulting performance. It can be generally said however, that the current core objectives of research when it comes to process scheduling are scheduling the processes better against hardware and it's resources.

III. CACHE EFFICIENT SCHEDULING

As a process is running, relevant data that might be reused will be loaded into the cache. As the cache is very limited in size, if the process pauses execution or gets swapped out other processes might overwrite it's data in the cache. If a scheduler was aware of the cache-use behavior of a process, it could perform scheduling in a way that reduces interference and unnecessary cache waste. The scheduler needs to know when exactly the process will behave in a cache intensive way with data reuse. One approach for modeling the cache use is to use two values, working set size and reuse factor. The working set size describes how much total memory the process consumes during one period, while the latter indicates how much this memory will be reused. To keep track of the demands and changes, a user level-interface is implemented. Figure 1 shows a visualization of such an interface. It monitors the current demands and provides predicates for scheduling decisions. When a program hast sections or periods with intensive cache use (referred to as **Progress Period**), it reports those period's entry and exit point to the Progress Monitor by means of a simple function call in the program's code, with parameters being the working set size in MB and data reuse as a relative variable in the context of the other processes. By actively declaring these periods, the process will only be scheduled by the cache aware scheduler when it actually needs to. When executing outside of a progress period, the process will be scheduled by the default policy. Separating these execution periods avoids scheduling processes using unnecessarily complex method when no benefit is expected to be gained. The **Progress Monitor** then communicates the resource demand to the Resource Monitor, which tracks the load of the Last Level Cache (LLC) and the resource demands of the Progress Periods. Finally, a **Scheduling Predicate** decides whether a thread will be run or paused. For that, hardware capacity, current hardware load/usage and the demand of the new Progress Period are evaluated. Figure 2 shows the described algorithm; the policy is an alterable degree of strictness where a compromise policy would optimize for cache with a degree of care for maintaining concurrency, while a strict policy seeks to only maximize cache efficiency[2].

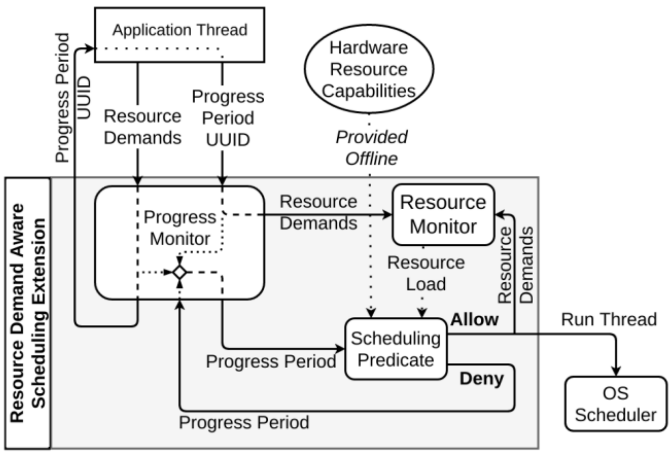


Fig. 1. Workflow of a Progress Period [2]

```
function TRYSCHEDULE($pp$, $resource$)
    $remaining \leftarrow resource.capacity - resource.usage$
    $outcome \leftarrow remaining - pp.demand$
    $runnable \leftarrow apply\_policy(outcome, resource)$
    if $runnable$ then
        $increment\_load(pp.demand)$
        $schedule(get\_process(pp))$
    else
        $waitlist(pp)$
    end if
end function
```

Fig. 2. Example algorithm for the scheduling predicate [2]

IV. CORE AND MEMORY MIGRATION

With the large variety of system architectures present, specific scheduling extensions need to be created for optimization. Just like the previous example scheduler, this one is also extends the Linux Kernel Scheduler in form of a user level process. Aside from that, the methodology is radically different originating from the aim to better schedule on NUMA (Non-Uniform Memory-Access) systems with respect to the varying interconnects between the different CPU and memory nodes. Linux for example tries to schedule threads onto the so called "home node" of the process. This policy can produce suboptimal results if the threads to place exceed the number of cores in a node, as depending on the architecture, "closer node" is not always equivalent to "better node". The challenges for scheduling optimally in this setting are again the difficulty of collecting and measuring communication during runtime and the high cost of migrating the memory if the need arises. What the extension tries to accomplish is reallocating the threads so that on the new placement, the communication for the thread, process and system as a whole is more effective, all while minimizing costly memory migrations from rescheduling. The *AsymSched* algorithm has three components. The **measurement** component steadily measures CPU access to a given node, or ideally a given CPU. This is preferably done through a service provided by the CPU manufacturer, although these are not necessarily provided with every detail so workarounds might be needed (measuring node acceses instead of CPU accesses in this case). The **decision** component calculates a variety of possible placement, filtering suboptimal ones below a specified interconnect-bandwidth out. As has been said, memory is a crucial factor when scheduling. Thus from the remaining, the one with me least memory migrations necessary is chosen. If that migration would incur too much overhead, it is dropped, otherwise the migration will be performed. The **migration** component migrates the thread or process to another node though provided syscalls from the kernel. Full process migrations with migrating their its pages are being avoided, instead only a smaller subset of pages the application uses is moved to another node at first. If the memory accesses on the old node of that application are above a certain threshold where the subset-migration is deemed insufficient (here: 90 percent), the scheduler will opt for a full migration of the process and it's pages. The the Linux syscall *migrate_pages* proved inefficient for larger working sets which is why for the purpose of extending the scheduler in this way, a new memory systemcall had to be designed[3].

Per cluster (C) statistics	
C_{rbw}	Remote bandwidth: the number of memory accesses performed by threads in the cluster to another node, i.e., *remote accesses*.
C_{weight}	"Weight" of the cluster. Clusters with the highest weights are scheduled on the nodes with the highest interconnect bandwidth. By default $C_{weight} = log(C_{rbw})$.
$C_{bw}(P)$	Maximum bandwidth of C threads on placement P.
Per placement (P) statistics	
P_{wbw}	Weighted total bandwidth of P. Is equal to the sum of the $C_{bw}(P) * C_{weight}$ for every placed cluster C.
P_{mm}	Amount of memory that has to be migrated to use this placement.
Per application (A) statistics	
A_{tm}	Time already spent migrating memory.
A_{tt}	Dynamic running time of the application.
$A_{mm}[node]$	Resident set size of the application, per node.
A_{oldA}	Percentage of memory accesses performed on nodes on which the application *was* scheduled but is no longer scheduled on.

Fig. 3. Relevant definitions for the algorithm in Fig. 4 [3]

```
1: if Threads of nodes $N_1$ and $N_2$ access a common memory controller and threads of $N_1$ and $N_2$ have the same pid then
2:     Put all threads running on $N_1$ and $N_2$ in a cluster $C$ and Increase $C_{rbw}$
3: end if
4: Compute relevant cluster placements
5: $Max_{wbw} = 0$
6: for all P $\in$ computed placements do
7:     $P_{wbw} = \sum_{C \in clusters} C_{bw}(P) * C_{weight}$
8:     $Max_{wbw} = max(Max_{wbw}, P_{wbw})$
9: end for
10: for all P $\in$ computed placements do
11:     Skip if $P_{wbw} < 90\% * Max_{wbw}$
12:     Compute $P_{mm}$
13: end for
14: Choose the placement with the lowest $P_{mm}$
15: for all A $\in$ migrated applications do
16:     if $A_{tm} + \sum_{n \in migrated\_nodes} A_{mm}[n] * 0.3 > 0.05 * A_{tt}$ then
17:         Do not change thread placement
18:     end if
19: end for
20: Migrate threads
21: Use dynamic memory migration
22: After 2 seconds:
23: for all A $\in$ migrated applications do
24:     if $A_{oldA} > 90\%$ then
25:         Fully migrate memory of A
26:     end if
27: end for
```

Fig. 4. Detailed AsySched algorithm [3]

V. Evaluation

Table 1 shows the benchmarks for a selected policy of the Resource and Demand Aware Scheduler (RDA) in comparison to the default Linux Scheduler's results. The results show that the new scheduler can achieve greatly increased performance while also making better use of the cache memory, which can be seen in the large reductions of energy consumed by DRAM. For jobs with lesser amount of memory reuse however, the scheduler does not produce a better outcome, as seen in cases water_sp and BLAS-1. Little room for optimization leads to no better cache use, as the DRAM energy consumed is nearly identical in both cases, while the overall energy consumption and performance have worsened[2]. A similar trend can be observed for other research as well. The extended scheduler would attain significantly better results for more specific or extreme cases, but it would show reduced performance compared to the default Linux Scheduler for worst case scenarios or simpler jobs[1][2][3][4]. In the case of RDA, this is due to a loss in concurrency while trying to optimize for cache use. Using a compromise policy instead of the strict policy would improve average performance for on average worse cache management; compromise performs worse than strict on cache intensive tasks, and better on the simpler tasks. But

TABLE I
PERFORMANCE ON SELECTED WORKLOADS FOR LINUX SCHEDULER AND (STRICT) RDA[1]

Operation	**Scheduler**	**Energy consumed (kilojoule)**		**CPU perf. (GFLOPs)**
		CPU+memory	only DRAM	
Raytrace[a]	Linux	3.80	0.30	20
	RDA	2.00	0.03	38
Volrend[b]	Linux	1.90	0.05	14
	RDA	1.30	0.02	17.5
BLAS-1[c]	Linux	5.00	0.40	1
	RDA	8.00	0.39	0.5
BLAS-3[d]	Linux	30.25	1.45	36
	RDA	29.00	0.67	38
Water_sp[e]	Linux	0.25	0.50	47
	RDA	0.30	0.50	31

[a]Working Set Size: 5.1MB, 5.2MB; Reuse: high
[b]Working Set Size: 1.8MB, 1.7MB; Reuse: high
[c]Working Set Size: 0.6MB; Reuse: low
[d]Working Set Size: 1.6MB, 2.4MB, 2.4MB, 3.2MB; Reuse: high
[e]Working Set Size: 1.6MB, 1.3MB, 1.3MB, 1.6MB; Reuse: low

the worse performances don't scale in the same way and the relative decreases are drastically lower than the gains by using the extended scheduler for more complex jobs, warranting their use in the end. This approach might directly involve the developer, as they need to identify the Progress Periods and be aware of the data used in order to call to the extension. But automating the process of determining Progress Periods has at least been shown to be possible by creating predictions based on the first few inputs of a Progress Period[2].

Examining the select few results in the following Fig. 5 and Fig. 6, a similar trend can be identified as with the RDA Scheduler. For a smaller subset of seemingly simpler jobs the scheduler can achieve no real improvement over the average static case, seen in bt.B or mg.C. In contrast, for larger jobs the improvement in memory latency and performance seems to scale with size, as seen with streamcluster or pcs, outperforming the best static placement. Looking at lu.B and is.D, a large discrepancy between the good results regarding memory latency compared to the respectively below average and moderate performance result is visible. This can be traced back to memory migration; in is.D's case, the improved memory latency is mitigated by the large amount of migration operations performed during execution, showing less performance gain. In lu.B's case, the issue stems from a memory intense behavior at start with barely any memory accesses afterwards. As system size increases and process scheduling is deeply intertwined with memory management in an OS, the issue of memory migration and latency will stay prevalent. In AsymSched, the majority of overhead generated is due to the memory migration, up to a huge 50% overhead using the Linux syscall (only 1.5% with the newly implemented syscall)[3]. As opposed to RDA which needs developers to declare critical sections, AsymSched relies solely the CPU's measurements.

Other research focusing on core assignment and portability instead used a machine learning and offline training approach in order to delegate some of the data collection for making scheduling decision away from runtime[1]. Finally, both extensions show great improvement for their domain, albeit with some very minor trade-offs.

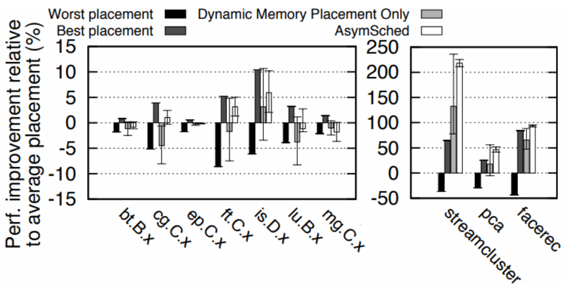


Fig. 5. Performance [3]

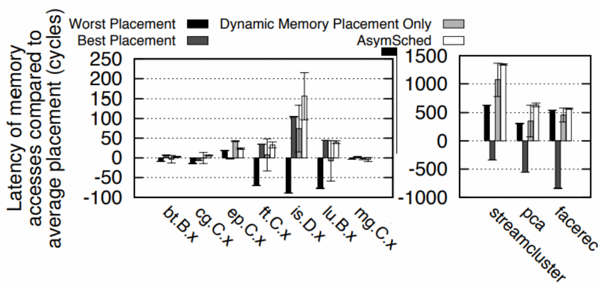


Fig. 6. Memory latency [3]

VI. CONCLUSION

Creating an efficient multi-purpose scheduler is an immensely difficult task, which is why the Linux Scheduler has been in use in so many systems ever since it's introduction. Specific optimization for certain architectures and problem statements can be made quite well. Major challenges are collecting relevant data for making the scheduling decisions and scheduling in a way that minimizes migrating processes and their memory around. This is done by introducing extensions which greatly decrease power consumption and optimize memory access behavior, but in doing so often can add additional overhead from collecting data and migrating a lot of data, or perform slightly worse at simple tasks. Nonetheless they achieve an overall considerable improvement in their use-case, laying the groundwork for future research to build upon to create broader or more efficient scheduling applications.

REFERENCES

[1] T. Yu, R. Zhong, V. Janjic, P. Petoumenos, J. Zhai, H. Leather and J. Thomson; Collaborative Heterogeneity-Aware OS Scheduler for Asymmetric Multicore Processors; in IEEE-TPDS; 2021, pp. 277-289
[2] B. Nesterenko, Q. Yi, J. Rao; Improving Resource Utilization through Demand Aware Process Scheduling; in ICPP 2018; 2018
[3] B. Lepers, V. Quéma, A. Fedorova; Thread and Memory Placement on NUMA Systems: Asymmetry Matters; in USENIC ATC '15; 2015
[4] I. Jibaja, T. Cao, S. M. Blackburn, and K. S. McKinley; Portable performance on asymmetric multicore processors, in Proc. Int. Symp. on Code Generation and Optimization; 2016; pp. 24-35
[5] A. Silberschatz, P. B. Galvin, G. Gagne; OS Concepts; Wiley; 2018
[6] A. S. Tanenbaum; H. Bos; Modern Operating Systems; Pearson; 2015
[7] "Linux Scheduler", in docs.kernel.org/scheduler, 2022

Modern Batch Scheduling by example of Flux

Leander Hacker
Bachelor's Degree Program (Computer Science)
Technische Universität München
leander.hacker@tum.de

***Abstract*—Scheduling many heterogeneous tasks with varying resource needs is usually accomplished with job scheduling. High-Performance Computing (HPC) centers often use centralized resource and job management software (RJMS) like SLURM.**

However, this approach has limitations regarding large numbers of jobs and so-called ensembles of jobs. The latter have very dynamic and often unpredictable behavior concerning resource needs. Current RJMS gets challenged by these Workloads.

In this paper, we analyze these challenges and introduce the Flux framework. It utilizes a hierarchical, tree-like structure of nested scheduler instances to divide and conquer these problems. We also show how Flux can increase the system performance while obeying given constraints.

This is done by reviewing already written papers on this topic and condensing them into a beginner-friendly introduction to the subject.

***Index Terms*—batch scheduling, flux, job scheduling, hpc, scheduling**

I. Introduction

A. HPC Center structure

"High Performance Computing (HPC) is used to solve a number of complex questions in computational and data-intensive sciences. These questions include the simulation and modeling of physical phenomena, such as climate change, energy production, drug design, global security, and materials design; the analysis of large data sets, such as those in genome sequencing, astronomical observation, and cybersecurity; and the intricate design of engineered products, such as airplanes." [4] The most common architecture in modern HPC Centers are cluster-based systems with many off-the-shelf CPUs structured into compute nodes. [4] These nodes are then connected and often accelerated with additional hardware like GPUs which provide another type of resource that has to be managed. [4] Modern HPC Centers are very fragmented and contain a large number of different resources which is why there is a need for efficient communication between nodes and resource management. Since there is almost no need for interactive programs or regular user inputs, the main paradigm for HPC is batch scheduling.

B. Job scheduling

The user interacts with the HPC center by relying on a dedicated Job Scheduling framework. "For the execution of applications on HPC systems, a so-called job is created and submitted to a queue. A job describes the application, needed resources, and requested wall time. An HPC Job Scheduler manages the queue and orders the jobs for efficient use of the resources." [5] Currently, this System works by running a Resource and Job Management Software (RJMS) Instance on every node which is responsible for the local Scheduling and Management. An example of such software would be SLURM. The whole HPC Center also uses grid software like MOAB, PBS Pro, and LSF to combine these local RJMS instances. [1] The architecture of this approach uses a static and flat hierarchy for managing the Center. [1]

C. Technology Trends

There are however problems with this approach: First of all the ever-increasing size of the systems in combination with growing resource diversity makes these common strategies less effective. [1] Modern scientific applications often rely on having access to many different resource types and fast communication between these. [1] Another Problem with current solutions is the difficulty to enforce global constraints on resource usage. [1] "Finally, the workloads themselves are becoming diverse, dynamic, and large, and are moving away from individual monolithic jobs. Instead, ensembles of jobs, [...] are becoming increasingly commonplace." [1] These ensembles are defined by the large number of small jobs that are involved and by their dynamic nature. Such workloads can unpredictably spawn many jobs that terminate quickly and send the results back to their origin. [2] Spikes in job creation can be a real problem for traditional approaches since the centralized grid management software represents such a bottleneck for the whole system. The steadily growing HPC centers around the world demand a novel approach to deal with challenges like these. Without such a solution the performance gains of the upcoming exascale of large Computation centers will be eaten up by an ineffective, static, and overhead burdened scheduling infrastructure.

II. Flux

A. Exploration of the Challenges

1) Throughput Challenge: The emergence of large ensembles of jobs as visualized in Fig. 1, lies at the core of the throughput challenge. Current centralized Schedulers are unable to support the creation and management of thousands to millions of small jobs. A currently active workaround is the system-wide cap on the number of jobs that can be created at once. This however limits the needed job throughput of applications and throttles the overall performance of the HPC center. [2]

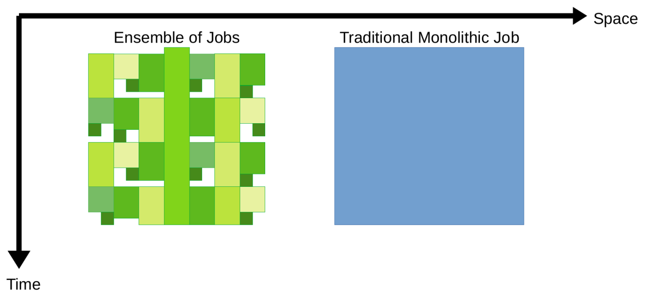


Fig. 1. Ensembles of Jobs vs. Traditional Monolithic Jobs

2) Co-Scheduling Challenge: It is often important for different parts of an application to be tightly coupled. The amount of information that has to be transferred between various jobs is also increasing because of diverse resource types being used. This is where co-scheduling comes into play: It is effective to deploy other jobs alongside a primary job on a single compute node. These can then analyze the behavior of said primary job and relay information to other jobs. [2]

3) Job Coordination and Communication Challenge: The need for reliable and efficient data transfer between different components of a large-scale application is ever-growing. If, for example, an unusual scenario is recognized, additional computational effort is often needed to analyze it further. This is mostly done by different components that need to coordinate and communicate efficiently. Since current scheduler architectures don't support this fully, a workaround through the file system is regularly employed. But since the file system is slow and lots of empty files and unnecessary metadata is a burden on the whole system, this approach is ineffective. [2]

4) Portability Challenge: The effort to deploy a new workflow to a wide range of RJMS can often be overwhelming, since they may not support needed features and require workarounds. This process of rewriting scripts and tweaking the workflow can lead to several bugs that have to be ironed out before execution. All of this reduces productivity and wastes computational resources. It is, therefore, necessary to provide platform-independent APIs to save time and ease the deployment of new Applications to varying HPC centers. [2]

5) Multidimensional Scaling Challenge: It is difficult to adhere to center-wide or local resource bounds like power while maximizing resource utilization and efficiently scheduling workloads on different levels. The System must be scalable and handle great amounts of nodes, jobs, and generated data. This multidimensionality poses a big challenge. [1]

6) Diverse and Dynamic Workloads Challenge: Since different applications need different types of resources, it is important, that those resource types are represented in modern RJMS. Such a system must be able to allocate these diverse resources to jobs while tailoring said allocation to their limiting factors. This however must adhere to strict global bounds. [1] If a job suddenly requires more resources, its allocation should be able to grow or shrink dynamically. This so-called elasticity is important for applications that go through different phases that have differing needs. Since each resource has a unique level of elasticity ("e.g., power is a much more elastic resource than compute nodes" [1]) the implementation of such a system can be complex. [1]

7) Productivity Challenge: Since the development of modern scientific applications for HPC centers is getting more and more difficult, a modern RJMS framework should provide a set of tools for efficient diagnostics and analysis. This can help developers and system administrators simultaneously. [1]

B. Introduction of Flux

A proposed solution to these challenges is the relatively new open-source RJMS framework called Flux. It is supposed to be "scalable, easy-to-use, portable, and cost-effective" [2]. Its conceptual design will be presented in the coming subsections:

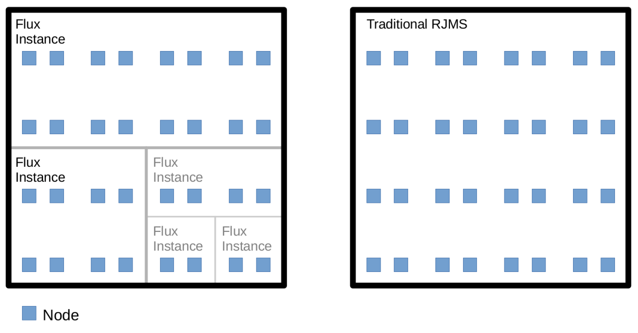


Fig. 2. Flux vs. traditional RJMS node allocation

1) Job Hierarchy model: "At the core of Flux lies its ability to be seamlessly nested within allocations created by other resource managers or itself, along with allowing for user-level customization of policies and parameters." [2] As depicted in Fig. 2 Flux employs the "divide-and-conquer" [1] strategy to ease the burden of the root level Scheduler and leave the fine-grained details of scheduling jobs to a hierarchy of Flux instances. [2] Flux utilizes the described hierarchical job management by "organizing itself in a tree-based hierarchy of Flux jobs." [1] Three principles decide which instance has control over and responsibility for resources: [1] "Parent bounding rule: the parent job grants and confines the resource allocation of all of its children. [. . .] Child empowerment rule: within the bounds set by the parent, the child job is delegated the ownership of the allocation and becomes solely responsible for most efficient uses of the resources. [. . .] Parental consent rule: the child job asks its parent when it wants to grow or shrink the resource allocation, and it is up to the parent to grant the request." [1] These rules govern the processes of resource allocation, job scheduling, and elastic allocation resizing. This allows a Flux instance to only focus on managing its children jobs. "As sibling jobs run simultaneously, their independent Flux instances will perform concurrent management services." [1] This enables scheduling parallelism. [1]

2) Unified job model and generalized resource model: In Flux, a job is an independent Flux instance that can run a single Application or a full job management service. This recursive nature allows for the utilization of specialized service plugins that alter the behavior of the Flux job. Custom plugins can be useful for defining unique scheduling strategies or resource constraints. [1] This advanced feature is useful for researchers because it enables them to test new scheduling policies by just writing a simple plugin. Flux also extends the notion of a resource beyond nodes and ”introduces a generalized resource model that is extensible and covers any kind of resource and its relationships. This enables scheduling decisions based on many types of resources.” [1] These resource allocations can then dynamically grow or shrink by traversing the job hierarchy tree until all resource constraints are guaranteed to be met. The three rules come into play when it comes to this elasticity.

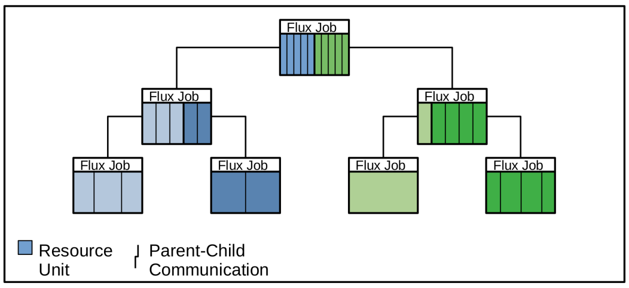


Fig. 3. Tree hierarchy of Flux jobs and their communication links (color = job)

3) Common scalable Communication infrastructure model: ”Flux provides a common scalable communication framework within each job.” [1] All allocated nodes are connected by this framework to enable efficient communication within the application. When communicating across jobs, the communication session can only interact with its parent and children (as can be seen in Fig. 3). This limits communication between jobs, ”addressing the multidimensional scale as well as security issues.” [1]

4) Common universal APIs: Flux also provides a set of APIs for: ”job submission, job-status and -control, messaging, as well as input and output streaming” [2]. These can be leveraged by user Applications to solve the communication and coordination challenges as well as the development- and debugging challenges. [1] [2] Since the APIs are consistent regardless of the used platform, the portability of Flux is greatly increased. [2] ”Each level also allows customizable scheduling policies and parameters, addressing both the throughput and co-scheduling challenges.” [2] This customizability is essential for implementing advanced features that promise to improve efficiency. [1]

III. Experimental Results

Researchers at LLNL conducted experiments to verify, that Flux increases the job throughput when dealing with a large number of jobs. [2] They measured the ”average number of jobs ingested, scheduled, and launched per second (the higher, the better).” [2] and compared three different depth hierarchies. The depth-1 hierarchy corresponds to a flat and static centralized scheduler, while the higher depths ”distributed the jobs equally among the lowest level of schedulers” [2]. The researchers also compared a real-world Uncertainty Quantification ensemble workflow with a stress-test ensemble where the jobs instantly exited after being launched. [2]

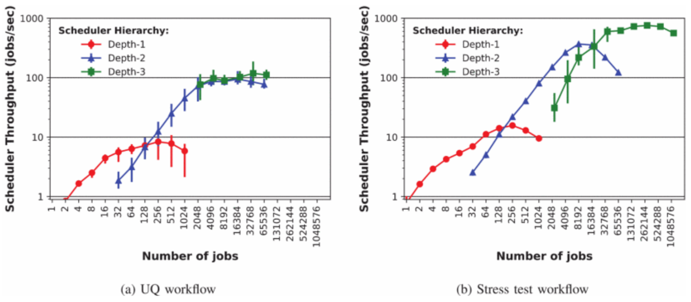
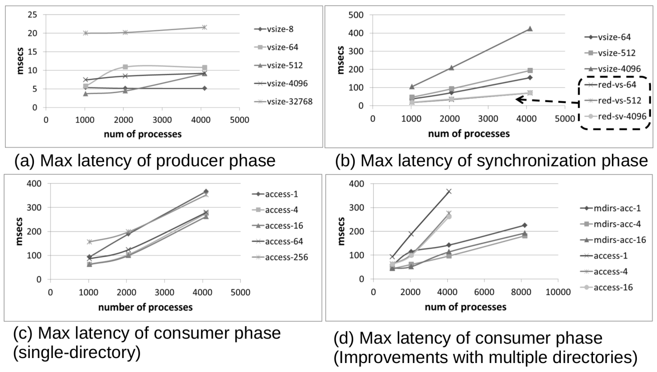


Fig. 4. Job throughput (in jobs/sec, on a logarithmic scale) for the depth-1, depth-2, and depth-3 scheduler hierarchies for fixed-size clusters and differing numbers of total jobs (on a logarithmic scale) [2]

Figure 4 demonstrates the depth-1 scheduler can only handle up to 10 jobs/sec. This wastes computational Resources since new jobs have to wait to be launched and nodes idle. The higher depths increase the number of jobs/sec by a full order of magnitude and dramatically cut back on the scheduling overhead by employing scheduler parallelism. [2] When the Flux framework is no longer limited by the computational resources of the HPC center, or when the job runtime is negligible like in this test, the depth-3 scheduler can improve upon this and reach a peak throughput of 760 jobs/sec. This is a 48-fold improvement over the flat hierarchy. [2]

Another group at LLNL tested the scalability of the communication infrastructure of Flux. This includes the Comms Message Broker (CMB) which relays information as well as the integrated Key-Value Store (KVS), which is used for storage and data access. They used the KVS Access Pattern (KAP) Method, which uses many producers which write data to the KVS and several consumers which read said data. This test walks through 4 phases: setup, producer, synchronization, and consumer phases. First, the agents are distributed across the hierarchy, then the producers put the correct number of items into the KVS. Afterward, synchronization happens and the consumers read the items and verify data consistency in the KVS. [1]



(a) Max latency of producer phase (b) Max latency of synchronization phase
(c) Max latency of consumer phase (single-directory) (d) Max latency of consumer phase (Improvements with multiple directories)

Fig. 5. Max latency of different KAP phases [1]

Figure 5 (a) demonstrates, that writing data to the KVS scales very well. Even if the number of processes increases dramatically, the latency of putting the item into storage is still small. Bigger items naturally have higher latency. [1]

Figure 5 (b) shows, that the latency does increase as the number of processes synchronizing grows. When redundant (red-) values in the KVS are used, the scalability greatly increases. [1]

Finally Fig. 5 c and d point out, that the latency of KVS data access also scales linearly. The more objects are retrieved at once (access-1 vs. access-4), the longer it takes. When using multiple KVS directories to store the data, the scalability is also greatly increased (mdirs-acc). [1]

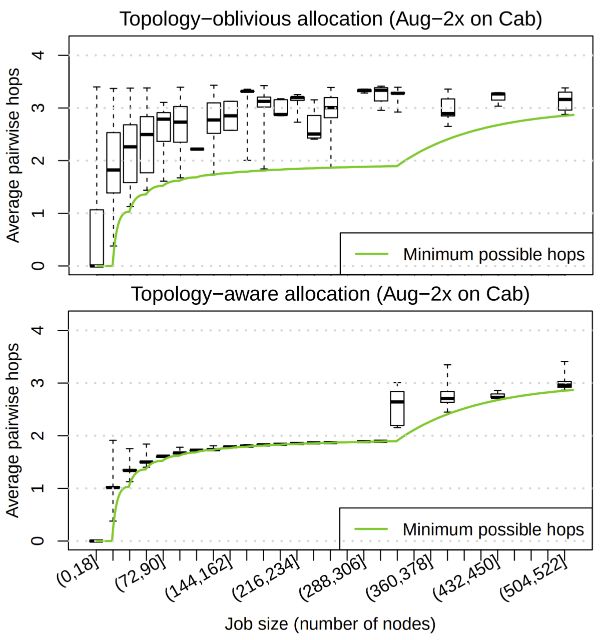


Fig. 6. Average pairwise hops for the Aug-2x logs on Cab. [3]

Finally, the third group of researchers used Flux's capabilities to be customized to implement topology-aware scheduling. The idea of this approach is to allocate nodes in a way, that minimizes the number of hops across network switches and eliminates network interference between jobs (e.g. two jobs should not use the same communication link). The experiments assumed, that the network infrastructure of the center was a fat-tree, which can be characterized as a tree topology, where the higher-level branches have a higher capacity. [3]

When comparing this new approach with the status quo using historical scheduling logs from the Cab cluster, Fig. 6 demonstrates, that the number of hops is greatly decreased and comes very close to the theoretical minimum. Their paper also demonstrates, that this new approach has no big drawbacks, when it comes to throughput, utilization, or wait times. [3] Flux's ability to be customized by inserting a custom scheduling policy into the job scheduler is very advantageous for researchers who want to try out novel approaches.

These Experiments clearly show that Flux can increase the throughput of Large HPC centers by a lot. Flux itself can scale very well and doesn't buckle under the load of many processes. Its great customizability is also very practical and can lead to further performance gains. The classical "divide-and-conquer" principle still applies to this day and can leverage the upcoming hardware performance increases.

However, the higher complexity of Flux compared to a centralized approach is only worth it, if a very large number of jobs (e.g. ensembles of jobs) are involved.

IV. Conclusion

We have analyzed the challenges that modern centralized RJMS frameworks face. The biggest among them is the emergence of large ensembles of jobs. To solve these challenges, we examined Flux, which utilizes a hierarchical layout and employs the "divide-and-conquer" strategy to enable scheduler parallelism. Flux also provides a modern unified resource model and can handle elastic resource allocation as well as co-scheduling and efficient intra- and inter-job communication. We finally presented experimental proof of some of these characteristics. However, its use-case should be limited to large HPC centers handling ensembles of jobs to justify the higher complexity.

References

[1] Dong H. Ahn et al., "Flux: A Next-Generation Resource Management Framework for Large HPC Centers," Lawrence Livermore National Laboratory, Computation Directorate

[2] D. H. Ahn et al., "Flux: Overcoming Scheduling Challenges for Exascale Workflows," 2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), 2018

[3] Samuel D. Pollard et al., "Evaluation of an Interface-free Node Allocation Policy on Fat-tree Clusters," Lawrence Livermore National Laboratory, Computation Directorate

[4] Jeffrey S. Vetter, "Contemporary high performance computing," Boca Raton, Taylor Francis, 2013

[5] Mehmet Soysal and Achim Streit, "Collection of Job Scheduling Prediction Methods," Springer, Job Scheduling Strategies for Parallel Processing 24th International Workshop, JSSPP 2021 - Revised Selected Papers

Large-Scale Cluster management by example of Borg

Simon Räde
Technical University of Munich

***Abstract*—Google's Borg system is one of the largest Cluster management systems in the world. To manage multiple tens of thousands of requests a second, it uses a mix of ordinary scheduling techniques like a classical round-robin and priority based approach, mixed with more unique systems. The main key to working at such a scale is the communication between each part of the system. Borg gives good exmamples of a hierachical structure which is in place to prevent conflicts. Aditionally Borg is a prime example of how many saftey/documentation layers are needed for a system of that scale. I present a summary of Borgs general functions, its architecture, some of the intricacies that are relevant to the scheduling aspects of Borg and lastly I take a look at some of the numbers behind Borg to allow a more detailed view of the sheer scale of this system.**

***Index Terms*—Borg, Scheduling**

I. Introduction

The scheduling of normal user systems has barley changed over the last decades. The same cannot be said for the systems that handle the ginormous workloads that modern day tech giants such as Google incure. The Borg system is one of the premier Large scale cluster management systems, it manages multiple tens of thousands of requests each second. Thus making it a very capable example to understand the metrics and techniques used in these immense structures. It can show off many unique challenges that only exist when taking a closer look at a system of that size. Aditionally it gives an insight into some of the peculiarities that come with working on such a scale and the small details that can't simply be ignored like they would be on a smaller system. By looking at the development of Borg over the last decade, we can also see the way such systems adapt. Aditionally it has given us the possibility to look at the numbers behind such a system. With systems like this one where a single percentage point of the throughput can be due to multiple thousands of machines, the cost of resources can mean that any small performance decrease can lead to massive lost revenue. This in practice means it is paramount to closely inspect each of the systems aspects to make sure that no resources go to waste.

II. The physical architecture of Borg

To allow for the quickest response time Borg is made up of a multitude of clusters spread around the globe. A single cluster usually consists of about 10000 machines.[2] The machines itself vary widely in terms of the hardware that is used. Each machine is capable of running multiple tasks at once. This setup allows Borg to adapt to many different outside influences, such as the varying usage of the system throughout the day. Two machines compute completely independent from the each other and only communicate with their assigned superior. Inside the cluster Borg consists of a multitude of independent cells. Each cells operations are handeled by the so called "BorgMaster". The BorgMaster's job consists of both scheduling the tasks and handling different requests, like starting a new task or changing some of the parameters for one of the current jobs[1]. Under the Borgmaster, and present on every single one of the cells machines, is a Borglet. This Borglet handles all operations on its machine, and is responsible for reporting back to the Borgmaster. Every few seconds the Borgmaster requests an update on the current state from the Borglet and gives it any new requests. The Borgmaster then saves this information in a local storage in case the Borglet becomes unresponsive. This backup data can then later be used for debugging or simply to start up another machine to replace the failed one. In case Borglet becomes unresponsive for multiple polling cycles, the Borgmaster decides to reschedule the tasks on another machine to ensure that they are completed on time.[1] Should the original Borglet come online the Borgmaster sends it the signal to kill all tasks that were rescheduled to prevent duplication. This can lead to some wastefulness, but is still the most reliable and quickest way to deal with such a machine failure. To ensure that the Borgmaster stays available at all times, it is always replicated five times. The currently active Borgmaster writes

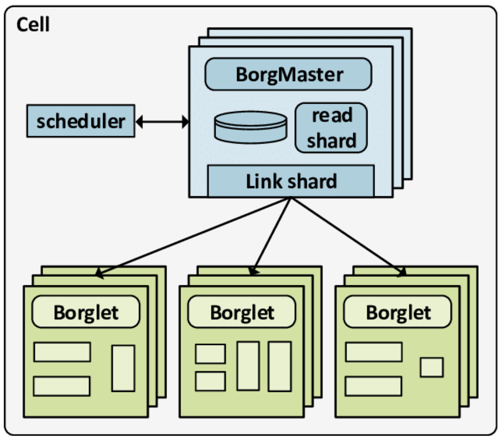


Fig. 1. A small fraction of the high-level architecture of Borg!

all its data into local memory. Should the Borgmasters now cease execution for whatever reason, one of the others would simply take over since it can replicate all the information it needs from the locally saved storage.

III. Challenges of Large Scale Scheduling

Many of the goals and with them the requirements of Scheduling are directly ported over from normal User Machines to the Large Scale Clusters. However with the larger and more diverse workload comes an increased need to be able to handle each different task adequately. For that purpose Borg cells are essentially split into two different parts. One for services that are end-user-facing, the other is for larger Jobs. Not all cells are split equally into end-user and Batch part, this allows Borg to better adabt to the change on user requests, since the normal userbehavior dictates that the number of user requests fluctuates based on the time of day [1].

IV. How does Borg schedule different tasks

Borg assigns different Priorities to each task. These can be structured into five categories.[2] The first and highest priority category is refered to as the ”Monitoring Tier”. This class is reserved for jobs that are essential to keep the infrastructure working as is, an example would be jobs that are responsible for detecting whether jobs have failed and the handling of such errors. Right behind these jobs is a category of scheduling that gets classified as ”Production Tier”. These jobs are mainly end-user oriented and thus are very timesensetive. The three remaining categories can all be classified as batch jobs with different priorities, so we don’t need to further subdivide them. These large scale and slow Batch jobs that can take anywhere from a couple of minutes to multiple months to complete.
Once a job has been selected for scheduling, the sheduler goes through two processes. Firstly it needs to check every machine to find the ones on which the task could possibly be executed on. After that it needs to pick out one of those to actually let the job run on it. In determening which of the machines is best to have the task be running on the scheduler takes into account what number of tasks would be preempted, their priorities, but also what machine already has the required packages. Aditionally the scheduler is concerned with trying to minimise the damage a potential shutdown could cause by allocating different tasks belonging to the same job on different powergrids.[2] Lastly it is important for the scheduler to try and keep some headroom incase of a sudden increase of requests. The current system for allocating the tasks is especially concerned with stranded resources, which are resources that can’t be used since some other resource on the machine is fully occupied. The resulting system is reportedly about 3-5 percent better at packing the machines than the simple approach of best fit.[2] After all of that the only problem the scheduler still has to deal with is the problem of starvation. To solve this last problem Borg determines jobs that can fit the same niche and have similar priorities and uses a round robin system to allow each job to progress calculation.

A. *Scheduling of high Priority jobs*

Since most of the High Priority jobs are extremely time sensetive, the system doesn’t have the time to assert the resource requirements for most of these jobs, so they just get everything they request. But since these jobs rarely stay in the system for longer than a second, this means that fixing these small inaccuracies is simply not worth the extra time investement of trying to more accurately assess the job. Otherwise the high priority job essentially just gets the slot it needs to allow it to best calculate efficiently. There is few consideration for other jobs when a high Priority job needs a spot so evictions of batch tier jobs are very commonplace.

B. *Scheduling of low Priority jobs*

With low Priority jobs come new challanges. One of them is that a lot of these batch jobs typically request less resources to allow them to better run on unused processor space. This means that despite their immense size they often use up a lot less than a single CPU core, leading to them taking often months at a time to complete.[2] These jobs also often come with a lot of baggage in the form of packages, that can take over 30s to install. This is especially problematic since these jobs only ever run on leftover CPU space meaning, that they have no guarantee to even run for that long at a time. The scheduler tries to minimise this through careful consideration of which machine is best suited for running a certain task, based on thich of the machines has the necessary packages already installed.

V. Different kinds of usages for Borg

With Borg being used for all kinds of Computations, there are a lot of different types of usages to keep track of.

A. *Allocs*

One of the more important things for Borg to keep track of are Allocs. An alloc refers to a set number of resources that can be reserved on a machine. This allows certain tasks to be executed faster than others since they already have a reserved spot on the machine and don’t need to go through the normal scheduing process. Inside the alloc the system works just like the one used on a single machine. Even though the number of allocs compared to normal jobs is miniscule, they are often used for more calculation intensive tasks. They are despite their small number still responsible for about 20 percent of the total CPU allocation and 18 percent of the RAM.

B. *Particularities of the end-user side of Borg*

One of the types of requests Borg has to deal with are small, but time sensetive requests, that directely impact end-users. Here, the requirements for Scheduling dont’t change all that much from platform to platform, be it a normal Home Computer or the massive infrastructure, that makes up Borg. These requests usually only take up to a few hundred ms to complete, most of them are however a lot faster.[1] Hence a single job isn’t all that hard to handle and problems only really arise when it comes to handling hundreds of thousands of jobs a second. As is in the nature of user-facing services, the first and foremost concern is reliability, so for these jobs it is vital to be able to reroute them should a machine fail. The second most important thing these jobs need is speed. This is achieved by not routing them to Clusters that are geographically far away, but rather handling them locally. These requirements lead to these jobs typically having a very high priority, so that they are immideatly processed on arrival.

C. Specific problems with scheduling of batch jobs

On the other hand we have the batch jobs. These are large jobs that can in some cases take months to complete. This obviously means that small delays are a lot less impactful, so that these jobs can be assigned a lower priority. The biggest problem coming with jobs of this immense size, is allocating enough space to allow the calculation. For that reason these jobs are often split into many smaller tasks. However most batch jobs can still evict an even lower priority job, this can in some cases lead to cascading evictions, since this is still a very unlikely scenario Borg has no special protocolls to prevent this[1]. This set of circumstances leads to a lot of batch jobs using only small amounts of the CPU to allow them to better run in the background while a higher priority job is taking up most of the actual capacity. Otherwise batch jobs are waiting for a window when there's not a lot of high priority jobs around to use the CPU for their purposes until another task comes in, that evicts them again

VI. Monitoring

Due to the sheer size of the Borg infrastructure, it is to be expected that there are some machine failures. This means that it is essential to keep a close eye on each machine. Borg achieves this by having the BorgMaster communicate with the Borglet on each machine under it every few seconds. Through this communication the Borgmaster collects information on the current state of the Borglet and its assigned tasks. To allow users to access this information it's in most cases also stored on a HTTP server together with extensive logs of the machine to allow for debugging in case of an unexpected failure. This information includes the state of the job and the state of its cell [1]. With this information it is then possible for the user to examine the resource usage of single tasks. Additionally all the execution logs are also stored here for a time. Due to the size of these logs it is impossible for Google to store them for longer periods of time without eventually running out of space to store new ones. They are however kept long for a while even after finishing the task to assist the user in debugging.

VII. Scalability

In 2019 the each Borg cell received 3360 jobs per hour on average. This marks an immense increase when compared to 2011 where only 964 jobs would be expected in the same timeframe. This scale means that even a single Borgmaster may need up to 50GiB of RAM and more than 10 CPU cores all to himself. To allow operations on this scale to go smoothly Borg employs some techniequs to simplify the processes. These are:

a) The ordering of tasks into different Equivalence classes, based on their requirements. This removes the requirement of checking wether each individual task can fit into a given Slot since now only one task per equivalence class has to be checked [1].
b) The principle of relaxed randomization has the same effect, wherein not all possible tasks are considered, but simply an arbitrary number of randomly selected possible candidates. From this smaller samplesize the Scheduler can then pick out the best candidate, reducing the overall time since not all tasks have to be assessed [1].

Currently the absolute limits of Borg's Scalability are not yet known since even though the workload is ever increasing Borg has not yet hit a point where clear shortcomings could be noticed

VIII. Autopilot

One of the biggest problems facing Borg is the waste of resources. To ensure tasks don't run forever and also don't start leaking, users have to define a set boundry of CPU and memory capacity their task is not allowed to exceed. This leads to most of the users entering information that would greatly exceed the tasks needed maximum resources, to make sure it doesn't accidentally get shut down should it require more of the CPU than previously thought [3]. In turn a lot of space on the machine gets wasted, without any real reason. To combat this Google now employs Autopilot. A program that amongst other things, tries to tune the limits set by the user to more accurately reflect the actual needs of the program. This means that Autopilot needs to accurately assess the needs of each job so that it doesn't end up causing an Out of Memory event, resulting in that job getting killed.[3] For this purpose autopilot takes into consideration a number of factors inherent to the task. Amongst other things, these factors mainly concern the resilience of the task as well as the importance of a quick execution. For latency sensetive tasks, the autopilot has to make extra sure, it doesn't reduce the tasks resources so far that it would fail. This is less important for tasks that can simply be restarted without problem. Autopilot is however still somewhat of an opt-in experience since the user has the option of setting a boundary of how far the pilot can reduce or increase the tasks resources.

IX. Developments

To better assess Borg we can use some of the Data Google has given us. May 2011 aswell as May 2019 they monitored parts of the system, allowing us to take note of some changes that occured in this timeframe and also get a better feel for the scale of operations. One obvious development is that, as mentioned previously, since 2011 the workload has grown by a factor of about 3.7. Despite this the median time it takes the scheduler to get a task up and running has actually decreased.[2] This could simply be a result of more lower effort tasks being scheduled, since the data shows that the scheduling delay for bigger low effort jobs has increased. Aditionally it is visible that the free tier of prioritiy seems to have fallen out of favor. Instead they are scheduled as Best Effort Batch tier, allowing them to take advantage of the batch scheduler. This batch scheduler is also something new in the 2019 trace, it is a new type of scheduler that queues all the available batch jobs and waits until they can be handeled to hand them over to the regular scheduler. Another thing we can take from the trace is that on average the utilization has gone up. This means that less resources are idle at a time, thus marking an increase in efficiency. Whilst in 2011 the surveilled cell only averaged around about thirty percent of total CPU capacity, in 2019 the cell shot up to a usage of over 50 percent of the total CPU capacity. Similar things can be observed with the Memory usage, where a comparable increase from 30 to 60 percent of the total capacity is present.

X. Conclusion

Through closer examination of Borgs systems we can assess that whilst many of the core principles that were already applied when trying to schedule a lower workload. Many things that were trivial or simply didn't make a difference now have to be taken into consideration. The biggest difference is the wide variety of tasks the system has to deal with. On a network of a normal scale there's not enough of a discrepancy to warrant the extra effort, but for something the size of Borg, with calculations that can run up to months at a time, these small performance increases add up. Another thing that leads to rather interesting differences to a normal scheduling system, is the way the Borg clusters are spread across the world. To cope with this amount of variance Borg has needed to be very adaptable. It has achieves this using wildly different machines and a great deal of careful consideration.

Refrences

[1] Abhishek Verma and Luis Pedrosa and Madhukar R. Korupolu and David Oppenheimer and Eric Tune and John Wilkes, "Large-scale cluster management at Google with Borg", 2015
[2] Muhammad Tirmazi and Adam Barker and Nan Deng and Md Ehtesam Haque and Zhijing Gene Qin and Steven Hand and Mor Harchol-Balter and John Wilkes, Borg: the Next Generation, 2020
[3] Krzysztof Rzadca and Paweł Findeisen and Jacek Świderski and Przemyslaw Zych and Przemyslaw Broniek and Jarek Kusmierek and Paweł Krzysztof Nowak and Beata Strack and Piotr Witusowski and Steven Hand and John Wilkes, Autopilot: Workload Autoscaling at Google, 2020

Job Scheduling for Heterogeneous HPC Systems

Oskar Thaeter
Bachelor's Degree Program (Computer Science)
Technische Universität München
oskar.thaeter@tum.de

***Abstract*—Heterogenous systems are being increasingly deployed in high performance computing systems, growing the need for schedulers which consider this heterogenous architecture and use it effectively. Scheduling only for CPUs must be differently considered than scheduling for CPUs, GPUs and hardware accelerators. This paper presents different approaches and goals when scheduling for a heterogeneous system. The paper reviews literature on scheduling for CPU-GPU systems and considers the impact and importance of custom scheduling for diverse computing systems. Accounting for the rise in heterogeneous HPC, we conclude scheduling strategies will only increase in importance.**

***Index Terms*—heterogeneous system, scheduling, HPC, GPU**

I. Introduction

In recent years, the use of specialised hardware beyond the CPU has increased, revolutionising the architecture of high performance computing systems. This is illustrated by the TOP500 list, ranking the fastest high-performance computers in the world: Whereas in November of 2018 [2] five of the top ten systems used GPU accelerators, in 2021 [3] this number increased to seven. Overall accelerator usage increased from 104 in 2015 [1] to 138 in 2018 and 151 in 2021. This trend reflects the changing requirements for HPC workloads, increasingly benefitting from highlyparallel compute devices. First, we define what a heterogeneous system is and the differences to a traditional HPC system. Then we review the approaches proposed and challenges to scheduling for heterogeneous systems. Lastly, we discuss the presented methods and possible future developments.

II. Background

A heterogeneous system is comprised of the traditional CPU architecture with additional accelerator hardware, often in the form of graphical processing units (GPU). Because these GPUs excel in parallel execution, leveraging many processing units, they multiply performance on some tasks like matrix and linear algebra calculations compared to only using CPUs. These improvements especially benefit machine learning tasks, such as training artificial neural networks.
With the non-uniformity of the system's computing devices, traditional scheduling strategies will be unable to fully utilise the components. To maximise the usage of the available computing resources, different approaches are needed.

III. Scheduling for heterogeneous systems

To effectively make use of a heterogeneous system, the scheduling strategy needs to consider the differences of the installed computing devices. Assume we have a scheduling problem, consisting of K jobs, with N CPUs and M GPUs. As the jobs are diverse in their resource requirements, let job k_i have requirements $n_{r_{k_i}}$ and $m_{r_{k_i}}$ for CPUs and GPUs respectively. If we were to use a conventional scheduling strategy, we would either completely ignore the GPUs, just using the CPUs of the available resources, or consider CPUs and GPUs to be the same, limiting the performance of both CPUs and GPUs to have them comply with a common performance target. These options are obviously not satisfactory.
Ideally we need to consider that jobs require x CPU and y GPU resources, the different scaling behaviours of specific jobs on CPUs and GPUs. Say we want to train an artificial neural network. Although this job can run on either CPUs or GPUs, training on a CPU will take significantly more time than on a GPU. Using more CPUs will scale differently compared to when using more GPUs.

A. Unrelated-machines scheduling problem

Scheduling on heterogenous systems can be abstractly viewed as unrelated-machines scheduling. We need to schedule n jobs J on k different machines M. Job j is processed by machine m in $t_{m,j}$ time.

$$z_{m,j} = \begin{cases} s & s \in N, \text{ m is processing j s times} \\ 0 & \text{otherwise} \end{cases}$$

denotes if machine m is processing job j. We constrain us to one machine working on one job at a time.
Supposing all n jobs arrive in the beginning and all jobs are independent of each other and can be run in parallel on an arbitrary number machines, let c_j be the cost of processing job j and $v_{m,j} \in [1, c_j]$ be the value of machine m processing job j. Here, cost c_j would be the longest completion time when a baseline machine processes job j, $v_{m,j}$ being a factor of that c_j proportional to the amount machine m is faster compared to that baseline. For example: job A takes CPU E 25 time units, which means $c_A = 25$. But GPU F only takes 5 time units, making F 5 times faster than E on job A, i.e. $v_{E,A} = 1$ and $v_{F,A} = 5$.

$$t_{m,j} = \max\{0, \min\{v_{m,j}, c_j - \sum_{x \in M \setminus \{m\}} z_{x,j} \cdot t_{x,j}\}\}$$

We want to minimise $\max\{z_{m,j} \cdot t_{m,j}\}$ in order to minimise the maximum completion time.
Now taking into account that jobs do not all arrive at the same

time, adding more and more compute devices does not scale linearly and other complex and importing factors, it becomes apparent, that scheduling on heterogeneous scheduling is more complex than homogeneous scheduling. All this is to say, unrelated-machines scheduling is NP-hard, i.e. finding an optimal solution in polynomial-time is not possible. [4]

B. Genetic algorithm to approximate an optimal solution

Genetic algorithms (GA) are often used when an approximate solution is acceptable and a measure of optimality, fitness, is easily available. A GA is kind of like a guided, randomised algorithm, employing the concept of darwinian evolution [5]. It is best employed in the exploration of vast search spaces which do not allow for methods such as gradient descent. The main components of a GA are a fitness function, selection, crossover and mutation. Important is also the encoding of a solution, called chromosome, the pool of solutions, called population, and the iteration of the GA, called generation.
In the case of scheduling, fitness can be denoted by the completion time of a scheduling plan. The selection process is where the most fit chromosomes are selected to be carried over to the next generation. Crossover recombines two chromosomes genes into a new chromosome, incorporating parts of both "parent" chromosomes. Mutation is the random change of a gene in the chromosome, which is supposed to keep the population diverse.
Ayari et al. [6] propose an improved genetic algorithm for scheduling on heterogeneous multi-core systems. Their model is composed of a pool of n preemptive tasks to be executed on m processing elements. A task is defined as the tuple $t_i = <T_i, C_i, D_i, \Pi_i>$ with T_i being the period of task t_i, C_i is the worst case execution time vector of task t_i on all PEs, D_i is the deadline of task t_i and Π_i its priority. Vector C_i considers the heterogeneity of the system. Integer coding is used, an array of size n being a chromosome, each position i holding the number of the PE assigned to execute task t_i, see Table I and Fig. 1.

i	0	1	2	3	4	5	6	7	8	9
PE	0	0	2	0	1	2	1	1	3	3

TABLE I: A scheduling solution encoded as a chromosome

An initial population is generated using a climbing hill repairing strategy on a random initial generation. Meaning, the $|P|$ solutions in P are each evaluated for if a small change in their genes can produce a fitter solution. Potential mating (crossover) candidates are randomly chosen to compete in 1-versus-1 tournaments, the fitter of the two being selected for the mating pool. The fittest resulting children will be in the new population, while the best chromosomes of the current generation are also carried over. Ayari et al. [6] introduce a guided crossover operator, which gives a PE with low utilisation a higher chance of being inherited by the child. This will lead to a more distributed load on the system. For

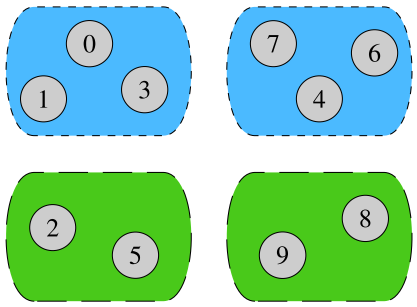


Fig. 1: The resulting scheduling representation of Table I

mutation, a circular permutation is used, in which all genes are shifted by one step to the right. The assigned tasks of the lightest and heaviest processing element are also swapped. To further avoid early convergence, new, random chromosomes are injected into the population when convergence is detected. The presented approach is able to improve upon conventional genetic algorithms, yielding higher quality solutions and slower convergence towards a near-optimal solution. It also was able to achieve a higher ratio of schedulable tasks according to the schedulability test by Liu et al. [7] which states that a schedule is feasible, if the total processor utilisation remains below a certain upper bound: Task-set τ_j containing k tasks of processor j is schedulable, if

$$U_j = \sum_{i=1}^{k} \frac{c_{ij}}{T_i} T \leq k \cdot (2^{\frac{1}{k}} - 1)$$

holds.

C. Assignment of optimal number of GPUs in deep learning

Part of the reason GPU usage has increased in HPC systems is the popularity of machine learning (ML). ML benefits greatly from GPUs, as they are well suited to handle large amounts of data in a parallel fashion. In particular, deep learning (DL), a subset of ML, depends on a high number of multi-dimensional calculations. Often, multiple GPUs are used to improve DL training throughput. Counterintuitively, just adding more GPUs to the system is not an effective solution, as DL training does not scale linearly. Additionally, it is often not possible to change the allocation of GPU resources during training.
To improve utilisation of GPU resources in this setting, Han et al. [8] propose a multi-GPU scalability-aware job scheduler called MARBLE. It uses a suspend and resume method to dynamically assign GPU resources and runs multiple DL jobs on a single node. It also incorporates the scalability of a specific job to assign the optimal number of GPUs, which increases DL training throughput.
The goal is to minimise total training time $T = \sum\limits_{\forall i} t_i$. MARBLE uses a FIFO-based scheduling policy to preserve task order. A primary job is defined as a job which has the optimal number of GPUs assigned to it, a secondary job is a job which does not have the optimal number of GPUs available

and shares GPUs with other secondary jobs. MARBLE starts with the first jobs in the queue as primary jobs until not enough GPUs are available anymore.

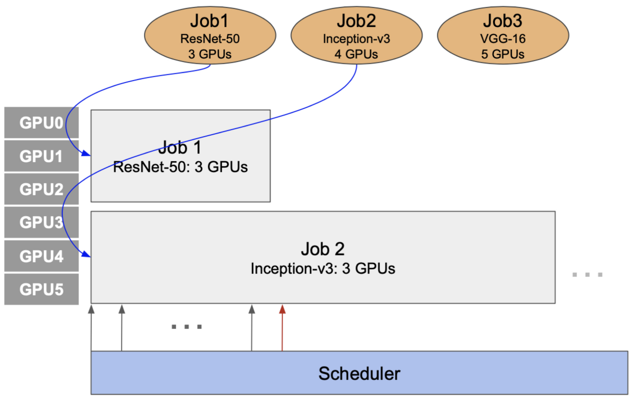


Fig. 2: Initial job assignment [8]

If some GPUs remain available, the next job in the queue is marked as a secondary job, using a sub-optimal number of GPUs to execute, see figure 2. As jobs complete, GPUs become available to be newly allocated to secondary running jobs, promoting them to primary jobs. The secondary job is suspended in training and resumes as a primary job, see figure 3. This ensures a high throughput of jobs, as all jobs run optimally or will run optimally at some point. Only when no secondary jobs are left, an entirely new job can be assigned. Historical execution data is used to deduce the optimal number of GPUs for a job beforehand. This reduces the runtime and overhead of the actual scheduler.
MARBLE is able to improve performance by up to 48.3% and GPU utilisation by up to 86% compared to the widely used LSF scheduler. [8]

D. *Accounting for reliability*

As GPUs are being increasingly used in high performance computers, their reliability and longevity in these harsh conditions is coming into question. The Oak Ridge Leadership Computing Facility OLCF has faced the issue of leadership jobs, which use 20% or more of the compute nodes, having a higher failure rate due to ageing GPUs. In a system of the OLCFs size, 18 688 GPUs, the higher than expected rate of failures of the GPUs lead to the replacement of about 8 500 GPUs overall. It was discovered that stability was correlated to the number of past failures, age and physical location of the device. The physical location significantly dictates the operating temperature as cooling potential can vary substantially. Higher temperatures lead to faster degradation of compute devices increasing their failure risk.
To mitigate the impact on the user, OLCF developed a scheduling strategy which takes the GPUs stability into account. GPUs deemed stable are placed higher in the resource allocation list, making them more likely to be allocated to leadership jobs. Jobs needing less stability, for example small jobs with a short execution time and also CPU-only jobs, are assigned to suitable resources, meaning even unstable GPUs can still be utilised and contention for stable GPUs is reduced. This approach managed to achieve an additional 100 000 stable hours per week on large GPU jobs and a reduction of failures in leadership jobs from 65% to 46%. [9]

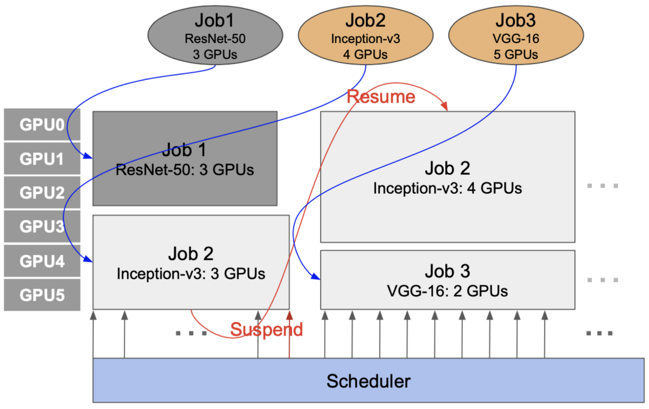


Fig. 3: Suspend/resume during training [8]

IV. Challenges

As mentioned before, the inherent challenge of scheduling for heterogeneous systems lies in the differences the compute devices have. Not accounting for these differences is simply not an option, as this negates the benefits of heterogeneity. Ideally, a scheduling strategy is able to incorporate the diverse resources of a system and use them to their fullest extend. But, this adds even more complexity to an already highly complex system. As illustrated in section III-A, devising an optimal scheduling plan is not possible in polynomial time.
One could rely on the user of a system to efficiently use the available hardware. But this will lead to under-utilisation of the system and end in a free-for-all of who gets hardware time and over-allocation by users.

V. Discussion

With section III-A illustrating the complexity of finding an optimal scheduling plan for a heterogeneous system, it is clear that approximating algorithms are the best way of squeezing every last bit of performance out of the system. Here the balance between overhead and potential optimality comes into question, for when does a scheduling plan become not worth it anymore considering its overhead on the system. But this approach is quickly turned ad absurdum, as the search space of even small scheduling problems grow dramatically when accounting for more and more complexities, making finding an optimal solution unthinkable. Alternatively, general approaches to approximating a scheduling solution like the genetic algorithm discussed in section III-B can provide a great balance between near-optimality, while still remaining within a set timeframe. The number of iterations can be adjusted to suit deadlines, making use of the available time for scheduling, providing a proportionally approximated near-optimal solution.

Customising the GA for the specific purpose of scheduling on heterogeneous system can yield even better results. Considering the ability of a task/job being executed on either CPU or GPU as in [10] could further improve the yield of usable solutions, for example a probability parameter for each task and processing device pairing. This makes favouring pairings based upon a myriad of factors possible when incorporated into the fitness function. It is also possible to seek near-optimal solutions according to alternative scheduling metrics like throughput, turnaround and so on. This only requires a simple change in the fitness function.

With heterogenous HPC systems becoming more common and workloads often relying on GPUs to execute in acceptable times, scheduling for heterogeneous systems is becoming increasingly more important. Machine Learning and specifically deep learning requires heterogeneous hardware environments to work effectively, using CPUs to handle pre- and post-processing and GPUs doing the highly parallel training of artificial neural networks. As reviewed in section III-C, application specific scheduling strategies can have a considerable impact on performance and utilisation metrics when compared to other scheduling strategies. This approach of optimising a scheduler for one specific purpose is promising, although reliant on the enduring usefulness and relevance of said purpose. Developing such a specialised scheduler is probably not feasible in most cases.

Considering the benefits of heterogenous scheduling beyond CPU-GPU systems, section III-D demonstrates a model which can be applied on scheduling for a different kind of heterogeneous system. One can consider any homogeneous system, say a HPC system with 10 000 CPUs, as a heterogeneous system, because of silicon quality differences and their physical locations in the system. Differing operating temperatures may lead to different rates of degradation of seemingly identical processors. Although these differences are minimal, a system may benefit from accounting for degradation of processors to ensure a uniform degradation of the hardware of the overall system. This may also lead to less unexpected stability issues and compensate for poorer silicon quality.

Looking at the future hardware being incorporated into heterogeneous system, scheduling strategies will remain important to actually leverage these new diversified computing environments. Cardwell et al. [11] describe the promise of integrating analog and digital neuromorphic computing to implement large-scale calculations with a low power-footprint. In reference to section III-C, instead of repurposing graphics processing units for training artificial neural networks, Google's tensor processing units (TPU) [12] are especially designed to accelerate the inference phase of neural networks. Custom application specific integrated circuits (ASIC) or field-programmable gate arrays (FPGA) should also provide performance gains in the specific application.

VI. Conclusion

We illustrated the increasing importance of heterogeneous systems and the necessary specific scheduling strategies for them. An abstract look at the unrelated-machine scheduling problem showed the unfeasibility of finding an optimal scheduling solution in our scenario. We reviewed state-of-the art methods for scheduling on heterogeneous systems with different approaches and goals, including bottleneck avoidance, deep learning specific and stability-aware scheduling. Lastly, we discussed the potential of the reviewed approaches and the future of heterogeneous systems and the scheduling strategies used by them.

References

[1] top500. (2015, November). Highlights - November 2015. https://www.top500.org/lists/top500/2015/11/highlights/

[2] top500. (2018, November). Highlights - November 2018. https://www.top500.org/lists/top500/2018/11/highs/

[3] top500. (2021, November). Highlights - November 2021. https://www.top500.org/lists/top500/2021/11/highs/

[4] Sahni, S. (1976). Algorithms for Scheduling Independent Tasks. Journal of the Association for Computing Machinery, 23(1), 116-127. https://dl.acm.org/doi/pdf/10.1145/321921.321934

[5] Darwin, C. (1859). On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life. 1st edition.

[6] Ayari, R., Hafnaoui, I., Beltrame, G. et al. (2018). ImGA: an improved genetic algorithm for partitioned scheduling on heterogeneous multi-core systems. Des Autom Embed Syst, 22, 183–197. https://doi.org/10.1007/s10617-018-9208-1

[7] Liu, C. L., Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM, 20(1), 46–61. https://doi.org/10.1145/321738.321743

[8] Han, J., Rafique, M., Xu, L., Butt, A., Lim, S., Vazhkudai, S. (2020). MARBLE: A Multi-GPU Aware Job Scheduler for Deep Learning on HPC Systems. IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, . https://par.nsf.gov/biblio/10167822.

[9] Zimmer, C., Maxwell, D., McNally S., Atchley, S., Vazhkudai, S. (2018). GPU age-aware scheduling to improve the reliability of leadership jobs on Titan. SC '18: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. IEEE Press, Article 7, 1–11. https://doi.org/10.1109/SC.2018.00010

[10] Yesil, S., Ozturk, O. (2022). Scheduling for heterogeneous systems in accelerator-rich environments. The Journal of Supercomputing, 78(1), 200–221. https://doi.org/10.1007/s11227-021-03883-5

[11] Cardwell, S.G. et al. (2020). Truly Heterogeneous HPC: Co-design to Achieve What Science Needs from HPC. In: Nichols, J., Verastegui, B., Maccabe, A., Hernandez, O., Parete-Koon, S., Ahearn, T. Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI. SMC 2020. Communications in Computer and Information Science, vol 1315. Springer, Cham. https://doi.org/10.1007/978-3-030-63393-6_23

[12] Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. ISCA '17: Proceedings of the 44th Annual International Symposium on Computer Architecture, 45(2), 1–12. https://doi.org/10.1145/3140659.3080246

Workflows and Scheduling

Deren Ertas
Master's Degree Program (Information Systems)
Technische Universität München
ga27gox@tum.de

Abstract*—The workloads of today's High Performance Computing (HPC) systems are strongly impacted by scientific workflows. They may be very significant, performing out a lot of activities and computations, managing a massive quantity of data, and finally appearing as thousands of concurrent process instances. Nevertheless, aside from the ability to express dependencies between their activities, HPC schedulers do not have workflow-specific features. They are commonly job-centric and therefore can not deal with the complexities of workflows which increases both the response time and the likelihood of missing workflow deadlines. Workflows are thus executed as tasks with dependencies or as a single job that contains the complete workflow. While workflows as chained jobs cause long intermediate wait times and, as a result, large workflow response times, single job workflows might waste resources despite their shorter turnaround times. For the optimization of turnaround times without reducing the efficiency of HPC systems, workflow fragmentation and scheduling have been widely studied in the past.We analyzed in this paper newly introduced approaches and discussed their benefits and challenges. We brought in this paper three of these important progresses for optimization of workflow scheduling together. These are a workflow-aware scheduling (WoAS) system optimizing the detection of workflows' resource needs and constructing an improved system with reduced respond times; and a strategy called GLUME dividing a workflow into batch jobs to optimize execution time of workflows on batch-scheduler managed platforms; and finally an adaptable, fault tolerant and flexible framework called Melissa proposing global sensitivity analysis together.

***Index Terms*—scientific workflows, batch scheduling, execution time, optimization**

I. Introduction

A. Background and Motivation

Workflows are a sequence of tasks and the interdependence of these tasks, which are categorized as either scientific workflows or business workflows [16]. As the database community recognized well [5], [6], scientific data management differs from more traditional business data management [5], [6]. Hence, workflows are called "business workflows" [5] when used in business process modeling for executing a task containing human elements. In that case the dependency within their tasks is control-driven, i.e. the next task can be executed only once the preceding task has been completed [9-15]. Scientific workflows' dependencies among their tasks are instead data-driven that the preceding task's output data is used in the following task. The database community's early work on scientific workflows adopted a database-centring approach, establishing data models and query languages. The ZOO experiment management system based on an object- oriented database [7], the FOX query language, and the MOOSE data model are a couple of examples introduced between 1980 - 1990 highlighting the role of workflow concepts in scientific data management.

B. Scientific Workflows and Scheduling

A scientific workflow describes a strategy for achieving a scientific goal using tasks and dependencies. Scientific workflow tasks are usually simulation or data analysis computations [5]. They include data collecting, integration, reduction, visualization, and publication [5]. Scientific workflows' jobs are ordered at design time according to data-flow and other requirements indicated by the designer [5]. Visual block diagrams or domain-specific languages can be used to create scientific workflows [5].

For the execution of workflows, scheduling systems are critical. So far, numerous scheduling frameworks have been developed. However, proposed scheduling systems include limiting constraints. These systems differ from each other based on their fragmentation method, run time, response time, execution of the fragment, the use of resources, run time conditions, throughput improvement and bandwidth costs. CTC, FPD, SLV, and QDA could be counted as some scheduling frameworks proposing some improvements on workflow fragmentation and execution costs. There are major problems with these frameworks such as mapping tasks to resources without considering all the necessary limiting factors. An important major problem is that the above mentioned frameworks generate a large number of fragments and communication messages among those fragments increase bandwidth usage. Increased delay time because of the number of communication messages and increased response time are also critical problems to be considered.

C. Historical Background

Examining the place of scientific workflow systems in history in more detail, we see that they have already been appeared in problem-solving environments in the 1990s. An instance that the computational sciences community came up with is a set of straightforward tools to fix an intended series of problems for scientific computing [4]. Laboratory information management systems (LIMS) [9] are another example. Such systems can be thought of as special scientific workflow systems which are used in laboratories as it's name states to manage samples, take measurements with instruments, analyze data, and automate workflow. Furthermore, the rise of e-Science has had quite an impact on scientific workflow research and development as well and they have gotten a big boost. In addition, computational techniques and technologies from the computational sciences, high-performance computing, databases, data analysis, visualization etc. have been combined by e-Science. Numerous innovative open source and proprietary scientific workflow systems, such as Kepler, Taverna, and Triana, are already available at the moment and developed actively.

II. SCIENTIFIC FUNDAMENTALS AND SYSTEMATIC OVERVIEW

When it comes to science, it is a process of discovery that incorporates cycles of observation, hypothesis generation, and experimentation. Nowadays, more and more scientific knowledge is found through data analysis and computational techniques. This is because there is an increasing number of useful observation tools and commodity clusters for high-performance scientific computing and simulations in the computational sciences. Consequently, the use of scientific workflows increases in different phases of science processes, such as modeling automated computational experiments or data management and analysis [5]. This evolves more and more need for optimization of workflow scheduling in HPC systems. Because workflow sequences could also give new information and analytic, which can be used to confirm, modify, or disprove a given hypothesis or experiment result [5].

Scientific workflow systems with extra functionalities might optimize, support, automate, monitor, and control the execution of scientific workflows. These may also play an important role in the workflows' design and management, making them more error-tolerant, efficient, and quick. Focusing on data-flow and concurrency data is critical to optimize the parallel execution of workflows, which is another crucial issue. This type of additional functionality differentiates scientific workflow systems from typical solutions that are script-based and lack equivalent functionality [5]. To provide these functions, many types of information, such as workflow history information, is necessary. This information, for instance may be utilized to enhance interpretation, debugging, and consistency of scientific works [5]. The utilization of different information leads generating different functionalities for scientific workflows.

A. Classification of Scientific Workflows

The workflows can either be submitted as a chained job, which minimizes the consumption of workflow provenance or as a pilot job, which focuses more on optimization of turnaround time. If a job can not start before its predecessors are completed, i.e. one batch job is submitted for each job in the execution plan, it is called chained job. In this approach, each job receives the precise resource set required to run, and assigned resources are not purposefully left idle. However, the workflow's overall run time will be increased further more by each jobs' critical waiting time.

On the other hand , workflows can also be submitted as a single pilot job. There is no intermediate waiting time, which consequently leads limiting job's time with the estimated critical waiting time. This approach assumes the maximum resource demand of any process during the execution refers the resource demand of the pilot job. Thus, the overall run time of the workflow is reduced. However this method may raise it presents the possibility that some of allocated resources will be unused.

In general, pilot job approach has an advantage in terms of run time but disadvantage in terms of provenance cost. When interpreting the facts regarding both approaches, it can be concluded that it depends on characteristics of the workflow when distinguishing one over the other.

Another classification of workflows is made based on whether they are dynamic or static workflows. Static workflows are used when prior information of the workflow architecture is required. Dynamic means that the structure of workflows is defined at run time.

B. Modelling of Scientific Workflows

Scientific workflows are represented visually as directed graphs as shown in “Fig. 1” and they are made up of different parts, which are called ”sub-workflows” [5]. They are frequently coarser-grained and include connecting pre-existing components and specialized algorithms [5]. Figure 1 depicts a workflow in Taverna using various services [5].

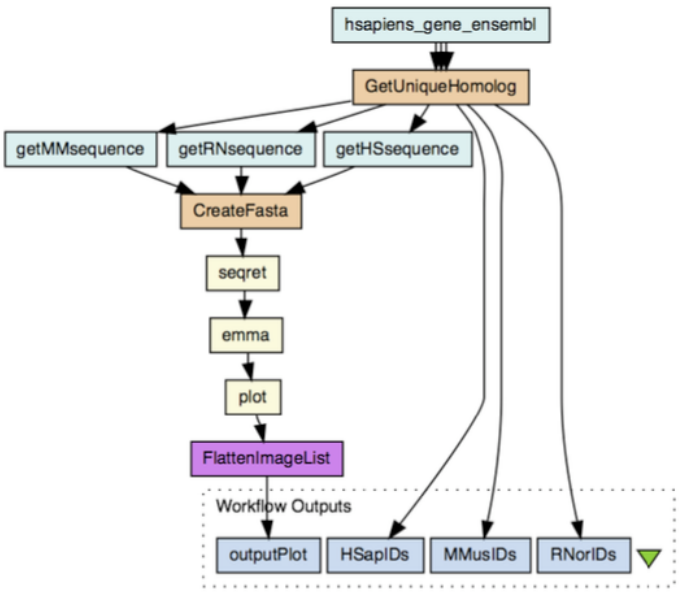


Fig. 1. Workflow in the Taverna workflow system [5]

It can not be said that there is a specific standard language for scientific workflows, and related standards such as BPEL4WS have not been widely embraced [5]. Commonly,

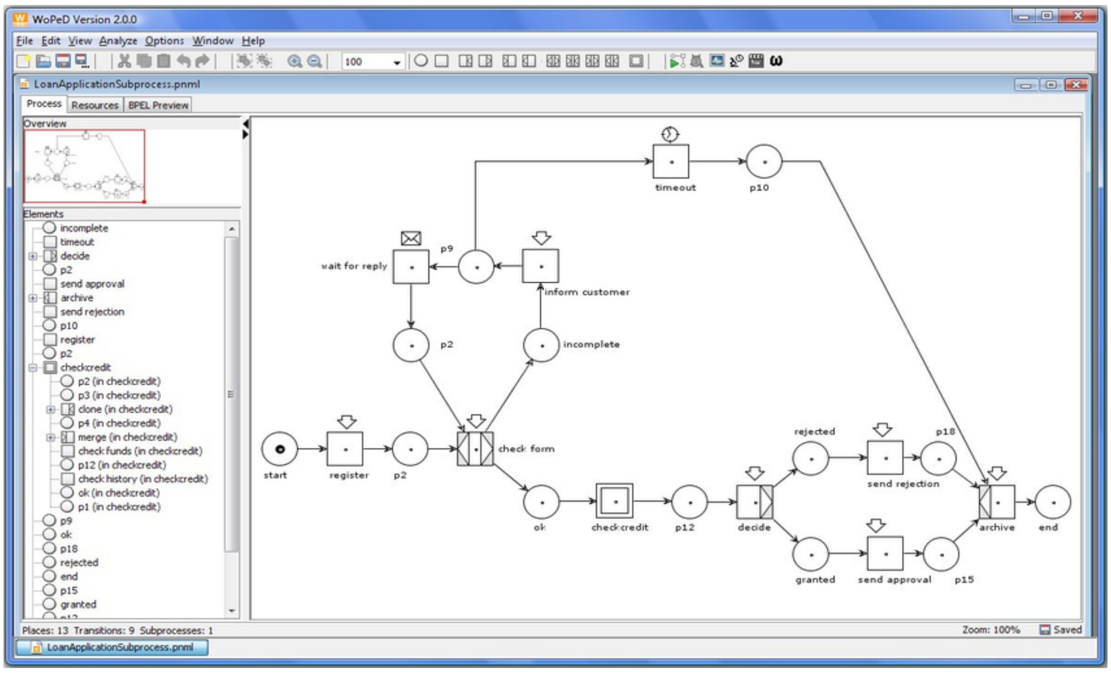


Fig. 2. Petrinets used to describe scientific workflow execution semantics [5]

directed acyclic graphs (DAGs) are used to depict job-based grid workflows. These models execute each task after each workflow execution. The job scheduling process involves computing a topological sort for the DAG's partial order [5].

There are also systems representing workflows in a more formal way by focusing on scientific workflow execution semantics such as Petrinets as shown in the Fig. 2. However, there are standard computation models used when there are specific issues or requirements, for example data-flow systems giving too much importance on token order. In such case the standard computation model called Kahn Process Network model would be used. Another special model of computation is linear workflows, for which an example appears on Fig. 3. This is the structurally simple linear Kepler workflow created by the COMAD (Collection-Oriented Modeling And Design) director [5], which is special for workflows consisting from continuous data stream components which can be computed only on tagged data-sets [5]. Moreover, the derived linear workflows are simple to interpret and adapt over time, which is a significant benefit over script-based solutions [5].

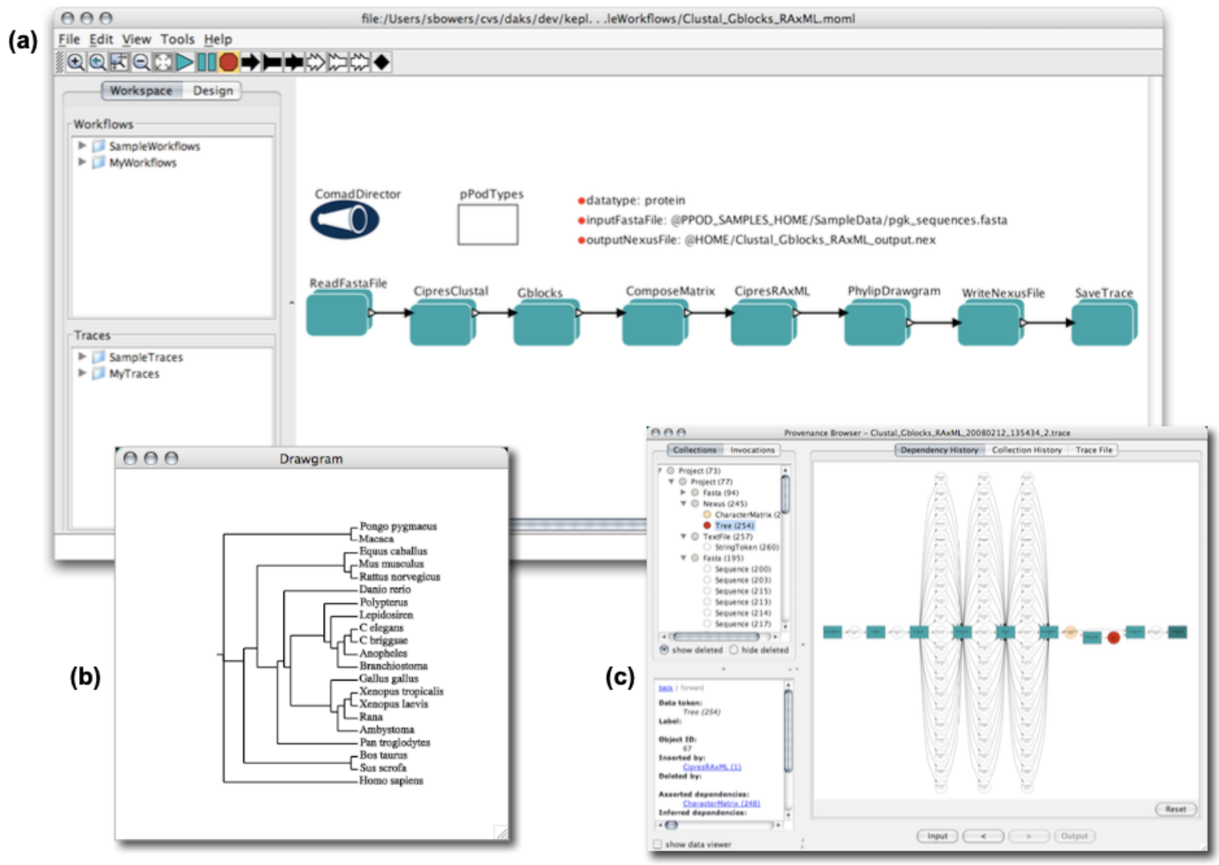
III. Related Work

This conference paper contains the summary and highlights of papers chosen by Dr. rer. nat. Matthias Maiterth, Prof. Dr. rer. nat. Martin Schulz, Eishi Arima, Dr. rer. nat. Isaìas A. Comprès for the Seminar: Scheduling – Modern Problems in a Seemingly Solved Discipline. Submitted papers went through a complete review process, with the full version being read and evaluated. This part of the seminar focuses on the current state of art of workflow management systems and scheduling methods needed to perform and optimize scientific workflows.

The authors present frameworks and studies to optimize workflow scheduling that will help future research. The first comprehensive paper by Rodrigo et al. proposes a workflow-aware scheduling (WoAS) system that exploits without modifying fine-grained information about the provenance needs and structure of a workflow[1].The paper analyzes the HPC batch scheduler Slurm, which WoAS is now integrated into, using a simulator with real and synthetic workflows and a synthetic baseline workload that captures task patterns from NERSC's supercomputer Edison [1]. Finally , the paper studies the impact of the WoAS on workflow turnaround times and system utilization without interrupting regular workloads [1].

Terraz et al. represent a sensitivity analysis using workflows and offer a file-free, adaptive, fault- tolerant, and elastic framework called Melissa [2]. This paper distinguishes between different workflow management systems focusing on combination of iterative statistics and in-transit processing [2]. The represented framework allows high resolution global sensitivity analysis at large scale [2].

The third article describes GLUME, a workflow execution time-saving solution. This system separates the workflow into sub-workflows with the objective of achieving the least run-time and predicted waiting period combination, hence achieving the most optimized completion time [3]. Based on this study, Hataishi et al. compare GLUME to other strategies exploit that as each task is finished, the remaining workflow becomes simpler and estimates become more accurate, thus the completion time gets shorter.



datatype: protein
inputFastaFile: @PPOD_SAMPLES_HOME/SampleData/pgk_sequences.fasta
outputNexusFile: @HOME/Clustal_Gblocks_RAxML_output.nex

Fig. 3. Example scientific workflow in the Kepler system. The part (a) describes the user interface for creating, editing, and executing scientific workflows. The part (b)is a visual representation of the data product computed by a workflow run. The part (c) is a viewer for navigating the data provenance captured in an execution trace. Local and remote (web) services are combined to create multiple sequence alignment on input DNA sequences. [5]

IV. OVERVIEW AND DISCUSSION

This section describes the state-of-art and current challenges in scientific workflow management, and discusses related work.

The Workflow-Aware Scheduling technique (WoAS) presented by Rodrigo et al., which is a new model for a batch queue scheduler [1] implemented within common HPC workload manager called Slurm [1], can be seen as one of the most beneficial approaches in comparison with current scheduling approaches and workflow life-cycles. After performing WoAS for the chain and pilot(single) job techniques, it can be said that for workflow-dominated workloads and for workloads with moderate workflow contents, WoAS achieves the shortest workflow turnaround times, high system usage values without idle provenance. In other words, it appears to work much better than the current workflow scheduling methods used in HPC systems and has no major drawbacks [1]. The findings show however that performance enhancements achieved during the construction of the back-filling method have a negative impact on the scheduling of very large workflows. Because they consider job priority values and dependencies during scheduling, which harm work priority computations on current HPC schedulers. As a result, even if a dependent job is part of a workflow that was submitted much earlier, the submission time given to it is the moment when its prerequisite job completes [1].

The slowdown of regular jobs was greater than when using chained job scheduling for LongWide workflow scenarios. To keep the system as highly utilized as possible FCFS or back-filling algorithms can be used. These algorithms keep turnaround times as short as possible by submitting workflows as single jobs. They can decrease that time even more than chain jobs can do. It can be expected from future studies to focus on performance optimization for workloads with more diverse workflows using WoAS [1]. As discussed in previous sections, optimizing intermediate storage throughout a workflow run is also important. Using repeated simulation runs, the Melissa approach can calculate ubiquitous Sobol indices, and it has been presented since it keeps statistics up to date without the use of an intermediate storage [2]. By doing so, this helps relieve the I/O bottleneck and provides for much larger scale sensitivity analysis [2]. Iterative statistics may be used in conjunction with the suggested client/server architecture to produce a fault-tolerant and flexible executable process [2].

	Accurate requested run times			Real requested run times		
	Worklows			Workflows		
	short	medium	long	short	medium	long
OneJobPerTask	14/18	**40/0**	**37/0**	18/19	**27/8**	**37/0**
LevelByLevel	**22/6**	**36/1**	**35/0**	**21/10**	**33/1**	**34/0**
OneJob	**9/4**	**10/1**	**5/0**	**17/6**	**17/1**	**9/0**
Zhang	5/18	**17/1**	**9/0**	13/20	10/11	**14/0**

Fig. 4. Wins/losses of GLUME against competitors [3]

While the Workflow-Aware Scheduling strategy fails to tackle the the delay of the execution of multi-run simulations with large data-sets, the use of Melissa approach could help to solve this challenge. After analysing both approaches, one may propose to combine both techniques to optimize the turnaround time for scheduling of very large workflows, on which the back-filling method showed a negative impact.

To deal with statistics computation from big data sets stored on drives, iterative statistics and Melissa's fault toleration may be coupled. This combination results in a small memory footprint and optimization for interruptions and restarts, even on a high-performance computing platforms.

Because all of these advantages have previously been evaluated on simulation groups that vary just in their input parameters, it has been authorized that the framework might begin simulation groups with various amounts of resources or even alternative simulation codes. The challenge then is for Melissa Server to appropriately combine the various data for updating the statistics. Further focus could be on the loop-back control, since all the strategies such as adaptive sampling strategies are adapted to dynamical control [2]. Because it has been seen that such strategies could be needed in future for the cases of time-consuming large-scale numerical simulations to build accurate surrogate models to be used for uncertainty and sensitivity analysis. It is still challenging to do in-transit analysis in such an adverse environment [2].

The framework presented for Melissa launcher is asynchronous and the scheduling of the simulation groups used for the tests depending on the supercomputer load lead also the parameter sets to be created randomly [2]. To conclude, the current contribution of Melissa to the large scale global sensitivity analysis are its' optimization by reducing the I/O time of imitation of processes [2]. It also provides a better visual environment for pervasive index maps of Sobol for large scale [2]. Currently, Melissa is a trustful approach which has been approved on many use cases and offers unreplicable workloads for specific sensitivity inquiries.

Another challenge for scientific workflows is their need for batch scheduler platforms when medium to high complex computations demanded. For this computations, High Performance Computing platforms for workflows involve adaptability issues with batch scheduling. Additionally, there are not enough approaches proposed for batch-scheduled platforms. Resource and Job Management Software (RJMS) performing batch scheduling face adoption challenges while application-level solutions are impeded by constraints imposed on batch jobs. GLUME (Group Levels Using Makespan Estimates) strategy helps to suit these applications to batch schedulers by dividing them into batch jobs. Thus, the workflow execution time on batch-scheduled platforms is optimized and it has been approved that it's more effective than currently used methodologies based on the experiments/simulations.

The current two partition approaches are one- job-per-task approach and the one-workflow-as-a- single-job approach. Both approaches yield to long execution times because of waiting times on work- flows with long/many tasks and job expiration when they overlap. The application-level approach, GLUME, can be used to solve this issues even with non-workflow-aware, standard RJMS-level solutions. So, they make it feasible to determine how to combine sequential levels fairly. This is made by relying on wait time estimates as provided by production batch schedulers instead of predictions. The previously used algorithms such as Zhang leading to overlap run/wait times and unsatisfied task dependencies also cause the job to idle before being able to execute its tasks and fail to complete within the job's requested run time [3]. Fig. 4 shows wins and losses of GLUME against competitors and helps to observe how it results in workflows with different sizes.

GLUME and Zhang are both techniques that are used continuously during workflow execution to choose the next group of sequential workflow levels to submit as a single task [3]. The difference is that Zhang optimizes the wait time/run time ratio of the next task to be submitted using a greedy approximation algorithm, while GLUME seeks to lower the make-span directly. Furthermore, GLUME's default is not the one-job-per-task method, and it only enables two active workflow jobs in the system at the same time, avoiding the influence of per-user limits on the number of running jobs [3].

GLUME beats its competitors for batch workloads provided workflow computational demands are high enough. Its another advantage is that it provides accurate wait time estimations, which can not be provided by other batch schedulers currently. Glume divides workflow levels into jobs more efficient than Zhang as well. The only challenging situation for GLUME is when it is applied to short workflows. Because other algorithms benefit from back-filling and suffer from fractional wait times. This issue has been overcame by setting One-job-per-task as default making it effective for short workflows as well [3]. However, current trends make it clear that longer workflows are more broadly relevant to current practice in most scientific application domains [5].

After handling all challenges and benefits of algorithms and approaches, it can be discussed that GLUME could be optimized by assigning each work-flow job minimum number of nodes and user- provided resource. Another challenging optimization to prevent long waiting times on single but large workflows could be to make horizontal as well as vertical partition of workflows possible, i.e. dividing each level into multiple jobs [3]. The best earning from GLUME would be making its usage wider on all standard batch scheduler managed tools.

V. CONCLUSION

After getting used with the characteristics of workflows and workflow scheduling methods, we interpreted current methodologies and experimental results that we received from research papers. All approaches proposed to optimize the execution time of workflow scheduling and each has a different contribution for future improvement. WoAS (workflow-aware scheduling) uses fine-grained in- formation about the provenance demands and structure of a workflow [1] without altering that information. The research proposing Melissa, a file-free, adaptive, fault-tolerant, and elastic framework for sensitivity analysis, discriminates between various workflow management systems by combining iterative statistics with in-transit processing [2] . It has been shown with this research that high-resolution global sensitivity analysis at large scales are made possible using this framework [2]. Finally the last research focuses on the challenge for scientific workflows' need for batch scheduler platforms in terms of computation. Against that challenge, GLUME, a time-saving solution for process execution, which is the subject of the third article has the goal of obtaining the least possible run-time and expected waiting time combination. This system breaks down the workflow into sub-workflows [3]. The research paper on GLUME examines how it stacks up against alternative approaches based on the observation that when tasks are completed, workflows simplify and estimations become more accurate, ultimately resulting in reduced completion times. In conclusion, it can be said that the combination of these approaches and the problems, which could not have been covered with these approaches are building the focus for the future studies on workflow scheduling.

REFERENCES

[1] Rodrigo, et al., "Enabling Workflow-Aware Scheduling on HPC Systems.", 26 June 2017, https://dl.acm.org/doi/pdf/10.1145/3078597.3078604.

[2] Terraz, et al., "Melissa: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.", ACM Conferences, 1 Nov. 2017, https://dl.acm.org/doi/pdf/10.1145/3126908.3126922.

[3] Hataishi, et al., "GLUME: A Strategy for Reducing Workflow Execution Times on Batch-Scheduled Platforms." Job Scheduling Strategies for Parallel Processing, 2021, https://link.springer.com/content/pdf/10.1007/978- 3-030-88224-2.pdf.

[4] Momenzadeh, et al., "Workflow Scheduling Applying Adaptable and Dynamic Fragmentation (WSADF) Based on run-time Conditions in Cloud Computing - ScienceDirect.", 13 Aug. 2018, https://www.sciencedirect.com/science/article/pii/S0-167739X18305909.

[5] Ludascher, et al., "Scientific Workflows.", Encyclopedia of Database Systems, Springer, 2009, https://doi.org/10.1007/978-0-387-39940-9.

[6] Shoshani, et al., "Characteristics of Scientific Databases.", Very Large Data Bases (VLDB), pages 147–160., 1984.

[7] Ioannidis, et al., "ZOO: A Desktop Experiment Man- Agement Environment." . Vijayaraman, et al., Proceedings of International Conference on Very Large Data Bases (VLDB), 1996.

[8] Medeiros, et al. "WASA: A Workflow-Based Architecture to Support Scientific Database Applications.", In Database and Expert Systems Application (DEXA), Springer LNCS 978, 1995.

[9] E. Donyadari, F. Safi Esfahani, N. Nourafza, F safi esfahani n. nourafza scientific workflow scheduling based on deadline constraints in cloud environment, Internat. J. Mechatronics Electr.Comput. Technol. 5 (16) (2015).

[10] R. Khorsand, F. Safi Esfahani, N. Nematbakhsh, M. Mohsenzade, Taxonomy of workflow partitioning problems and methods in distributed environments, J. Supercomput. 132 (2017) 253–271.

[11] M. Naghibzadeh, Modeling and scheduling hybrid workflows of tasks and task interaction graphs on the cloud, Future Gener. Comput. Syst. 65 (2016) 33–45. [14] V. Arabnejad, K. Bubendorfer, B. Ng, Scheduling deadline constrained scien- tific workflows on dynamically provisioned cloud resources, Future Gener. Comput. Syst. 75 (2017) 348–364.

[12] M. Atkinson, S. Gesing, J. Montagnat, I. Taylor, Scientific workflows: Past, present and future, Future Gener. Comput. Syst. (2017) 216–227.

[13] F. Safi Esfahani, M. Azrifah Azmi, Md. Nasir Sulaiman, N. Izura Udzir, SLA- driven business process distribution, in: Information Process and Knowledge Management 2009 EKNOW'09 International Conference on, 2009, pp. 14–21.

[14] F. Safi Esfahani, M. Azrifah Azmi, Md. Nasir Sulaiman, N. Izura Udzir, Using process mining to business process distribution, in: Proceedings of the 2009 ACM symposium on Applied Computing, 2009, pp. 2140–2145.

[15] F. Safi Esfahani, M. Azrifah Azmi, Md. Nasir Sulaiman, N. Izura Udzir, Run- time adaptable business process decentralization, In: The Third International Conference on Information, Process, and Knowledge Management, 2011, pp. 76–82.

[16] Momenzadeh, Zahra, Safi-Esfahani, Faramarz, Workflow scheduling applying adaptable and dynamic fragmentation (WSADF) based on runtime conditions in cloud computing, Future Generation Computer Systems, 2019, pp. 327–346.

[17] Amalarethinam, D.I. George, Muthulakshmi, P., An overview of the scheduling policies and algorithms in grid comput., Internat. J. Res. Rev., 2011, pp. 280–294.

Modern Problems in Co-Scheduling

Seminar: Scheduling - Modern Problems in a Seemingly Solved Discipline

Jonas August
Bachelor's Degree Program (Computer Science)
Technische Universität München
jonas.august@tum.de

***Abstract*—Recent developments, like CPU-GPU architectures, Last Level Cache (LLC) partitioning, and memory bandwidth partitioning, opened new opportunities to improve co-scheduling. This paper gives an overview about co-scheduling: what it is, why we need it, and what factors can be considered (performance, minimizing co-run interference, partitioning of power, placement on CPU or GPU, LLC partitioning, and memory bandwidth partitioning). After an introduction to the topic, three algorithms for solving modern problems in co-scheduling will be explained, evaluated, and compared. The main contribution of this paper is to give an overview of modern problems and possible solutions for co-scheduling.**

***Index Terms*—Co-scheduling, Co-run theorem, CPU-GPU architectures, LLC partitioning, Memory bandwidth partitioning**

I. Introduction

A. History

In the early days of computer science, most scheduling techniques assumed that concurrent processes are independent. However, with the introduction of Multiprocessor Systems, it started that a collection of cooperating processes uses multiple processors concurrently to solve a problem. The previously used scheduling techniques were very inefficient for this case. John K. Ousterhout noticed this problem in 1982 and introduced the term co-scheduling. ”Parallel Programs have a process working set that must be co-scheduled [...] simultaneously for the parallel program to make progress.” [4]

B. Modern problems

Even if co-scheduling itself has been a known problem for years, new technologies offer more opportunities to enhance co-scheduling further. For example, the trend toward integrated CPU-GPU architectures reduces the communication latency between processes, but intensifies the co-run interference. [1] Furthermore, supercomputers have so many cores (i.e., 2048 cores of the PEZY-SC2) that we can only deploy a few applications on the whole platform. So co-scheduling of multiple applications is needed for not wasting resources. [2] Additionally, Intel introduced the Cache Allocation Technology (CAT) that allows to reserve Last Level Cache (LLC) subsections for an application. [2] On the latest commodity server CPUs, a Memory Bandwidth Allocation (MBA) feature was introduced to throttle the traffic from the private level 2 cache to the LLC. [3] These new technologies open new possibilities for enhancing co-scheduling, which requires new algorithms.

C. NP-completeness

It has been shown that generally co-scheduling is an NP-complete problem. However, there are exceptions with less complexity, like when having only two cores. From this follows that there is in general no efficient way to find optimal solutions to co-scheduling. So most algorithms have to approximate optimal co-scheduling. [5]

D. Outline

This paper gives an overview of some recent problems for co-scheduling and algorithms to solve them. In section II, we look at the co-run theorem to see where co-scheduling is beneficial. Then, we introduce three algorithms:

- Section III-A: An algorithm to ”co-schedule independent jobs on integrated CPU-GPU systems with power-caps considered” [1].
- Section III-B: A dynamic programming algorithm with cache partitioning using the Intel Cache allocation technology (CAT) for iterative applications. [2]
- Section III-C: HyPart as a ”hybrid technique for practical memory bandwidth partitioning on commodity servers” [3].

Afterward, in section IV, we will discuss the introduced algorithms. Finally, in section V, the main results will be highlighted.

II. Co-Run theorem

The co-run theorem describes when it makes sense to co-schedule a job and when the job should run with exclusive access to the resources. The jobs J_1 and J_2 have a standalone length l_1, l_2, and a co-run degradation d_1 and d_2. The co-run lengths are $l_1 + l_1 \cdot d_1 \geq l_2 + l_2 \cdot d_2$. The co-run produces only higher throughput when $l_1 \cdot d_1 < l_2$. A visualization of a co-schedule, degrading the execution time, can be found in Fig. 1. This shows that sometimes, it can be better not to co-schedule jobs due to the co-run degradation. [1]

III. Algorithms for Co-Scheduling

A. Algorithm for CPU-GPU Systems with power cap

We can observe a trend towards chips with integrated CPU-GPU architectures that share the Last Level Cache

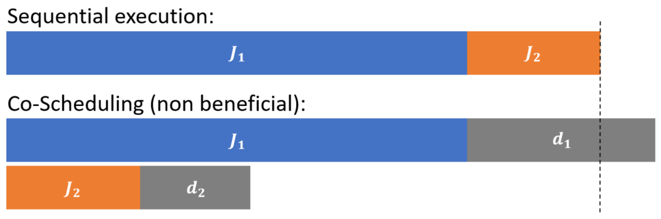


Fig. 1. Visualization of the co-run theorem.

(LLC) and the main memory. This architecture helps to shorten the latency of communication. However, the co-run interference between GPU and CPU Programs becomes more complex. This section presents an algorithm that considers job placement (CPU or GPU), power caps, and memory contention. The algorithm's output is a sequence of processes for the CPU and GPU. A schematic representation can be found in Fig. 2. [1]

1) Collecting metrics: In the first step, we have to collect some metrics of the jobs. The co-run performance of two jobs is obtained by measuring the standalone performance of each job and then calculating the co-run performance of each pair by staged interpolation. Second, we run a micro-benchmark on the CPU and GPU with eleven evenly distributed parameters of memory bandwidth for approximating the co-run degradation and the space memory contention. Third, offline profiling obtains power consumption at each frequency level. [1]

2) Creating sets: With this data, we can start with the scheduling algorithm. The input is a set of jobs and the collected metrics. These jobs are partitioned into two disjoint sets S_{co} and S_{seq} where S_{co} contains all jobs where a co-run exists that can profit from co-scheduling. We determine this by using the co-run theorem. The jobs in S_{co} are further partitioned into three sets: $CPU - preferred$, $GPU - preferred$, and $non - preferred$, using the comparison of the performance on the CPU and GPU. [1]

3) Greedy scheduling: We use greedy scheduling to create a sequence of jobs for the GPU and CPU: First, the algorithm picks the longest $GPU - preferred$ job for the GPU and then the job for the CPU, which is $CPU - preferred$ and has the lowest co-run interference. When a job is finished, we always try to pick a job from the $preferred$ set with the lowest co-run interference. If this is not possible, we pick from the $non - preferred$ set and then from the $un - preferred$ set. When all three sets are empty, the jobs in S_{seq} are scheduled sequentially. For considering the power cap, the algorithm has to traverse all possible frequencies for a co-run to assure that the power consumption stays under the power cap. [1]

B. Algorithm for LLC partitioning

Dynamic Programming with Cache Partitioning optimizes for the problem to execute m iterative applications on P identical cores. The applications share the cache of size C that can be partitioned into a fixed number X of fractions. An application gets p cores and x fractions of cache. The idea is to approximate the time $T(p, x)$ taken for one iteration of the application.

$$T(p, x) = t(p)(1 + h(x))$$

We do this by approximating the computation cost $t(p)$ and the slowdown $h(x)$ created by LLC cache misses. [2]

1) Obtaining the computation cost and slowdown: We approximate $t(p)$ with Amdahl's law, which considers which fraction s of the code can be executed sequentially. T_{seq} describes the sequential execution time with the complete cache:

$$t(p) = sT_{seq} + (1 - s)\frac{T_{seq}}{p}$$

It was observed that the slowdown $h(x)$ can be described with the power-law (also known as the $\sqrt{2}$ rule). The power-law describes the cache miss ratio $r = r_0(\frac{C_0}{C_{act}})^{\alpha}$, with the baseline cache C_0 (what the application needs), r_0 as the baseline cache miss ratio, and the available cache of size C_{act}. The parameter α ranges from 0.3 to 0.7. By generalizing the formula with $\alpha := 0.5$ we find the slowdown:

$$h(x) = a + \frac{r_0\sqrt{\frac{C_0 X}{C}}}{\sqrt{x}}$$

where a is a constant to avoid side effects. [2]

2) Time taken per iteration: Different applications may be required at different rates, like every iteration or every n-th iteration. So we introduce a weight β. Where $\beta = \frac{1}{s}$ means that $s - 1$ steps are skipped (i.e., $\frac{1}{4}$ means three steps are skipped). We can now describe the time taken per iteration as the weighted throughput:

$$\frac{1}{\beta T(p, x)}$$

[2]

3) Dynamic Programming: The objective is to minimize the time taken by the slowest application by maximizing its throughput. A solution can be found with the Dynamic Programming Approach (Division in partial problems and saving intermediate results [7]). $\hat{T}_i(p, x)$ describes the maximum weighted throughput for the first i Applications $A_1, ..., A_i$ using $p \in \mathbb{N}$ cores and $x \in \mathbb{N}$ fractions of cache. We start with the first application ($i = 1$) and add in each iteration step one more application until we arrive for all m applications at $\hat{T}_m(P, X)$:

$$\hat{T}_i(p, x) = \begin{cases} max_{p_1 \leq p, x_1 \leq x} \ \frac{1}{\beta_1 T_1(p_1, x_1)} & \text{if } i = 1, \\ max_{p_i \leq p, x_i \leq x} \ \{min\{ & \\ \qquad \hat{T}_{i-1}(p - p_i, x - x_i), & \\ \qquad \frac{1}{\beta_i T_i(p_i, x_i)}\}\} & \text{otherwise.} \end{cases}$$

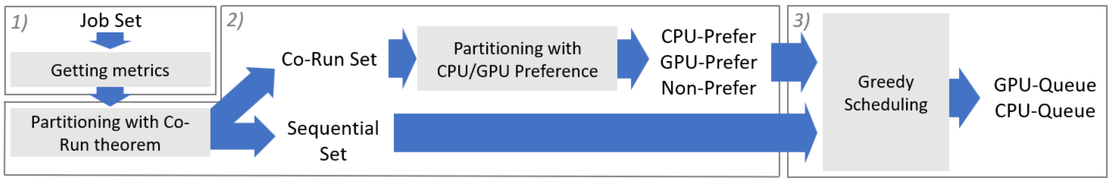
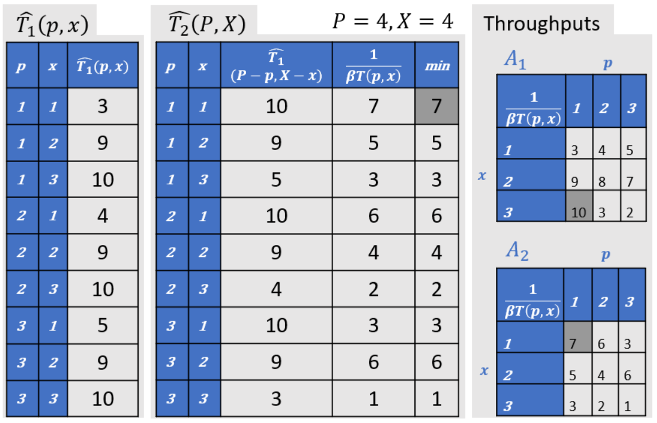


Fig. 2. Algorithm for CPU-GPU Systems with power cap. [1]



$\widehat{T_1}(p,x)$

p	x	$\widehat{T_1}(p,x)$
1	1	3
1	2	9
1	3	10
2	1	4
2	2	9
2	3	10
3	1	5
3	2	9
3	3	10

$\widehat{T_2}(P,X)$ $P = 4, X = 4$

p	x	$\widehat{T_1}$ $(P-p, X-x)$	$\frac{1}{\beta T(p,x)}$	*min*
1	1	10	7	7
1	2	9	5	5
1	3	5	3	3
2	1	10	6	6
2	2	9	4	4
2	3	4	2	2
3	1	10	3	3
3	2	9	6	6
3	3	3	1	1

Throughputs

A_1

$\frac{1}{\beta T(p,x)}$	p = 1	2	3
x = 1	3	4	5
2	9	8	7
3	10	3	2

A_2

$\frac{1}{\beta T(p,x)}$	p = 1	2	3
x = 1	7	6	3
2	5	4	6
3	3	2	1

Fig. 3. Example for Dynamic Programming.

In this way, an optimal solution under the assumption of the approximated metrics is found step by step. It can be shown that the algorithm has for m applications a complexity of $\mathcal{O}(mP^2X^2)$. [2]
An example of dynamic programming can be found in Fig. 3: For $m = 2$ applications, $P = 4$ cores and $X = 4$ fractions of cache, the solution (maximum of the min column) to $\hat{T}_2(P, X)$ is to use for application A_2 one core and one fraction of cache. Application A_1 has three cores and three fractions of cache available but uses only one core to maximize its throughput. If a third application would be considered it would use the just calculated table for $\hat{T}_2$ as intermediate result.

C. *Algorithm for bandwith partitioning*

The technique HyPart can be used to partition the bandwith of commodity servers. It composes three memory bandwidth partitioning techniques with specific advantages and disadvantages to dynamically perform optimizations. The three techniques for bandwidth partitioning can be characterized in dynamic range (range from minimum to maximum bandwidth), granularity (average memory bandwidth difference between levels), and efficiency (number of execution cycles per time). [3] First, the sole use of the three techniques is described, which are then combined in HyPart:

1) Thread packing: "Thread packing specifies how many threads should run on how many cores, and is used for packing multithreaded workloads onto a variable number of active cores" [6]. By varying the number of cores, the bandwidth can be controlled. However, this can not be applied to single-threaded applications. Performance anomalies can be observed when the allocated core count is not a divisor of the thread count. Some benchmarks are more tolerant against performance anomalies. This is the case, when the threads need little or no synchronization. [3]

2) Clock modulation: Clock Modulation skips a defined number of duty cycles, thereby reducing bandwidth. The disadvantage is that also non-memory-related instructions will be delayed. So the number of execution cycles needed is linearly increased. [3]

3) Hardware memory bandwidth allocation: Hardware-based bandwidth partitioning is possible threw a new technique from Intel called Memory Bandwith Allocation (MBA). MBA throttles outgoing traffic from the L2 cache to the LLC using a delay. As a result, mainly the bandwidth is affected without significant performance degradation, but a small dynamic range reduces its effectiveness. [3]

4) HyPart: The goal of HyPart is to configure the state (settings for thread packing, clock modulation, and MBA) to maximize the overall throughput. It is assumed that there are more or equal cores than the number of applications. The algorithm can be divided into 5 phases that work towards finding a good state out of the system state space (all combinations of settings). A schematic representation of the algorithm can be found in Fig. 4.

- **Elimination of sub-optimal states:** The observation that the fewest clock modulation and the lowest MBA setting tend to achieve the best performance helps to determine the sub-optimal states, which are removed from the state space.
- **Analysis of thread packing tolerance:** This can be done by setting the core count not to a divisor of the thread count for a short time. Then we count the idle cycles. Applications exceeding a threshold of idle cycles are non-tolerable to performance anomalies.
- **Elimination of thread packing anomalies:** Here, the states are eliminated where an application intolerable to thread packing would get a core count, which is not a divisor of the thread count.
- **State-space exploration:** With the tabu search (consider last results and information of the exploration process [8]), the state space is explored to find the most efficient

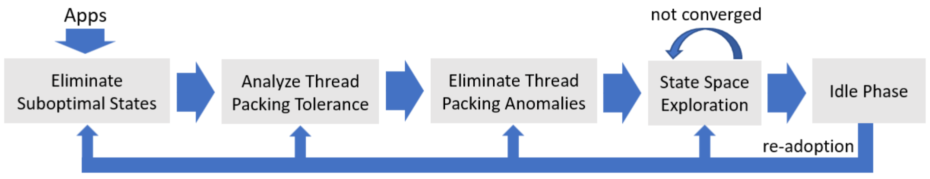

Apps
Eliminate Suboptimal States
Analyze Thread Packing Tolerance
Eliminate Thread Packing Anomalies
State Space Exploration
not converged
Idle Phase
re-adoption

Fig. 4. Algorithm for bandwith partitioning. [3]

state. At each adoption period, a new state is explored. It replaces the current state when it is more efficient than the current state. This is repeated until n-times a less efficient state was explored, where n is a predefined threshold. In this case, the idle phase will start.

- **Idle phase:** The applications will be monitored for changes like the termination of an application or changes in the memory bandwidth budget. When a change occurs, the procedure will be restarted.

[3]

IV. Discussion

A. Discussion of the algorithms themselves

1) Co-scheduling on CPU-GPU systems with power cap: The described algorithm is the first co-scheduling algorithm for CPU-GPU systems. It predicts the performance ($\leq$ 20% error for 70% of co-runs) and power consumption ($\leq$ 8% for all co-runs) of the co-runs very accurately. The algorithm shows 41% (37%) improvement over Random scheduling for 8 (16) program instances where the default scheduler performance is 32% ($-$9%) compared to Random scheduling. The algorithm scales reasonably well for more program instances compared to the default scheduler. The drawbacks are that this algorithm relies on micro-benchmarks and offline profiling. However, offline profiling could be replaced by existing, more lightweight methods. [1]

2) Co-scheduling with Cache partitioning: For this algorithm, no benchmarks are needed. The power-law struggles with memory-intensive applications. The prediction of the execution time loses accuracy when using small cache fractions. The model even though provides relatively good accuracy for the purpose of scheduling. The best results are achieved in co-scheduling memory-intensive with computation-intensive applications. Furthermore, the algorithm is considered fair. The algorithm can increase performance up to 100% compared to no cache partitioning. However, the benefit decreases with scheduling more applications or when applications of the same behavior (memory-/computation-intensive) are scheduled. [2]

3) Co-scheduling with memory bandwidth partitioning: HyPart combines three techniques for memory bandwidth partitioning. Thereby a robust performance can be seen across all settings. HyPart achieves a higher dynamic range, granularity, and performance than the three techniques themselves. It also performs optimizations depending on the characteristics of the target applications. However, application profiling is needed in the beginning. [3]

B. Discussion of the algorithms compared

The CPU-GPU scheduling algorithm is the only one considering a power cap which is a critical metric ignored by the other algorithms. It is also the only algorithm that uses the co-run theorem to avoid co-schedules that are degrading execution time. The cache partitioning algorithm approximates the computation cost and the slowdown by cache misses with formulas, whereas the other algorithms need profiling and micro-benchmarks from the applications. HyPart is the only algorithm that monitors the applications at run time and dynamically reacts to changes.

V. Conclusion

In this work, we presented three algorithms using new technologies to improve co-scheduling. It will be interesting to see if it is possible to combine these ideas to create new algorithms that use the benefits of CPU-GPU architectures, cache partitioning, and bandwidth partitioning to create an even better algorithm. We can conclude that for co-scheduling current research makes good progress in further enhancing scheduling and that there likely will be more such advancements in the future.

References

[1] Q. Zhu, B. Wo, X. Shen, L. Shen and Z. Wang, “Co-Run Scheduling with Power Cap on Integrated CPU-GPU Systems,” National University of Defense Technology China, Colorado School of Mines USA, North Carolina State University USA, pp. 1–10, 2017, IPDPS.

[2] G. Aupy, A. Benoit, B. Goglin, L. Pottier and Y. Robert, “Co-scheduling HPC workloads on cache-partitioned CMP platforms,” University Bordeaux France, Laboratoire LIP de Lyon France, University of Tennessee USA, Georgia Institute of Technology USA, pp.1–11, 2018, CLUSTER.

[3] J. Park, S. Park, M. Han, J. Hyun and W. Baek, “HyPart: A Hybrid Technique for Practical Memory Bandwidth Partitioning on Commodity Servers,” UNIST, pp. 1–14, 2018, PACT.

[4] J. Ousterhout, “Scheduling Techniques for Concurrent Systems,” University of California USA, pp. 22-30, 1982, IEEE.

[5] Y. Jiang, X. Shen, C. Jie, and R. Tripathi, “Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors,” College of William and Mary USA, Thomas Jefferson National Accelerator Facility USA, University of South Florida USA, pp. 220-229, 2008, IEEE.

[6] R. Cochran, C. Hankendi, A. Coskun, and S. Reda, “Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps,” Brown University USA, Boston University USA, p. 176, 2017, IEEE.

[7] E. Sean, “What is dynamic programming?,” Harvard University USA, 2004, Nature Portfolio.

[8] A. Hertz, E. Taillard, D. Werra, “A tutorial on Tabu Search,’ EPFL France, Montreal University France, p. 2, 1995, Proc. of Giornate di Lavoro AIRO.

I/O-Aware Scheduling

Marc Pavel
Bachelor of Informatics
Technische Universität München
pavel@in.tum.de

***Abstract*—Scheduling has been undergoing significant changes in the last few years. With the more and more data intensive applications, I/O Interference has become crucial to avoid. Therefore, the scheduler has to take the bandwith (BW) of the Burst Buffers (BBs) and the Paralel File System (PFS) into consideration. In this paper, we will analyze, compare and review two different papers that provided solutions for schedulers dealing with I/O heavy applications. While the first solution creates an I/O model and makes the scheduler I/O-Aware, the second paper introduces the idea of having a Multi-resource scheduler called BBsched, which considers necessary resources and provides solutions for different objectives. Both solutions show that by considering I/O resources while scheduling, the overall performance of a High Performance Computing (HPC) system can be increased at the cost of scheduling decision time.**

***Index Terms*—Burst Buffers (BBs), High Performance Computing (HPC), Parallel File System (PFS), Multi-resource scheduling, Multi-objective Optimization (MOO), I/O Interference**

I. Introduction

Over the last few years, there has been a growing demand in cluster computing systems at universities all over the globe in order to stream the different lectures reliably. While the HPC systems have been evolving at an exponential rate, the storage infrastructure performance is enhancing at a significantly lower rate [4]. This has led to a common issue, which HPC systems have been facing over the last few years: the I/O bottleneck. Since HPC systems are nowadays also used to run data intensive I/O applications, the Parallel File System (PFS) will not be able to provide the I/O performance required by those data-intensive I/O applications [7]. There are multiple solutions on how to deal with this problem. This paper will cover two solutions making use of Burst Buffers (BBs). BBs are a small size, intermediate storage layer between the computing nodes and the Parallel File Systems [8]. Since the applications in HPC systems are running parallel, the I/O phases of the different applications often occure at the same time leading to high I/O peaks [3]. Previous work by the authors of [3] show that BBs can absorb those I/O heavy bursts enabling the use of underprovisioned PFS. According to Herbein et al. [5] in HPC systems with Burst Buffers, an I/O-Aware Scheduler becomes necessary to reduce I/O Interference, which can be defined as a ”performance degradation observed by an application in contention with other for the access to a shared resource” [2]. Another solution using BBs is the Multi-resource scheduling introduced by Fan et al. [8] which schedules the user jobs not only on their CPU usage but also on the BB usage to avoid I/O Interference. The goal of this paper is to present a systematic-review of the two presented solutions, which came up with different solutions on how to include the BB usage during scheduling. The paper is structured as follows: In section 2, the necessary background information about the storage infrastructure is given. In Section 3, the two different solutions are being introduced and analyzed, while the pros and cons of those concepts are compared in Section 4. In Section 5, the conclusion is presented.

II. Background Information about Burst Buffers

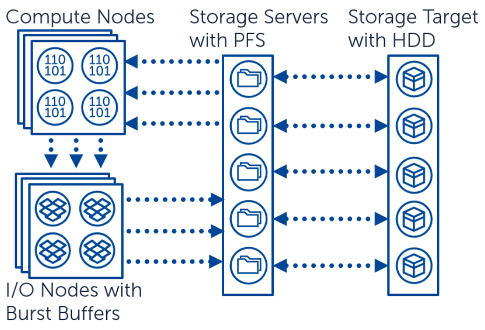


Fig. 1. Overview of the storage infrastructure.

Figure 1 shows a common setup of the storage infrastructure used in the papers [1], [3] and [9]. The compute nodes (CNs) are running concurrently and can have I/O heavy phases. If multiple CNs have I/O heavy phases simultaneously, I/O peaks occur which may overload the PFS. Since BBs have significantly higher bandwith (BW) than the PFS, because they are built from SSDs [8], they can absorb those I/O peaks coming from the CNs and turn them into a constant I/O stream. The BBs can be either attached to the CNs as a local resource or be stored in I/O Nodes, which are shared across different CNs as shown in Fig. 1. The constant I/O stream is then being passed from the I/O Nodes to the storage servers, which are equipped with a PFS. A PFS is a file system, which spreads the data on different storage targets and is designed to provide the necessary data through simultaneous, coordinated input/output operations with the storage targets, which consist of multiple HDDs [10]. The PFS is provisioned to handle the constant I/O stream from the BBs and provide the CNs with the necessary resources. [5]

III. Integration of Burst Buffers into Scheduling

In this section, the two main approaches, I/O-Aware Job scheduling as introduced in [5] and Multi-resource Scheduling introduced in [8], are presented and discussed.

A. I/O-Aware Job Scheduling

A scheduler is defined as I/O-Aware if I/O BW is being taken into account while scheduling jobs. In order to provide the scheduler with the necessary information about the I/O BW, Flux [6], a center-wide resource and job manager for next-generation centers, is used alongside BBs. The goal of this scheduling idea is to avoid I/O Interference even if the PFS is underprovisioned (the BW of the PFS is lower than the BW of the constant I/O stream coming from the BBs). [5] The idea is to model the I/O resource hierarchy using Flux as shown in Fig. 2, providing the scheduler with the information about the jobs needed BW, the BB BW as well as the PFS BW and the BW between BBs and PFS [5]. With this information the scheduler can make informed decisions to maximize the BB usage and minimize I/O Interference. As an example in Fig. 2 Job0 gets scheduled, because the BB of its CN can return a constant I/O stream up to 192MB/s and Job0 only needs 64MB/s. The switch as well as the PFS also have enough BW therefore no I/O Interference occurs. Next up, Job1 is being scheduled in parallel since its CN, the switch and the PFS have more than the needed 128MB/s BW left over. The same applies for Job2. But now there is not enough BW to schedule Job3, while Job0-2 are still running, because the local switch would need to provide 192MB/s to each Job2 and Job3, but can only has 256MB/s available. Therefore, if the scheduler is not taking I/O into consideration, from now on called I/O ignorant, I/O Interference occurs, while in an I/O-Aware environment no I/O Interference arises. However, the I/O-Aware scheduler will only schedule Job3 after Job2 finishes.

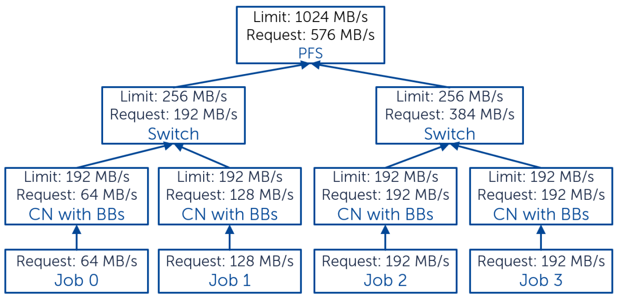
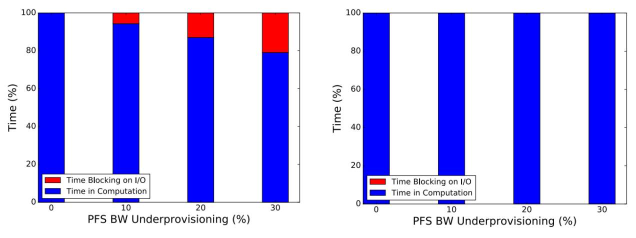


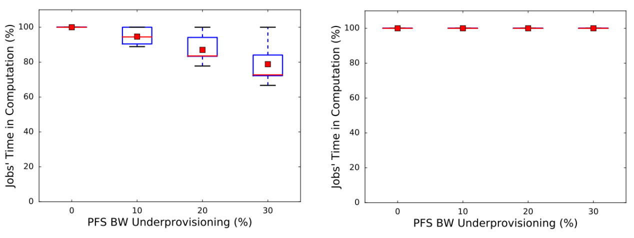
Fig. 2. Overview of an example I/O-model.

The next step is to compare I/O ignorant EASY backfilling with an extended version of EASY backfilling, which also takes the I/O-model into account, considering different critical questions. EASY backfilling is similar to FCFS, but if the next job cannot be scheduled due to a lack of resources another job will get scheduled, but only if it will not delay the original next job. [5]
The model in the comparison was built after the Commodity Technology System 1 (CTS-1), which consists of 3,888 CNs, 216GB/s per edge IB switch, 70GB/s PFS with perfect provisioning and a 432 GB/s core switch pool [5].



(a) I/O ignorant (b) I/O-Aware

Fig. 3. Percentage of total time spent by the entire CTS-1 cluster in computation and blocking on I/O [5].



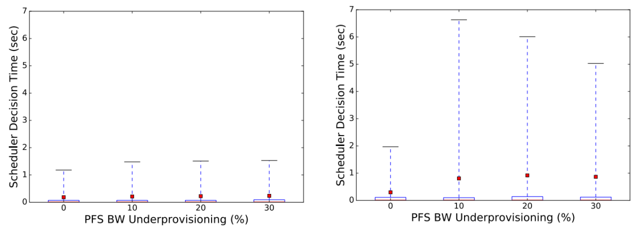
(a) I/O ignorant (b) I/O-Aware

Fig. 4. Variability of individual jobs' time spent in computation [5].

1) Impact on Total Performance: In Fig. 3 the time in computation (blue) and the time blocking on I/O (red) is visualized for the I/O ignorant and the I/O-Aware scheduler considering four different levels of underprovisioning. For a perfect provisioned PFS, no job is blocking on I/0, but if the PFS is increasingly underprovisioned up to a total of 30%, the time blocking on I/O is increasing up to 20% with an I/O ignorant scheduler, while the I/O-Aware scheduler is not scheduling jobs which causes I/O Interference resulting in 100% compute time and therefore better performance.

2) Impact on Individual Job Performance: The impact on individual job performance yields similar results to the impact on total performance. Figure 4 shows a box plot, where the red square represents the arithmetic mean, the red line the median, the bottom line of the blue box the 75th percentile (this means, that 75% of the data lies below the line and the corresponding 25% above the line), the top of the box the 95th percentile and the huskers on the bottom and on the top visualize the 5th (bottom) and 95th (top) percentile [5]. While in Fig. 4(b) each job is never blocking on I/O, in Fig. 4(a) the time each job is blocking on I/O is increasing with the levels of underprovisioning and is on average 20% with an underprovisioning of 30%, because of the missing I/O-Awareness.

3) Impact on Scheduling Decision Time: In the box plot in Fig. 5(a) it is visualized that the level of underprovisioning is almost irrelevant, because 75% of the decision times lie below 0.07 seconds and 95% of the decision times lie below 1.43 seconds [5]. As for the I/O-Aware scheduling in Fig. 5(b) the decision time lies for 75% of the jobs under 0.12



(a) I/O ignorant (b) I/O-Aware

Fig. 5. Scheduler decision time distributions [5].

seconds and for under 95% of jobs in the range of 1.97s and 6.64s [5]. While the median in Fig. 5(b) is almost 0 for every provisioning level, the arithmetic mean is increasing for the underprovisioned PFS and takes on average 1 second compared to about 0.1 seconds for the I/O ignorant scheduler. The reason is that the I/O-Aware scheduler has to additionally check the I/O model every time, which can lead to increasing variability, especially if either data or I/O heavy jobs are followed by many smaller jobs [5].

4) System Efficiency: This paper showed that for I/O-Aware scheduler the system performance can be increased up to 29% at the cost of an up to 52% longer time between the job submission and the job completion in the worst case [5].

B. Multi-resource Scheduling

In this section, we will analyze and review the contents of [8], where the scheduler BBsched is introduced, which takes multiple resources into account similar to I/O-Aware scheduling. BBsched is developed as a plug-in into the base scheduler of HPC systems to comply to the systems policies. Therefore, BBsched will consider the original policy of the system as well as specified resources, e.g. BBs. With schedulers returning solutions for different single-objective optimizations such as CPU usage or BB usage, BBsched will return a pareto set [1] for Multi-objective optimization (MOO). This means that our plug-in BBsched will return the most efficient solutions for different optimizations like CPU/BB usage and the system manager can afterwards decide which solution fits the system the best. BBsched can be split into two parts: a window-based scheduling and the MOO solver. [8]

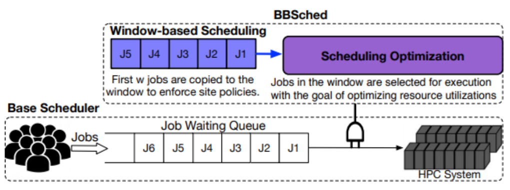

BBSched
Window-based Scheduling
J5 J4 J3 J2 J1
Scheduling Optimization
First w jobs are copied to the window to enforce site policies.
Jobs in the window are selected for execution with the goal of optimizing resource utilizations
Base Scheduler
Jobs
Job Waiting Queue
J6 J5 J4 J3 J2 J1
HPC System

Fig. 6. The overview of BBsched [8].

1) window-based scheduling: The idea of window-based scheduling as shown in Fig. 6 is that the first `w` jobs in the job waiting queue get copied into the window. This allows our BBsched to consider the systems policies and keep the order of the base scheduler as similar as possible. [8]

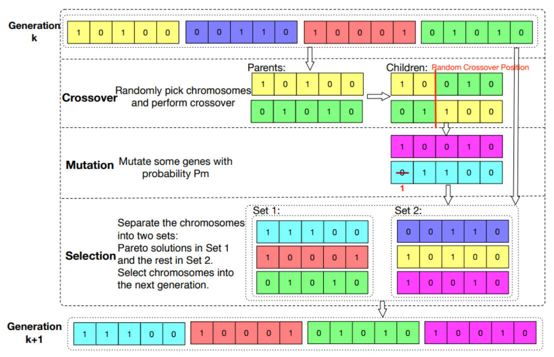

Generation k
Crossover
Randomly pick chromosomes and perform crossover
Parents:
Children:
Random Crossover Position
Mutation
Mutate some genes with probability Pm
Selection
Separate the chromosomes into two sets: Pareto solutions in Set 1 and the rest in Set 2. Select chromosomes into the next generation.
Set 1:
Set 2:
Generation k+1

Fig. 7. The MOO solver algorithm [8].

2) MOO solver: Figure 7 shows a genetic, iterative algorithm to approximate the real pareto set in much shorter time. BBsched is randomly initialized with `P=4` chromosomes. Each chromosome represents a scheduling decision, where the genes represent if a job is being scheduled or not. In the crossover phase, random chromosomes will be picked to perform a crossover at a random position visualized by the red line. Afterwards, the chromosomes genes from the crossover will mutate with a low probability p_m in order to introduce diversity. The selection phase will split up the new and old chromosomes in a pareto set called set1 and the rest will go to set2. A solution will go into the pareto set, if by improving one objective, other objectives would worsen. For the next generation, all chromosomes in set1 will be chosen and if necessary the remaining spots will be filled up with the newest chromosomes in set2. This process will be repeated `G` times. While p_m is normally set below 0.1%, the selection of the number of generations `G` and population size `P` is a trade-off between performance/accuracy and scheduling time with the best trade-off for G=500 and P=20 [8].

3) Evaluation: The evaluation is done with real workload traces collected from Cori at National Energy Research Scientific Computing Center (NERSC) using Slurm as scheduler with 12.076 CNs and 1,8PB BBs. In addition four workloads S1-S4 will be selected from the real one, where in S1 and S3 50% of the jobs request BBs and in S2 and S4 75% request BBs, while in S1 and S2 the BB request is greater than 5TB and in S3 and S4 greater than 20 TB. [8]
The workloads are evaluated using different schedulers, like EASY backfilling as baseline scheduler, BBsched, constrained_BB and constrained_CPU, where the only objective is to maximize BB/CPU usage and weighted methods, where node/BB usage gets weighted 50%/50% or 80%/20% in the weighted_CPU and weighted_BB variations. [8]
As shown in Fig. 8 with increasing BB BW and usage, BBsched improves the BB usage, because it can provide a pareto solution for multiple objectives like BB/CPU usage and still yields desirable results for other objectives, while

[1]A Pareto set is a set of optimal solutions, where no objective can be improved without worsening another objective [8].

other schedulers can only provide one solution. Therefore, BBsched is more likely to find a better solution. Since BB and node usage are correlated, most of the used scheduler only optimize one objective at the risk of worsening the other. However, BBsched is able to yield desirable results for both objectives. The comparison of average job wait time in Fig. 9 is also in favor of BBsched especially for higher BB requests, because our Multi-resource scheduling algorithm is avoiding I/O Interference resulting in no time blocking on I/O resources, while the BB/node usage for other schedulers is not as high resulting in longer wait times. Therefore, BBsched improves the reduction of average wait time compared to the baseline scheduler by up to 33.4% in Cori S3. Such increasing wait times can not be prevented, because the hardware is not able to cope with such enormous I/O demands more efficiently. However, BBsched is able to lower the average job wait time significant compared to other schedulers. All in all, BBsched outperforms the other scheduler in every metric and enhances the system performance up to 41% for the baseline and 20% for the weighted scheduler. [8]

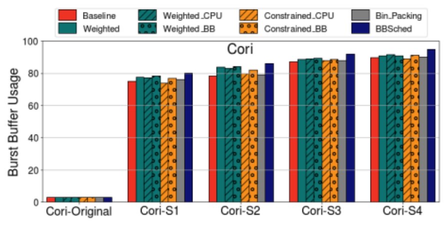


Fig. 8. BB usage on Cori

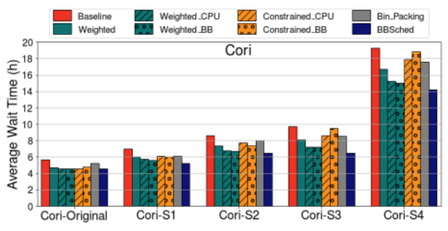


Fig. 9. Average job wait time on Cori

IV. Discussion

While paper [5] from Herbein et al. mostly focuses on creating a model for schedulers to be I/O-Aware, the authors of [8] present a Multi-resource scheduler, ultimately leading to longer job wait times, but far better system performances in HPC centers. The I/O model introduced in [5] is a simple, cost efficient solution that improves the performance well, but it forces the system to use Flux and is therefore limited in its use. BBsched on the other hand is more complex with multiple use cases. Since it functions as plug-in that complies with the systems policies, its only requisite are BBs, which are becoming more popular in HPC systems. Especially future related, BBsched is a great scheduler, since it is easily extensible for more resources that need to be taken into account. However, to maximize efficiency on a system, the necessary parameter for BBsched need to be tuned for every use case. Additionally, a solution has to be chosen from the returned pareto set for which a system manager has to be implemented to decide which solution fits the HPC systems policy the best. However, evaluation itself is not prove enough that these concepts will yield desirable results on HPC systems with different workloads.

V. Conclusion

Since HPC systems have to cope with I/O heavy applications, the underprovisioned PFS may result in I/O Interference and therefore wasted CPU time. In this paper two different approaches that based their work on BBs, which help avoid those I/O Interferences, were analyzed, reviewed and compared. Both schedulers considered the I/O BW leading to longer job wait times, but better overall system performance. This shows that I/O-Aware scheduling in HPC systems is crucial to improve performance especially with even heavier I/O applications being used in the future.

References

[1] Guillaume Aupy, Olivier Beaumont, and Lionel Eyraud-Dubois. What size should your buffers to disks be? In 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 660–669. IEEE, 2018.
[2] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. In IPDPS - International Parallel and Distributed Processing Symposium, pages 155–164, Phoenix, United States, May 2014.
[3] K. Tang, P. Huang, X. He, T. Lu, S. S. Vazhkudai, and D. Tiwari. Toward managing HPC burst buffers effectively: Draining strategy to regulate bursty I/O behavior. In 2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pages 87–98, Sept 2017.
[4] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich, Wei Tang, Jia Wang, and Zhiling Lan. I/o-aware batch scheduling for petascale computing systems. In Cluster Computing (CLUSTER), 2015 IEEE International Conference on, pages 254–263. IEEE, 2015.
[5] Stephen Herbein, Dong H Ahn, Don Lipari, Thomas RW Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, pages 69–80, 2016.
[6] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz. Flux: A Next-Generation Resource Management Framework for Large HPC Centers. In Proc. of the 10th International Workshop on Scheduling and Resource Management for Parallel and Distributed System (SRMPDS), Sep. 2014.
[7] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, and A. Moody. IO-Cop: Managing Concurrent Accesses to Shared Parallel File System. In Proc. of 2014 43rd International Conference on Parallel Processing Workshops (ICCPW), Sept 2014.
[8] Yuping Fan, Zhiling Lan, Paul Rich, William E. Allcock, Michael E. Papka, Brian Austin, David Paul. Scheduling Beyond CPUs for HPC. In Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, pages 97-108, June 2019.
[9] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, Carlos Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. On the role of burst buffers in leadership-class storage systems, 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), pages 1-11, 2012.
[10] Philip H. Carns and Walter B. Ligon III and Robert B. Ross and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters, 4th Annual Linux Showcase & Conference (ALS 2000).

Introduction to malleability

Lorenz Krakau
Bachelor's Degree Program (Computer Science)
Technische Universität München
ge93mok@mytum.de

***Abstract*—Job scheduling on High Performance Computing (HPC) systems is becoming more complex as these systems increase in complexity and size. Assigning nodes statically and exclusively to jobs is common in HPC, but can lead to under-utilization of the system's overall computational capabilities. Malleable jobs are flexible in their resource allocations, providing the resource and management system (RJMS) with new options to maximize utilization of a systems capabilities. This work gives an introduction to the topic of malleability in the context of job scheduling, by reviewing three different papers that each explore different aspects of malleability; exploring how development of malleable applications can be realized using a proposed MPI extension, how malleability can be exploited by the scheduler to improve average slowdown and response time and how malleability can be exploited to reduce I/O contention for applications with periodic I/O phases.**

***Index Terms*—scheduling, malleability, HPC, MPI, job scheduling, batch scheduling, I/O contention**

I. Introduction

With the increasing demand for computational resources, whether for public research or for commercial applications, HPC systems are becoming more important and powerful. Clusters, systems made up of hundreds of individual computers, so called nodes, are commonplace nowadays. Job scheduling on these systems is usually space-sharing and static; jobs are assigned to a set of nodes once scheduled, and do not change in the amount of assigned nodes. This however may leave nodes free or underused when a job cannot utilize all the computational resources it is given. Presumably that is why recently, malleability is receiving more attention in the HPC space. Malleability refers to the ability of a process to adapt to a changing number of resources (usually nodes) while it is running [1]. Another important term is moldability, describing a jobs ability to adapt to an assigned amount of resources only at initialization [4]. The next section will cover how development of malleable applications can be realized [1]. Section III covers how malleable and moldable jobs can be exploited to improve average response time and slowdown, while still supporting non-malleable and non-moldable jobs [2]. Section IV will examine how I/O conflicts can be avoided by leveraging malleability for applications with periodic I/O phases [3].

II. Realizing Malleable Applications

A. Motivation

Message Passing Interface (MPI) is a commonly used API for developing HPC applications. A typical HPC MPI application consist of multiple processes of the same binary, running on different nodes and communicating via an abstract communicator object by passing messages between processes. Since version 2.0 MPI allows applications to spawn new processes at runtime. This however is not commonly used, since spawning new processes blocks the calling process(-es), until the new process(-es) are spawned, which can take several seconds. Additionally processes created this way usually run within the same resource allocation, so no expansion of resources will occur. To realize malleability an API without these issues is needed. [1]

B. Proposed alternative

To facilitate development of malleable applications, [1] proposes an extension to the MPI API consisting of four new operations. Instead of the application requesting changes in resources, the scheduler will assign resources to the application. These new instructions allow the RJMS to spawn new processes *before* notifying the applications, thus hiding the latency of process creation and not blocking the application. [1]

```
function reconfigure:
  MPI_PROBE(&adapt, &status)
  if adapt:
    # Resource manager wants a reconfiguration
    MPI_COMM_ADAPT_BEGIN(&comm, &new_world_comm)
    switch status:
      # communicate with the other processes
      case JOINING: ...
      case STAYING: ...
      case LEAVING: ...
    MPI_COMM_ADAPT_COMMIT()
function main:
  MPI_INIT_ADAPT(&status)
  if status == JOINING:
    # we are spawned by the resource manager
    reconfigure()
  while not finished:
    reconfigure()
    ... # Do some computations
  MPI_FINALIZE()
```

Listing 1. Pseudocode example of an MPI application utilizing the proposed operations.

The new operations are *MPI_INIT_ADAPT*, *MPI_PROBE_ADAPT*, *MPI_COMM_ADAPT_BEGIN* and *MPI_COMM_ADAPT_COMMIT* [1].

Listing 1 shows the skeleton of an MPI application utilizing these. *MPI_INIT_ADAPT* initializes the MPI environment, and returns whether this process, was created when the application was started or due to a reconfiguration initiated by the resource manager. During execution, all processes of the application regularly call *MPI_PROBE_ADAPT* to probe whether the resource manager is reconfiguring this applications resource allocation, in which case *adapt* will be set to true and *status* will contain whether this process is joining, staying or leaving. After a process is notified of a pending reconfiguration (or when it is joining) it enters an adaptation window delimited by *MPI_COMM_ADAPT_BEGIN* and *MPI_COMM_ADAPT_COMMIT*. During this window two communicators are provided; *comm* connecting all pre-existing with all newly spawned processes, while *new_world_comm* excludes leaving processes. After the window closes the global communicator will be set to *new_world_comm*. [1]

To hide the latency of spawning processes so as to not block the already running processes for the time it takes to spawn new processes, processes are created without waiting for the existing processes to call *MPI_COMM_ADAPT_BEGIN*; the joining processes simply wait on the existing processes to enter the communication window during which they will communicate to coordinate the adaptation. [1]

In comparison to the the Master-Worker Pattern, where the Master process spawns Worker processes and delegates work to them, this avoids the communication and I/O bottlenecks that arise through the dependence on a singular master process for communication and coordination. Furthermore the Master process limits the complexity and size of a computation or simulation, since it needs to keep the whole state, being limited by the memory available to the Master process. Furthermore computation cannot continue when the Master process fails, thus hurting fault-tolerance. Applications utilizing this the proposed MPI Extension do not have to resort to the Master-Worker Pattern to realize malleability, thus avoiding these issues. [1]

Benchmarking the introduced operations on the SuperMUC HPC system confirms the success of the latency hiding approach, with the probe operation exhibiting latencies under 12ms and the adapt begin operation operation exhibiting latencies under 0.5 seconds with 512 processes. [1]

III. Exploiting malleability in Scheduling

In [2] a slowdown-driven scheduling policy (SD-Policy) leveraging malleability is presented, which is able to handle both static and malleable jobs. It refers to a broader definition of malleability, split into two levels. The first level being malleability in terms of processors/threads assigned to a job on a single node, and the second being malleability in terms of the number of nodes assigned to a job. The presented scheduling policy only concerns itself with the former, which can also be considered a form of co-scheduling. Here jobs' node allocations do not change by whole nodes, instead nodes get shared by different jobs, thus applications do not have to be able to handle getting whole nodes taken away from them or new nodes assigned to them by the resource manager. [2]

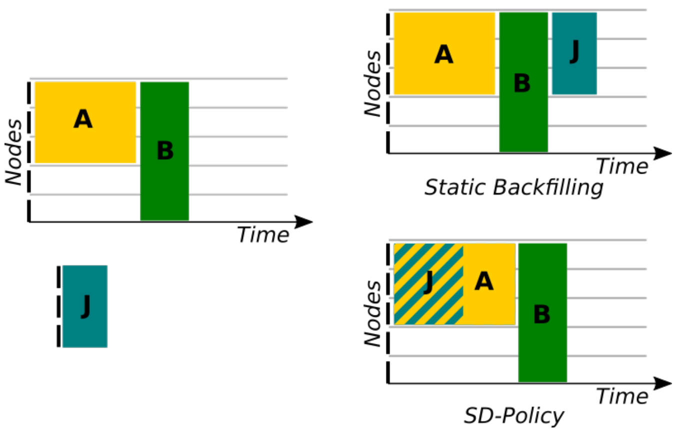


Fig. 1. Simplified example showing the scheduling of a new job *J* with static backfilling and the SD-Policy's malleable approach. Each rectangle represents a job, its height representing the job's requested amount of nodes and its width corresponding to the job's expected runtime.

This policy is implemented on top of the DROM interface, which integrates with the SLURM job scheduler and the OpenMPI implementation of the MPI API to efficiently partition a nodes resources between multiple jobs running on that node [6]. The SD-Policy presented in [2] is based on backfill scheduling. It first tries to schedule a new job statically, that is only on free nodes. If not enough nodes are free however, and the new job is malleable, then estimates for the end time were the process to be scheduled statically, and the end time were the process to be scheduled with malleability are calculated. If the latter is closer to the present than the former, the job mate selection algorithm gets invoked. This algorithm is tasked with finding the best job mates, which will share the nodes they are running on with the new job. Because this problem is NP-complete, heuristics are used to find a reasonably good selection of job mates. The algorithm tries to find the job mates that would experience a minimum cumulative slowdown, where slowdown of a single job is defined as:

$$(wait\_time + increase + req\_runtime)/req\_runtime \quad (1)$$

Where *increase* is the estimated increase in runtime the job mate would experience due to sharing its node with the new job. *req_runtime* is the requested runtime of the job mate. [2]

The effect of normalizing the wait time and runtime increase by the requested runtime for the slowdown penalty is best illustrated with an example: A job with a requested runtime of a week, will have the same slowdown penalty if delayed by one day, as a different job with a requested runtime of 7 hours if delayed by one hour. The underlying assumption is that both delays are equally acceptable to the users submitting the respective job. Additionally the authors propose three constraints. The first one limits the maximum slowdown per job to a dynamic or a static value *MAX_SLOWDOWN*. The second constraint is defined as:

$$\sum_{i=0}^{n} w_i = W \tag{2}$$

Where w_i is the number of currently allocated nodes for each selected mate and *W* is the number of nodes the new job requests. Our example schedule in n Fig. 1 fulfills this constraint, as the sum of allocated nodes of our selected mate(s) $\{A\}$ is 3, the same amount of nodes as our new job *J* requested. This constraint simply ensures that job mates share all their nodes with the new job. This is to ensure the jobs run balanced even when they are not able to balance load dynamically, which is a common case for HPC applications. The third constraint is that a new job finish before its selected job mates, since if that is not the case, the new job would delay jobs scheduled after the mates. This also avoids creating unbalance in case the new job cannot balance load dynamically [4]. [2]

Evaluating this technique in simulated workloads and on a real HPC-System workload, for different *MAX_SLOWDOWN* values including a dynamically determined average of system slowdown, results show a response time reduction of up to 50% and a slowdown reduction of 70% and an improvement of makespan and energy reduction of 7% and 6% respectively in the real workload test. [2]

IV. Exploiting malleability to avoid I/O Contention

[3] focuses on exploiting malleability to avoid I/O contention, for Single-Program-Multiple-Data (SPMD) applications with periodic I/O phases. The main idea is to predict I/O phases and then use malleability to affect their timing in order to prevent two or more I/O phases occuring at the same time. This is accomplished via two strategies; phase shifting and period coupling. The former uses malleability to speedup a job temporarily by allocating more resources to the job, in order to shift the I/O phase. The latter uses malleability to speedup a job permanently, such that its I/O phase is *period coupled* with another job, meaning their periods are equal in length. If two or more jobs are ”period coupled” and sufficiently out-of-phase (phase shifted) their I/O phases will never interfere with each other. [3]

In order to predict I/O phases MPI's I/O operations are wrapped and monitored to gather metrics on the applications I/O behavior, specifically it's I/O period and phase. In order to predict how a change in allocated resources will affect the performance of the application and thus its I/O phase timing, samples for different resource configurations of the application need to be gathered. This can be done off- or on-line. Off-line it is gathered by benchmarking and analyzing the application under various different configurations. On-line it is gathered by applying multiple different configurations to the application after it has started and measuring their effects on application performance. Only a few configurations are tested in both the off- and on-line method and then interpolated to build the performance model of the application. [3]

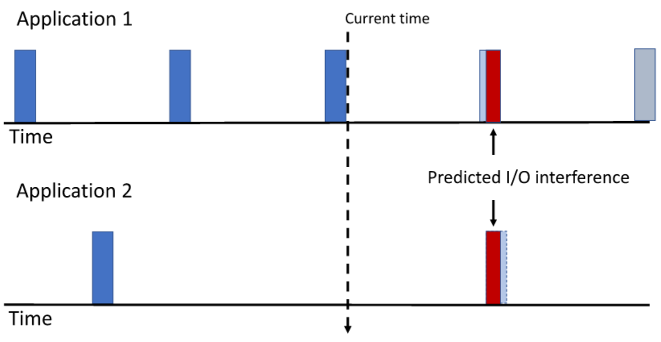


Fig. 2. Two applications which are predicted to encounter I/O contention in their next I/O phase(s). Bars represent I/O phases, the predicted I/O contention is marked red.

Depending on the application more or less samples are needed in order for the model to be sufficiently accurate. This model can now be used to predict I/O phases of applications with periodic I/O phases and thus to predict whether two applications' I/O phases will interfere with each other, by checking if the predicted I/O phases overlap. In Fig. 2 the I/O phases of Application 1 and Application 2 are predicted to interfere on the next phase. [3]

A. *Phase shifting strategy*

The predicted I/O interference depicted in Fig. 2 can be avoided by making use of the phase shifting strategy. This strategy increases the amount of processes allocated to the application temporarily, increasing the applications performance and thereby reducing the time left until its next I/O phase, in order to shift this I/O phase closer to the present time.

Which application will be phase shifted depends on its I/O period. If the I/O periods are not very different, but not equal either, the application with the shorter period will be chosen for phase shifting because after the phase shifting its I/O phase will occur almost immediately before the other application's I/O phase, thus if the phase shifted application has the smaller I/O period it will take longer for another I/O interference to occur. If the I/O periods are very different or equal however, the one whose I/O phase occurs next will be chosen, so as to minimize the required shift Δt. [3]

The overhead of the reconfiguration in the phase shift operation is also considered in the calculation of the actual shift, in addition to a user configurable uncertainty parameter. For the sake of simplicity, this actual shift will be treated as similar to the required shift Δt here.

In order to shift the I/O phase by Δt, the application needs to be sped up by the required speedup S:

$$S = \frac{T_{orig}}{T_{orig} - \Delta t} \tag{3}$$

where T_{orig} refers to the original predicted time of the next I/O phase of the respective application. Using the calculated performance model, we can calculate the number of new processes ΔP required to realize the speedup S. How this

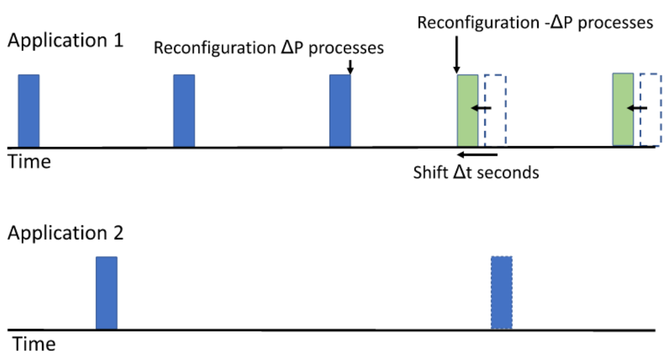


Fig. 3. Example demonstrating the phase shift strategy. Green bars represent the affected I/O phases.

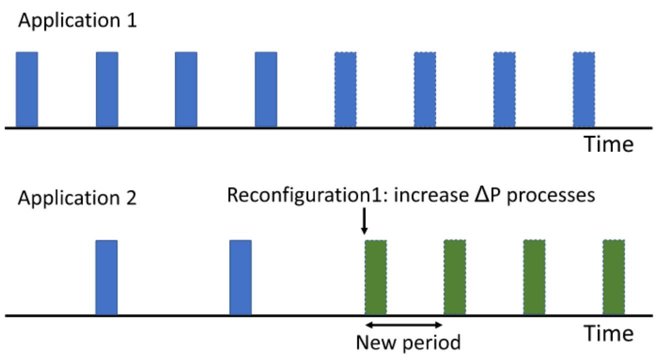


Fig. 4. Example demonstrating the period coupling strategy. Green bars represent the affected I/O phases.

is done depends on the specific performance model. If the amount of new processes ΔP is not available or the speedup S cannot be realized no matter the amount of new processes, the reconfiguration is simply canceled. Otherwise the reconfiguration will occur, and after certain time, at minimum the time until the end of the next I/O phase of the reconfigured application, the application will be reconfigured again, this time back to the previous configuration, so as to not affect the I/O period permanently, but only perform a phase shift. [3]

B. *Period coupling strategy*

In order to prevent I/O contention long term, the period coupling strategy can be used which permanently reconfigures applications to make their periods more similar, or in the ideal case equal. This way the need to repeatedly phase-shift, can be eliminated for some cases. [3]

If needed to prevent I/O interference when the I/O periods *and* the I/O phases are equal the phase shifting strategy is used to diversify the I/O phases. Two applications with very similar or equal I/O periods and different I/O phases will have little or no I/O interference with each other. Thus by using this strategy combined with the phase shifting strategy we can avoid I/O contention while minimizing reconfigurations. [3]

In order to change the I/O period P an application to a new shorter I/O period P_{new} similar to the period of another application we calculate the required speedup as P/P_{new} and use this together with the performance model to calculate the number of new processes required, similar to the last step of the phase shift strategy. If the speedup is not achievable due to insufficient amount of available processes or because no amount of new processes could realize the required speedup, the reconfiguration is canceled, similar to the phase shift strategy. [3]

A similar technique would be possible even for non-malleable jobs, but it would require artificially slowing down the job, since that can be achieved by reducing the performance of each process the job is composed of, without having to change the number of processes itself. This has the obvious downside of reducing the affected job's performance. The phase shifting and period coupling techniques covered here will only ever attempt to increase the performance of a job, never reduce it, except for when reverting to the previous configuration after a phase shift. [3]

Evaluating these techniques, with a scientific application, solving systems of linear equations iteratively, with periodic I/O phases on an HPC cluster yields promising results, reducing application I/O times up to 49%, compared to a 39% reduction using a blocking I/O interference prevention technique, which does not leverage malleability [5]. [3]

V. Conclusion

This work gave an introduction into malleability, ranging from the efficient realization on the application development side, to utilizing sub-node level malleability and moldability in job scheduling to improve slowdown and average response time dramatically. Last but not least a more specific exploitation of malleability with the goal of eliding I/O conflicts was covered. [1]–[3] The presented techniques each cover very different aspects of malleability and all provide promising results. Integrating these different techniques in a holistic manner might be interesting for future work.

References

[1] Comprés, Isaías and Mo-Hellenbrand, Ao and Gerndt, Michael and Bungartz, Hans-Joachim, "Infrastructure and API Extensions for Elastic Execution of MPI Applications", Proceedings of the 23rd European MPI Users' Group Meeting, Edinburgh, United Kingdom, 2016.
[2] D'Amico, Marco and Jokanovic, Ana and Corbalan, Julita, "Holistic Slowdown Driven Scheduling and Resource Management for Malleable Jobs", Proceedings of the 48th International Conference on Parallel Processing, Kyoto, Japan, 2019.
[3] David E. Singh and Jesus Carretero, "Combining malleability and I/O control mechanisms to enhance the execution of multiple applications", Journal of Systems and Software, 2019.
[4] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik and Parkson Wong, "Theory and Practice in Parallel Job Scheduling", Job Scheduling Strategies for Parallel Processing, 1997
[5] Isaila F., Carretero J. and Ross R., "CLARISSE: a middleware for data-staging coordination and control on large-scale HPC platforms", Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CC-Grid), 2016.
[6] Marco D'Amico, Marta Garcia-Gasulla, Víctor López, Ana Jokanovic, Raül Sirvent and Julita Corbalan, "DROM: Enabling Efficient and Effortless Malleability for Resource Managers", Proceedings of the 47th International Conference on Parallel Processing Companion (ICPP '18), ACM, New York, NY, USA, 2018.

Machine Learning Based Scheduling

Ruben Bachmann
Master's Degree Program (Computer Science)
Technische Universität München
ga53zaz@mytum.de

***Abstract*—Both HPC hardware and software have become increasingly complex over the years, which caused job scheduling to become even more difficult. Since good scheduling is essential for efficiently executing jobs, a lot of work has been put into improving schedulers. Machine learning in particular has played a large role in the development of new HPC schedulers, as it offers new and often times more efficient ways to approach the problem. This is especially true considering recent advances in machine learning techniques. In this paper we will give an overview over machine learning tools and techniques that have been developed to improve job scheduling. We will introduce their basic functionality and compare them to traditional approaches, both in terms of complexity and performance. At the end of this paper we will also provide a small overview over possible future advancements in this area, and evaluate the usefulness of machine learning for different aspects of job scheduling.**

I. Introduction

With hardware and software getting more complex over the years, efficient scheduling has become increasingly difficult. This is especially true for large scale high performance computing (HPC) systems, which have to deal with a large number of jobs with different characteristics. Since the quality of scheduling has a strong impact on the performance of the system, many different types of schedulers have been developed, trying to maximize performance. Each of these schedulers evaluate multiple metrics of the jobs that need to be scheduled and try to find the optimal allocation of resources. Depending on the implementation of the scheduler, different goals like lowering latency might be achieved, which can also be used to compare the performance of schedulers for certain types of jobs [1].

However most traditional HPC job schedulers have a common problem, which is that they do not react well to changes in the characteristics of jobs, like their expected runtime or resource requirements [2]. The reason for this is that they usually work in a static way, and require manual configuration changes in case a shift of job characteristics happens. The size of modern HPC systems in combination with the large number of jobs that need to be scheduled, often times make performing these manual changes unfeasible for system administrators [2]. Therefore an automated solution for this type of problem is required, since it would not only decrease the workload for system administrators, but could also improve overall scheduling performance.

A possible solution for this was found in machine learning based scheduling algorithms. Machine learning with its various sub-types like supervised-, unsupervised- or reinforcement learning [3] has become a powerful tool for many different types of applications. Often times machine learning makes it possible to find adaptive solutions for problems, which might not always be possible with traditional methods. This is also the case for modern scheduling problems. Machine learning based scheduling algorithms exist in multiple forms. One of those forms consists of algorithms which use machine learning to schedule jobs based on traditional input parameters as described in [2] [4] [5]. Another form is based around algorithms that use machine learning to increase the quality of input parameters in order to improve scheduling results. Some examples for this can be found in [6] [7] [8].

Machine learning based parameter tuning is interesting for traditional scheduling algorithms as well, since the improved input parameters can also help those algorithms perform better. Although some solutions for this have been found [9], it is generally desirable to have more accurate input parameters, since they give schedulers more accurate information about the workload.

In this paper we will present some machine learning techniques that have been developed to deal with the previously mentioned problems. We will compare them to traditional approaches and give some insight into possible future developments.

The rest of the paper will be structured as follows. Chapter 2 gives an introduction to machine learning basics, including supervised and unsupervised learning, as well as neural networks. In chapter 3 we will present some machine learning based scheduling and parameter tuning algorithms. Chapter 4 compares these machine learning based algorithms to traditional ones, both in terms of performance and complexity. In chapter 5 we give an outlook on a few possible future developments in this area. Finally chapter 6 Provides a conclusion to the content of this paper.

II. Machine Learning Basics

Machine learnig with its various sub-types is a much studied field, which has applications in many areas of research. Because of that, a lot of different approaches to machine learning have been developed over the years. In order to understand the following contents of this paper, it is necessary to have some basic knowledge of some of the most widely used machine learning techniques. We therefore introduce the basics of these techniques to give a short overview over the field, and to make it easier to understand its use cases for modern HPC scheduling.

A. Supervised Machine Learning

The first approach that needs to be discussed is supervised machine learning. This technique is often used for classification problems, and requires a labeled training data set, for which the required classification outcomes are already known [10]. Based on this labeled data set, the machine learning algorithm can be trained to classify data of equal or similar structure, for which the required labels are yet unknown.

In the first step of the training process, the algorithm is given the training data without its labels as input [10]. Based on the initial state of the algorithm, labels are calculated for this input. The aforementioned initial state of the algorithm usually consists of randomly chosen starting parameters, which causes the initial labeling not to be accurate in most cases [3]. In the second step the calculated labels are then compared to the known labels. Depending on the size of the classification error changes are made to the internal parameters of the algorithm, in order to improve the labeling of future runs [3]. This process can be repeated multiple times, until the quality of the labeling produced by the algorithm converges to a certain quality level, which depends on the quality of the algorithm and the training data set [3].

Supervised machine learning can once again be split up into multiple categories. These include decision trees, which can be used for classification problems based on a tree like structure, linear regression, which aims to find connections between different variables of a data set, or bayesian classifications [10]. These techniques will not be discussed in detail here, however interested readers can find some more examples in [11].

B. Unsupervised Machine Learning

Another widely used machine learning approach is unsupervised learning. Contrary to supervised learning, this technique does not require a labeled training data set. Instead the properties of the input data are used to identify clusters of similar data points, which can be useful for different types of applications [3]. In many cases not requiring a labeled training data set can be a deciding advantage of unsupervised machine learning over its counterpart, since these kinds of data sets might not always be available for certain types of problems and applications.

The training process of unsupervised machine learning algorithms also consists of two steps. In the first step unlabeled data is given to the algorithm as input. The algorithm then groups the datapoints into clusters, based on their properties [3]. In order to do this the distance between the values of datapoint properties has to be calculated, which can be done in different ways, depending on the implementation [12]. The initial number as well as the properties of cluster centers often times is randomly initialized, and is then adapted over multiple runs. The goal of this is to maximize the similarity of datapoints within clusters, while at the same time not creating too many clusters, as this would limit the usefulness of the found clusters. This once again can be calculated using different metrics [3].

Unsupervised learning can further be split up into categories like clustering and hierarchical learning [12]. Since detailed understanding of them is not required for the presented papers, these will not discussed here.

C. Neural Networks

Neural networks are another popular machine learning technique. As the name suggests, neural networks try to replicate the structure of a human brain. This is achieved with multiple layers of nodes that are connected to each other. These layers modify the input data using activation functions. The result of the computation is stored in the final layer of the network [3]. This structure makes neural networks a useful tool for multiple types of applications.

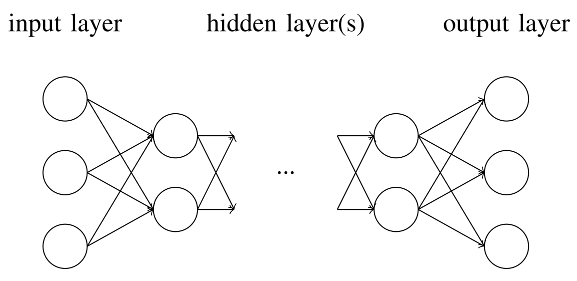


Fig. 1. Graphical representation of a neural network

Neural networks work by inserting the input data into the first layer of neurons, the so called input layer. The input layer is connected to the first of potentially many so called hidden layers, along which the data is passed. At each of these layers, an activation function is applied to the data, transforming it at each step through the network [13]. Depending on the structure of the neural network and its use case, these activation functions can differ from network to network. This process repeats until the final layer, the so called output layer, is reached. Here the result of the computation can be obtained from [13]. Fig. 1 provides a simple graphical representation of a neural network, showing the input-, hidden- and output layer as well as the connections between them along which the activation functions are applied.

Depending on the use case, neural networks require different structures and activation functions. A good example for this are classification applications, which typically contain as many nodes in the output layer as there are classes. Neural networks can also be differentiated by their number of hidden layers, with deep neural networks usually containing at least a double digit amount of them [13]. Certain neural network structures allow for useful applications like autoencoders, which can be used for dimensionality reduction, a form of input parameter tuning [12].

III. Machine Learning Based HPC Scheduling Approaches

In this chapter we specifically focus on machine learning applications for HPC scheduling. These applications can be split into two major groups. First there are algorithms that directly use machine learning for the scheduling process. The second major type of applications are machine learning algorithms that are used for parameter tuning. These take traditional scheduling criteria as input and transform them into more accurate and useful inputs with machine learning. In the following we are going to show some examples of both types of algorithms.

A. Machine Learning Based Scheduling

The RLScheduler described in [2] utilizes two neural networks for its reinforcement learning approach. First, there is the policy network, which is a 3 layered kernel network responsible for scheduling the waiting jobs. This network takes the current state of the system, including the waiting jobs and their characteristics, as input, and calculates a score for each of these jobs. According to this calculated score a scheduling decision can then be made [2]. Their second network is called the value network, which is a 3 layered network. This network is trained alongside the policy network and takes the reward, representing the quality of the previously made scheduling decisions, as input. With this input the network is trained to predict the rewards of future job sequences, which is then used to improve scheduling. Depending on the needs of the system, different reward functions can be chosen by the system administrators to allow for optimal results. The scheduling quality is also further improved by filtering out certain outlier schedules during training, since these could negatively impact overall performance [2].

In order to evaluate their approach they trained their network with multiple traces and compared the scheduling results to the ones of traditional approaches. First, they determined that their neural networks converge relatively fast, meaning that good scheduling results can be obtained after few training epochs. This was achieved mostly due to eliminating outlier traces as mentioned before, since without this technique they measured much longer convergence times, which already shows some possible problems with machine learning scheduling approaches [2].

Additionally the performance of RLScheduler was evaluated for different traces and scheduling goals, including resource utilization, average job slowdown and waiting time. They conclude that for most traces and scheduling goals RLScheduler converges fast and provides better scheduling results than most traditional schedulers. This could even be achieved with relatively little computational overhead, making it feasible to use in real world scenarios [2].

The deep reinforcement agent for scheduling (DRAS) described in [4] follows a similar approach. For their scheduling algorithm they also use two neural networks. However, in this case both networks are of the same 5 layered structure, and have slightly different purposes compared to the ones of RLScheduler. The first network receives the current system state, including waiting jobs and their characteristics, as input and is responsible for assigning jobs to resources for immediate execution. The first job that does not fit into currently available resources is also marked as reserved, and will be executed as soon as possible. For this first step jobs that have waited for a long time are given a increased priority in order to avoid starvation, which they found to be a huge problem for many traditional scheduling algorithms [4].

In the second step the other neural network is tasked with backfilling, which is a strategy to assign additional jobs to resources without hindering the previously scheduled jobs. With backfilling hardware utilization and overall performance can be improved. After making these scheduling decisions the networks receive a reward depending on the quality of scheduling, which can once again be defined by system administrators. This reward is used to adjust the networks for future steps in order to further improve scheduling results. For additional insights and potentially better results they implemented two versions of DRAS, namely DRAS-PG and DRAS-DQL, using different forms of reinforcement learning for the adjustment of the neural networks. PG standing for policy gradient, while DQL is short for deep Q-learning, both of which are popular techniques [4].

For their evaluation they trained both versions of DRAS with different job traces, each time starting with easier job patterns and introducing more complicated ones later on in order to improve training results. They then compared the scheduling results of DRAS to different traditional approaches for performance evaluation. In their evaluation they show, that convergence time greatly benefits from their approach of starting with easier schedules and later transitioning to harder ones [4]. This shows that a lot of thought has to be put into machine learning based scheduling algorithms, not only during their design, but also for their training.

In their performance evaluation it is shown that both versions of DRAS outperform many traditional approaches in multiple metrics, including job wait time, response time, slowdown and resource utilization. Additionally it was noted that job starvation was relatively low with maximum job wait times of 16 and 20 days, compared to 170 days for some other approaches. This is especially useful for large jobs, since they usually suffer most from job starvation [4].

Most of their performance gain over traditional methods comes from the intelligent assignment and reservation of resources, which minimizes overall job wait time. They also show that the performance increase stays present during shifts in the type and intensity of scheduling tasks, thereby solving an important scheduling problem. Similar to RLScheduler this could be achieved with little computational overhead, making it useful for real world scenarios [4].

Another interesting approach to machine learning based scheduling comes in the form of RLSchert as described in [14]. Here, a reinforcement learning based scheduler is combined with a remaining runtime predictor, which allows the scheduler to have more information about the current state of the system and make changes if necessary [14]. Additionally it makes the scheduler less dependent on potentially inaccurate runtime estimations, which can hurt overall scheduling performance if they are not dealt with correctly.

The scheduler of this approach works similar to the ones described above. It takes the current system state as input and allocates the currently available resources to jobs, which also includes backfilling. In order to make the scheduling problem less complex, only a certain number of waiting jobs is considered for each scheduling step, despite the total number of waiting jobs potentially being higher [14]. In parallel the dynamic remaining runtime predictor uses real time system data to predict the remaining runtime of currently running jobs. In case large discrepancies between the original and the current prediction of a jobs remaining runtime are found, the scheduler can kill this job and replace it with another one. This usually happens if the newly predicted remaining runtime is much larger than originally thought, and the job has not already run for a long time. If this is done correctly, the average slowdown of all processed jobs decreases significantly [14].

Their evaluation of RLSchert shows, that this approach can achieve good results in both convergence time and overall scheduling quality, especially when measuring average job slowdown. For this metric, the positive effect of their kill policy was visible very well [14].

The final machine learning based scheduling approach we are going to present here is described in [5], and utilizes an entirely different technique. Instead of using a reinforcement learning based scheduler, they aimed to create scheduling policies with nonlinear functions. In order to obtain these functions a simulation procedure in combination with nonlinear regression is used. The simulation procedure receives two distinct sets of jobs. At the beginning of the simulation the jobs of the first set are executed in an arbitrary order, which simulates a random initial state [5]. After that, each of the jobs in the second set is assigned a score based on its impact on the slowdown of the rest of the jobs. From these scores nonlinear functions are computed, which can be used for scheduling as described in [5].

For their evaluation they compared the scheduling results using these nonlinear functions to traditional approaches, with a focus on average job slowdown time. It was noticed, that the function strongly prioritized jobs with earlier arrival time, a shorter expected execution time, and low resource requirements, which is quite intuitive. For both real and user estimated job runtimes the nonlinear functions outperformed traditional methods. It was additionally noted, that they benefited less from aggressive backfilling, since the original scheduling already had superior performance. With these results it can be concluded that this method works well for real world workloads [5].

B. Machine Learning Based Parameter Tuning

Another way to utilize machine learning for scheduling is parameter tuning. The reason for this is that almost all scheduling approaches require accurate input parameters, which usually consist of information about the current system state and the waiting jobs. Although some algorithms have been developed to reduce the negative impact of inaccurate parameters [9], it is therefore desirable to increase the quality of input parameters. In the following we are going to describe some machine learning applications that try to reach this goal.

The first parameter tuning algorithm we are going to discuss here is described in [8]. That paper describes the combined utilization of supervised and unsupervised learning to more accurately predict job runtimes. In the first step, a shortest edit distance matrix is calculated from a combination of the user name and the job name. This matrix measures the distance between the combined strings, and is used as a basis for the following clustering algorithm [8]. For this next step the k-means++ algorithm is used, which groups the datapoints into k clusters. This needs to be done multiple times for different values of k and starting cluster centers to obtain the best result. Based on these clusters, for each of the jobs the k-nearest neighbor algorithm is used to obtain similar jobs, with the criterias being CPU requirements and submit time. This data is then used in unsupervised machine learning to predict job times [8].

In order to evaluate their approach they implemented the running time predictor using 3 different machine learning algorithms, namely linear regression, random forest regression and SVR. Additionally the prediction quality with and without previous clustering was compared in order to further analyze the usefulness of those steps. This evaluation was done with a real world data set from CARDC [8]. As their evaluation criteria mean absolute error (MAE), average prediction accuracy (ACA) and underestimate rate (UR) were chosen. The evaluation showed, that previous clustering reduced the MAE by an average of 74 percent, with SVR producing the best results. Similarly good results were achieved for ACA and UR, with SVR producing the best results in all cases. Since SVR not only performs well but also converges quickly, this makes it the best candidate out of the tested algorithms for this application [8].

The effect of their application on scheduling quality was also tested. Here they came to the conclusion, that with the help of their framework, the average waiting time (AWT) of jobs could be reduced by around 29 percent. This shows that the improvement of running time predictions, which are usually relatively inaccurate, can help improve scheduling performance and resource utilization [8].

Just like runtime predictions, the prediction of required resources for a job tends to be quite difficult. In many cases users overestimate the amount of resources required for their job, which can be detremental for overall scheduling, and therefore also system performance. The approach described in [6] tries to improve resource requirement predictions using supervised machine learning. The overall work flow of their approach looks as follows. In the first step, the user must submit their job including their own resource requirement predictions. These predictions are then updated using supervised machine learning, and given to the scheduler for resource allocation [6]. For the training of the algorithm, a dataset consisting of 14 million datapoints was used. For each of those datapoints 8 out of originally 45 attrbutes defining resource requirements were considered [6].

Similar to the previously described approach, multiple machine learning algorithms were considered for the evaluation. These included linear regression (LR), lasso lars ic regression (LLIC), elastic net CV regression (ENCV), ridge regression (RG) and decision tree regression (DTR). The final evaluation was done on the slurm simulator using DTR, since it was the most fitting for the data. To gain further insights, jobs were additionally split up into large and small jobs, making the effect of the approach to different job sizes better understandable [6]. For the larger jobs, consisting of jobs requiring at least 4GB of memory, a huge improvement in total required execution time was observed. While a total running time of 5 days was simulated using the originally requested resources, the simulated running time using the machine learning predicted resources only was around 10 hours, which is similar to the results obtained from using the actual resource requirements. These results show, that their application is able to accurately determine required resources, and improve overall system performance by better utilizing system resources. While not being as impressive as for the larger jobs, the application was also able to achieve considerable improvements for the smaller jobs. As expected this approach not only had a positive effect on total runtime, but also on average waiting times of single jobs [6].

In summary it can be noted, that many different approaches to utilize machine learning for scheduling exist. In the following we are going to show the advantages and disadvantages of these methods compared to traditional approaches.

IV. Comparison to Traditional Approaches

The previous chapter showed, that machine learning can be a powerful tool for HPC scheduling. Although the results of the approaches we described were mostly positive, there are some drawbacks to them as well. In the following chapter we are going to discuss both the advantages and drawbacks of machine learning based approaches.

A. Advantages

One of the most noticeable advantages of machine learning over traditional approaches is the potential performance increase. This is relatively easy to see, since all the approaches described above showed considerable performance benefits in almost all metrics used for evaluation, including total processing time, average job wait time, resource utilization and more. Not only does this increase the speed at which researchers can obtain their desired results, but it also helps to better utilize HPC systems. These systems have become very large in scale and therefore expensive, both in hardware and electricity costs, which is why such performance increases generally are a desirable goal for both researchers and administrators of HPC systems.

Another advantage that was specifically mentioned by some of the presented papers, is the ability of machine learning based tools to react to changes in the size and type of workload [2] [4]. Overall they proofed to be more flexible regarding the input data, and therefore less dependent on manual intervention by system administrators. Because of the size of HPC systems and the large number of jobs that need to be processed, this property is an essential characteristic of most modern scheduling solutions.

Furthermore, machine learning based scheduling approaches can be made less dependent on the quality of their input parameters, which includes predictions for job runtimes and resource requirements. This can be observed especially well for the parameter tuning based techniques. The main goals of these applications usually specifically include the reduction of such dependencies, and according to their evaluations, these goals were achieved. Improving input parameters is helpful, since they are the basis of the resource allocation done by both traditional and machine learning based schedulers. Because of the size and complexity of both HPC systems and the jobs that need to be processed, runtimes and resource requirements can be hard to predict by humans. This can lead to incorrect predictions, and therefore decreased scheduling quality for traditional approaches.

B. Disadvantages

Since machine learning is such a large field, it can often times be difficult to find the best algorithm for the problem that needs to be solved. This is also the case for scheduling algorithms, which is why most of the presented applications compared the results of multiple machine learning techniques in order to obtain the best results. However the additional implementation and testing that is required to do this also results in a lot of additional work, while it still can not be guaranteed that the absolute best technique has been chosen. This becomes even more obvious when compared to the relatively simple implementation of certain heuristics used by some traditional schedulers. However, it must also be said, that for most of the presented applications all tested machine learning techniques performed better than traditional approaches. Therefore not finding the very best technique for the problem most of the time might not be required to achieve performance improvements. Nevertheless some performance might be lost by not testing enough techniques.

An additional problem of supervised machine learning algorithms is that they require a labeled data set for their training [10]. This can potentially be a problem, since such a dataset is not always available. Even if one exists, it must be tested if the data actually is suitable for the problem that needs to be solved, since discrepancies in value ranges between the training data set and the data that actually needs to be classified can lead to insufficient results.

During the training phase of machine learning algorithms the problem of overfitting also needs to be considered. As already shortly described above, when a machine learning algorithm is overfitted, this means that the algorithm is overly specialized to solving a problem specifically for the training data set [3]. This in turn makes the algorithm less useful for solving the general problem with arbitrary data, which reduces some of the benefits gained from machine learning. Overfitting usually happens if not enough training data is available, resulting in too many training rounds with the same limited data set. Although the risk of this happening can be reduced with techniques like splitting up the data set into a training and a validation part as shortly mentioned in [7], it is still a problem that needs to be dealt with.

Despite these disadvantages, the papers we presented clearly managed to get good scheduling results from their machine learning based approaches. It can therefore be determined, that if the difficulties machine learning introduces to the problem of scheduling are known in advance and dealt with correctly, improved results compared to traditional approaches can still be achieved.

V. Future of Machine Learning in HPC Scheduling

Predicting future research developments is always difficult, especially in such complex fields as machine learning and scheduling. However some trends can be noticed, such as the fast paced development of HPC scheduling and parameter tuning tools based on machine learning. This can be seen by the relatively large number of papers published in the past few years, some of which were presented here.

Due to the current general popularity of machine learning, as well as the constant advances in machine learning techniques, it can therefore be assumed that more HPC scheduling algorithms will be developed and improved upon in the future. Some proposals for this have already been made in existing papers.

The next large step machine learning based approaches will likely take is their implementation in more real world HPC management and scheduling systems. Among others the authors of [2] [5] have mentioned similar plans in their papers. This step is important, since quite a lot of the applications that have been described in papers have not yet been implemented in real world HPC systems.

With more applications making it into real world schedulers, additional attention to the topic could be generated, resulting in new or improved upon implementations. This could also include testing the effectiveness of new machine learning techniques for this problem field. Such improvements to their existing implementations were also mentioned by the authors of [6].

Overall the future of machine learning in scheduling therefore seems to be heading towards a more widespread use and additional developments. These could also help other research fields, since the improvement of HPC scheduling and system utilization benefits researchers of all fields that require HPC resources in some way.

VI. Conclusion

This paper gives an overview over machine learning based HPC scheduling approaches, which can be split up into machine learning based scheduling and parameter tuning. It discusses the major strengths and weaknesses of these approaches compared to traditional techniques, and shows their applicability in real world scenarios. Additionally potential future work in this area is discussed, which could further increase the capabilities of the already existing approaches. It is shown that machine learning based approaches can have large performance benefits in almost all metrics that are commonly used for the evaluation of scheduling algorithms. Similarly their capability to handle inaccurate parameters is shown. Additionally it is pointed out that changes in job characteristics are handled well by machine learning based algorithms. Major drawbacks that are shown in this paper include the necessity for training data as well as the general complexity of choosing the right algorithm. Overall it is concluded that machine learning is a promising method that can be used to solve many major scheduling problems, especially when enough time is invested into properly designing the algorithm and dealing with the problems.

References

[1] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. Scalable system scheduling for hpc and big data. *Journal of Parallel and Distributed Computing*, 111:76–92, 2018.

[2] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: an automated hpc batch job scheduler using reinforcement learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[3] Taeho Jo. *Machine Learning Foundations: Supervised, Unsupervised, and Advanced Learning*. Springer Nature, 2021.

[4] Yuping Fan, Zhiling Lan, Taylor Childers, Paul Rich, William Allcock, and Michael E Papka. Deep reinforcement agent for scheduling in hpc. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 807–816. IEEE, 2021.

[5] Danilo Carastan-Santos and Raphael Y De Camargo. Obtaining dynamic scheduling policies with simulation and machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2017.

[6] Mohammed Tanash, Brandon Dunn, Daniel Andresen, William Hsu, Huichen Yang, and Adedolapo Okanlawon. Improving hpc system performance by predicting job resources via supervised machine learning. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, pages 1–8. 2019.

[7] Eduardo R Rodrigues, Renato LF Cunha, Marco AS Netto, and Michael Spriggs. Helping hpc users specify job memory requirements via machine learning. In *2016 Third International Workshop on HPC User Support Tools (HUST)*, pages 6–13. IEEE, 2016.

[8] Hao Wang, Yi-Qin Dai, Jie Yu, and Yong Dong. Predicting running time of aerodynamic jobs in hpc system by combining supervised and unsupervised learning method. *Advances in Aerodynamics*, 3(1):1–18, 2021.

[9] Mina Naghshnejad and Mukesh Singhal. A hybrid scheduling platform: a runtime prediction reliability aware scheduling platform to improve hpc scheduling performance. *The Journal of Supercomputing*, 76(1):122–149, 2020.

[10] Vladimir Nasteski. An overview of the supervised machine learning methods. *Horizons. b*, 4:51–62, 2017.

[11] Sotiris B Kotsiantis, Ioannis Zaharakis, P Pintelas, et al. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160(1):3–24, 2007.

[12] Muhammad Usama, Junaid Qadir, Aunn Raza, Hunain Arif, Kok-Lim Alvin Yau, Yehia Elkhatib, Amir Hussain, and Ala Al-Fuqaha. Unsupervised machine learning for networking: Techniques, applications and research challenges. *IEEE access*, 7:65579–65615, 2019.

[13] Rene Y Choi, Aaron S Coyner, Jayashree Kalpathy-Cramer, Michael F Chiang, and J Peter Campbell. Introduction to machine learning, neural networks, and deep learning. *Translational Vision Science & Technology*, 9(2):14–14, 2020.

[14] Qiqi Wang, Hongjie Zhang, Cheng Qu, Yu Shen, Xiaohui Liu, and Jing Li. Rlschert: An hpc job scheduler using deep reinforcement learning and remaining time prediction. *Applied Sciences*, 11(20):9448, 2021.

Organizers & Author Index

Head of Chair:

- Prof. Dr. Martin Schulz

Organizers:

- Dr. Matthias Maiterth
- Dr. Eishi Arima
- Dr. Isaias Compres

Author Index: