Freie Universität Berlin

Masterarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Memory Consumption Based Decision-making for Plan Based Scheduling on HPC Nodes

Cora Glaß
Matrikelnummer: 5200206
cora.glass@fu-berlin.de

Erstprüfer : Barry Linnert
Zweitprüfer: Prof. Dr.-Ing. Jochen Schiller

Berlin, March 6, 2024

Abstract

Plan Based (PB) scheduling is a concept introduced for High Performance Computing (HCP) systems. The concept of the scheduler works for HCP systems in which jobs/programs are executed repeatedly. This enabled the PB scheduler to retrieve data during the last execution of the job. The PB schedulers, like its peers, schedule processes based on the available central processing unit (CPU) resources. Additionally, the PB scheduler accepts as deadline for each job and schedules the job without exceeding the deadline. If that is not possible, the job gets declined. The focus of this thesis is to introduce the memory consumption of processes and to enable the PB scheduler to take an additional resource, the memory, into account during the scheduling of the process. During this thesis, a prototype is developed that consists of two parts that would run in different components of an HPC system. The first part realizes the schedule planning based on the CPU and memory resources available. The second part runs on the kernel-level and retrieves the memory consumption of processes that are currently running. The development of the prototype results in new concepts and design choices that can be valuable to further research of PB scheduling and HPC itself. Furthermore, the work results of this thesis can increase the probability that scheduled jobs do not exceed the deadlines as it is ensured that the available CPU- and memory resources can cover the resource requirements.

Zusammenfassung

Plan Based (PB) Scheduling ist ein Konzept, welches für High Performance Comupting (HCP) Systeme entwickelt wurde. Das Konzept des Schedulers ist kompatible mit HPC-Systemen in denen die gleichen Jobs/Programme wiederholt ausgeführt werden. Dies ermöglicht es dem PB Scheduler Daten während der Ausführung eines Jobs zu sammeln. Der PB Scheduler, so wie generell alle Scheduler, trifft seine Entscheidungen basierend auf den verfügbaren Central Processing Unit (CPU) Ressourcen. Des Weiteren, wir dem PB Scheduler mit dem Job eine Deadline gereicht, die erfüllt werden muss. Der PB Scheduler plant das Scheduling daher, sodass der Job die Deadline nicht überschreitet. Falls dies nicht möglich ist, wird der Job abgelehnt. Der Fokus dieser Masterarbeit ist es den Speicherverbrauch von Prozessen zu ermitteln und das PB Scheduler Konzept zu erweitern, sodass auch die Speicherressource beachtet wird. Innerhalb dieser Masterarbeit wurde ein Prototyp entwickelt, welcher aus zwei Teilen besteht. Diese wurden designet, um in zwei unterschiedlichen Komponenten eines HPC-Systems zu laufen. Der erste Teil des Prototypens realisiert die auf die CPU- und Speicherressourcen basierende Schedule-Planung. Der zweite Teil des Prototypens läuft auf Kernel Level und sammelt die Speicherverbrauchsdaten der aktuell laufenden Prozesse. Die Entwicklung des Prototypens resultierte in die Bildung neuer Konzepte und Design-Entscheidungen, welche wertvoll für die Weiterentwicklung des PB Scheduler Konzeptes ist, sowie für HPC selbst. Des Weiteren, ermöglichen die Resultate der Masterarbeit eine höhere Wahrscheinlichkeit, dass Jobs die geplant sind tatsächlich die Deadline nicht überschreiten, da die nötigen CPU- und Speicherressourcen verfügbar sind.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

March 6, 2024

Cora Glaß

Contents

1 Introduction

In today's age, exponentially growing data and the progress in artificial intelligence (AI) technology makes high performance computing (HPC) an important tool in many areas of application. HPC systems, with their enormous amount of resources, are making it possible to process and analyze high amounts of data, as well as, execute calculations on them at a great speed. This kind of system can be used e.g. in research labs to run simulations, in companies to verify and improve product designs and in multiple areas (including the ones mentioned before) to train machine learning models or other kinds of AI tools. In the context of this thesis, all types of programs that run on HPC systems (excluding HCP scheduling/monitoring programs) are called jobs.

An example of a job running in an HPC system is a large-scale weather simulation. Such a simulation is a parallel program and also requires a high amount of resources. An HPC system consists of multiple supercomputers and/or computer clusters that enable the system to satisfy also high resource requirements of programs. Furthermore, an HPC system can run programs in parallel. Therefore, an HPC system is suitable to run jobs such as large-scale simulations.

The maintenance and usage of HPC systems require knowledgeable staff. Additionally, there are use cases that require real-time data processing. This is because the resulting data of a job can be directly used to e.g. improve some sort of model. The adjusted model will be used in the next iteration of the job. The delay of a job iteration delays the improvement of the model and therefore every job iteration after. Because of that, the practice (for such cases) is to host an HPC for mainly one use case alone. This means that the HPC resources are not fully utilized for the benefit of ensuring that a job can be executed on time. Thus, the potential of HPC systems in terms of a performance-cost balance is not fully reached yet.

The concept of Plan Based (PB) scheduling is an approach to improve utilization while still ensuring that the jobs will finish at a desired point in time. To explain the concept of PB Scheduling and the topic of this master thesis, it is important to understand the general idea of schedulers.

The basic concept of a scheduler at the node-level is to manage the time processes have on central processing units (CPUs)[1]. This means the scheduler decides which process runs on which CPU and the specific time frame. A scheduler can take into account aspects such as, how long the process already waited to be scheduled, whether the process holds a high priority or not and whether the process is currently blocked as it is waiting for resources to be freed up or communication messages to arrive. In the latter case, a smart scheduler will free the CPU and schedule another process until the current process is unblocked and can continue. Simply put, the decisions of a non-trivial scheduler[2] are based on whether CPU units are available at a current point in time and which processes are available to be scheduled.

[1] Schedulers can also manage job scheduling or resource assignments at system- or job-level.

[2] A trivial scheduler could schedule without taking any data into account and do it at random, which would not be a sufficient solution in our context.

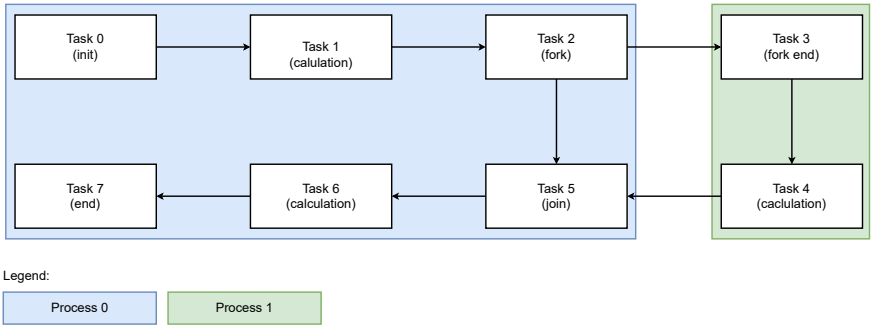

Figure 1: Task Net Example

A PB scheduler is a specific type of scheduler designed for HPC systems. Therefore, its use case differs from the ones of schedulers used in household devices, such as smartphones and laptops. PB Schedulers operate in systems in which users can request jobs to be run and finished before a desired deadline. The main advantage of the PB scheduler is that it enables the owner to use one HPC system to run all jobs while meeting the deadlines instead of maintaining multiple servers/HPC systems for each department.

The main precondition to using a PB scheduler is that the requested jobs are run repeatedly. Since the jobs are run more than once, the system can collect data during each execution of the job and use it to improve its scheduling, using the PB scheduler, for the next execution of the same job. A PB scheduler assumes that a job is already known to the system or that the user can initially provide the needed data to schedule the job efficiently. Otherwise, the first scheduling of the job cannot be scheduled by a PB scheduler. A job consists of processes. A process consists of several tasks that have to be run sequentially. The user has to initially provide the net of tasks. A simple example of a task net can be seen in Figure 1. The net shows all the tasks of a job. It also shows to which process each task belongs and how and with whom the processes/tasks are dependent on each other. This information is crucial for the PB scheduling. This is due to the fact that the PB scheduler receives a job request and produces a scheduling plan that plans out when and where each task will run. If the tasks are unknown to the PB scheduler, no plan will be created. In theory, it is possible to build a PB scheduler that schedules based on processes instead of tasks. However, it more complex to directly plan processes that depend on each other. For example, it is harder to estimate at which point in time a process will run or be blocked as it waits until a forked joins.

A PB scheduler creates a schedule plan for a job and collects data while the job is running. The data could be collected in different levels of depth. The following is an example of that. The data collection for three depth levels. The first level concerns the job in general. It collects the total and/or approximated runtime. The second level concerns the processes. It collects the approximated runtime of each process. The last level collects the number of instructions that are executed in a task. The number of instructions could also be provided by the user during the job request or a code analyzing tool can be used on the job code.

The next time the job is requested to run, the PB scheduler can use the data to estimate whether enough resources, here CPUs, are available to run the job while meeting the requested deadline.

A shortcoming of the current PB scheduler is the limited types of resources that are taken into account during the scheduling. Currently, the CPU and communication resource types are supported in PB schedulers. In the case of schedulers used in household devices, this can result in cache flushes or page swaps that will decrease the performance. For PB schedulers, it has an even more drastic effect. If the scheduler builds the plan, it does not take into account the memory consumption of a job, its processes and its tasks at all. Due to that, the plan might to be actually applicable. Imagen, the plan states to map two processes ($process_a$ and $process_b$) to the same CPU ($node_n$) onto different cores ($core_i$, $core_j$) to run at the same time. $node_n$ has enough cores to run both processes at the same time and all cores are free. However, the actual execution is not possible in the following situation:

$node_n$'s shared memory is: 1 GB
$process_a$ has a memory consumption of 0.7 GB and reserves it at the beginning of its execution until it finishes
$process_b$ has a memory consumption of 0.5 GB and tries to reserve it, but fails as only 0.3 GB is currently free

In this example, the plan is reasonable if we only take into account the available CPUs as all cores were free. However, due to the memory consumption of the processes, the plan is not applicable. The reason why the plan is not applicable is that the plan states that $process_b$ should run at the same time as $process_a$, but the amount of memory that is available when $process_b$ tries to reserve the memory does not cover the necessary 0.5 GB. In other systems, not working with fixed plans, $process_b$ would be blocked until enough memory is freed up and the schedule decides to assign it to a core or another process gets stopped to free up the memory for $process_b$. In our context, both solutions are not suitable as this means that the plan provided to the node is not executed as desired. The actual runtime of processes/tasks is delayed, the deadline might be exceeded and unexpected behavior can also cause other job plans to be inapplicable. Therefore, for HPC systems running a PB scheduler, $process_b$ would not be scheduled with a delay, but the whole job would be terminated. The job should have been, if possible, scheduled differently or not accepted in the first place.

The question addressed in this thesis is whether and how it is possible to extend the PB scheduler concept to support the memory resource type as well. The solution proposed in this master thesis is to extend a PB scheduler to also include the memory requirements/consumption of processes and their tasks. This thesis concentrates on building a prototype that realizes the first two steps of the following enumeration:

1. *Memory Usage*
   Obtain and process the memory consumption data.

2. *Task Mapping*
   Organize the mapping while ensuring that the tasks are mapped to CPU cores that can satisfy the CPU and memory requirements.

3. *Data Mapping*
   Focus on optimizing the mapping to make the most out of the cache hierarchy.

The prototype will not solve the occurrence of cache flushes as it will switch between processes on one core because the PB scheduler prioritizes meeting the deadlines of the jobs over only running one process at a time until it finishes to have as few cache flushes as possible.

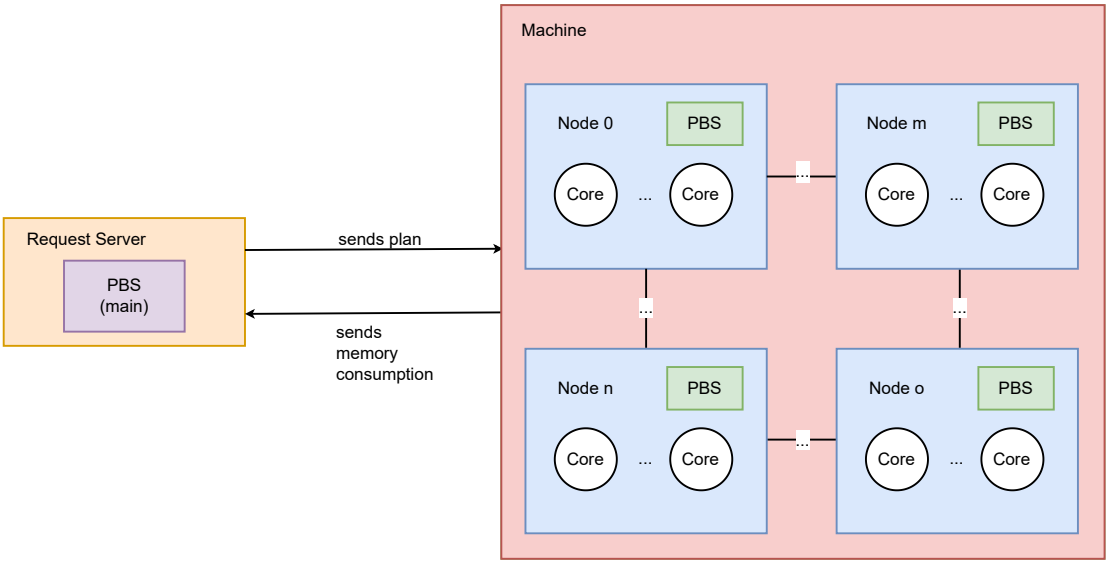

Figure 2: System Architecture

2 Background

This section gives an insight into the specific HPC system architecture that is assumed and on which the prototype design is based. Additionally, the criteria will be presented that are used to evaluate the prototype in Section 6.

2.1 The System

In the context of this master thesis, we will concentrate on one system architecture. A visualization of that architecture can be seen in Figure 2. The system consists of one machine that contains nodes. The nodes are arranged in a grid structure and each node has a multi-core CPU and shared memory that can differ in size. This means Node 1 can have more shared memory than Node m. Furthermore, each CPU can have a different amount of cores. All cores of one CPU will have the same tact frequency and are managed by one PB scheduler. In the following, the separation of nodes and CPUs will be neglected for reasons of simplicity. The shared memory of a node can be used by every core of that node. As seen in Figure 2 there is also a request server present. This architecture component can be a separate server/machine, which is the common architecture choice, or one node of the machine that will be seen as separated during scheduling, thus, no tasks of the requested jobs will be scheduled to be run there. In our case, the request server is separate. The request server is the interface to users. It receives the job requests from users.

The architecture contains two levels of PB schedulers. Figure 3 is showing an general overview of the architecture layers of a computer and in which layers the PB scheduler levels are located. It is important to note, that the overview is only used to visualize the layers and not a specific computer. As show in 2, the different PB scheduler levels are not running on the same architecture component.

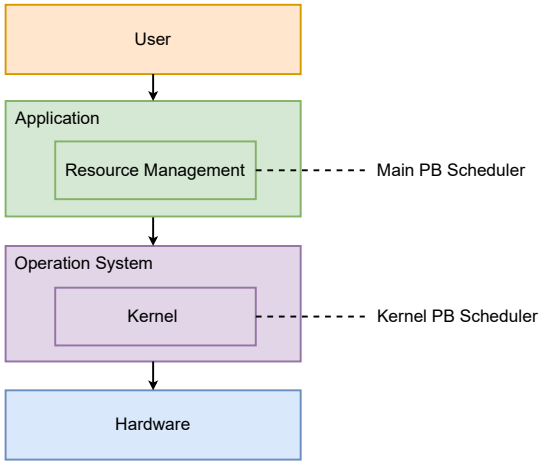

Figure 3: Computer Architecture Layer Overview

The first PB scheduler level is the main PB Scheduler that is run in the request server on the Resource Management level of the Application layer. It creates the scheduling plan which is based on the received job requests and the already planned executions. The second PB scheduler level is the kernel PB scheduler that is running on the kernel level of the Operation System layer. The main PB Scheduler and the kernel PB scheduler will be explained in more detail in Section 2.2.

2.2 The Prototype

As mentioned above, the prototype consists of two parts. The first part, the main PB scheduler, is running on the Resource Management level of the request server. The request server does store the data retrieved during the execution of the jobs. This data states the resource consumption of a job. The collection of data concerning the resource consumption of one resource type over all jobs that were observed is called a resource prediction model. The data can also be processed to determine and store average or approximated values in the prediction model. The initial main PB scheduler would expect a CPU prediction model. Such a model contains the number of instructions per task. Based on this data and assuming a CPU of average performance, the approximated total runtime of the job and the approximated maximal runtime of each process can be estimated. This results in an additional model, called the CPU runtime prediction model. The prototype of this master thesis is additionally provided with a memory consumption prediction model (MemC model). In the context of this thesis, the initial format of the task net is a net of tasks that indicates in which order the task should be executed. No further data concerning the tasks is present. The data of a resource prediction model can be processed and combined with the task net. As a result, the net stores the usage data of a specific resource type for each task. The task net concept is further discussed in Section 4.2.

The main PB scheduler is an off-line scheduler. This means it schedules prior and not at runtime[3]. The main PB scheduler takes the job request and executes pre-checks

[3] A scheduler that schedules at runtime is called on-line.

before starting the actual plan building. These checks should filter out jobs that cannot be executed and/or cannot be completed before the desired deadline due to the lack of available resources. As the checks work with rough estimation they should not be too strict to avoid declining reasonable job requests. There are also additional aspects that should be taken into account to build an efficient main PB scheduler. In the following the criteria are presented that are used to build and evaluate the main PB scheduler. They are divided into functional and nonfunctional criteria, sorted to the best of my knowledge.

Functional Criteria

1. Executable: It can be executed and can produce a plan.
2. Flexible: It takes into account the MemC model.
3. Tolerable: It tries to map more than once if the initial mapping of processes and tasks does not meet the deadline.

Non Functional Criteria

1. Filtering: It executes pre-checks and filters out invalid jobs while not being too strict.
2. Modular: The coding is modular, the interfaces are clear and the implementation can be extended.
3. Stability: There is exception handling in place and it does not break the execution.

The latter of the nonfunctional criteria, stability, could also be argued to belong to the functional.

The second kind of the PB scheduler is the kernel PB scheduler. It runs on each node in the kernel-level. It schedules the tasks to cores based on the provided plan and collects data about the processes/tasks during the execution. The prototype of this thesis does collect the memory consumption data. However, it does not do the actual mapping of processes/tasks to cores as this is out of the scope of this thesis. Therefore, strictly speaking, the second part of the prototype is an adjustment of the kernel scheduling, rather than, the implementation of a kernel PB scheduler. To simplify the visualization in Figure 2, the PB scheduler visualized in each node should represent the adjusted kernel scheduling.

Furthermore, the kernel PB scheduler forwards the collected data to the user-level of its node which sends the data to the request server, or to a database that the request server can request data from. This is also not the focus of this thesis and therefore not realized by the prototype.

The evaluation of the second part of the prototype is done using a subset of the criteria of the main PB scheduler.
The evaluation criteria for the kernel PB scheduler prototype are the following:

1. Executable: The code of the prototype is executable, gets called by the kernel schedulers and creates/updates the MemC model.
2. Modular: The coding is modular, the interfaces are clear and the implementation can be extended.
3. Stability: The coding is stable and does not run into access errors (such as `NullPointerExceptions`).

3 Related Work

The PB scheduler concept and research is part of Virtual Resource Management (VRM) project, supervisor of the topic: Barry Linnert. Since the start of this research topic, among others, multiple students have contributed to it.

As mentioned before in Section 2.2, the prototype of this master thesis is designed with a pre-defined, limited scope in mind. As a consequence, it does not e.g. execute the actual mapping of tasks to the cores. The author of [4] contributed the implementation of a working PB scheduler that is running in the kernel and does the scheduling based on the provided plan. The scheduler focuses on realizing the scheduling process and does not obtain resource data to build a prediction model.

The thesis [5] introduces a kernel module that enables the system to monitor the communication between parallel executed processes. Based on the data it produces the message length during communications that can be used as prediction model data and reused in the next execution of the job. The module is similar to what I want to realize with the first part of my prototype in terms of obtaining data for a resource type and reusing it in the next iteration. However, the work in [5] concerns communication resource type instead of memory resource type.

As seen in Figure 2, the request server and the nodes of the machine are sending data to each other. Furthermore, the plans can only be received from user-level processes. Thus, the plan has to be forwarded from the user-level to the PB scheduler at the kernel-level. The prototype of this master thesis does not realize the actual communication nor the forwarding. The author [2] focused on the topic of forwarding the received plan from the user-level process to the PB scheduler in the kernel.

The following work is not based on the context of HPC systems using a PB scheduler. It is also the only work presented in this section that is not a student thesis. It is also not a contribution to the Software Engineering work group. The author of [3] introduces the concept of Task and Data Mapping which inspired the three steps mentioned in Section 1. The author describes how he realized the Task and Data Mapping on a shared memory architecture.

job details	start deadline total memory consumption total runtime
process details	approximated memory consumption approximated runtime
task data	task net

Table 1: Job Request Data Structure

4 Design

At the current time, there is no HPC system available that is running the main or the kernel PB scheduler. Therefore, it was not possible to design and implement the prototype in a connected environment. To be able to build both parts of the prototype it was necessary to define requirements and assumptions beforehand. They enabled me to form concepts and interfaces for both parts of the prototype. This was done to ensure that the design of both parts would be compatible if they would be run at some point in the same system. Additionally, these interfaces are used to know what has to be provided or neglected artificially for the prototype parts to do their work in the development environment. Furthermore, the interfaces as well as their artificially provided data show what data should be sent by the user to the request server or between the request server and the machine.

4.1 Main PB Scheduler

The main PB scheduler expects specific data to be present for each job that is requested to be executed in the system. Table 1 shows how the request data for a job can be structured. Currently, that data is stored in the prototype itself. In a real application of the main PB scheduler, the user has to send more details with the first request to run a job. This means, the job is requested to run for the first time in that system and is yet unknown to it. The user would have to send the initial task net with approximated maximal memory consumption and the actual number of instructions as weights for each task of the task net. The start could be set by the main PB scheduler as, e.g. the point in time when the server received the request for the concerning job. The approximated and total data set in the job and process details could be calculated based on the provided task net. After the first run and the arrival of the memory consumption data, all memory consumption data will be updated[4].

The PB scheduler also requires knowledge about the current utilization of each node and core of the machine to build plans. The main PB scheduler prototype builds all plans for all jobs running on the machine. Therefore, the prototype can store the timeslot, core details and approximated memory consumption for each task during

[4]The same goes for the runtime data, however, the kernel PB scheduler prototype of this thesis does not collect the runtime data

the schedule planning. If the current schedule planning fails, the data can be deleted. Otherwise, it can be used as utilization data to know at what times the cores are free and how much memory is reserved in each node for how long.

4.2 Task Net

The task net can be visualized as a directed acyclic graph (DAG), as seen in Figure 1. This net enables the scheduler to know which processes are executed during the job and the specific tasks. The net states for each task the process to which it belongs, the task action and the next task in the process. For tasks of the action type communication, it is stated to which task it will communicate. It is possible to combine the task net with the content of the prediction models. It is important to note, that in the context of this thesis, the initial format of the task net does not contain prediction data. This data is added when a prediction model is combined with the task net. As mentioned in 4.1, the task net of the prototype combined the task net with the memory consumption model and the CPU runtime data. This results in a task net that contains prediction data of the memory consumption and CPU runtime for each task.

4.3 Utilization Model

The utilization data is used as a model of the utilization of the machine. The collected data of the main PB scheduler during the schedule planning will be processed for the model. The model stores for each planned task an entry that contains the time frame when it will run, the necessary amount of memory and data about the specific core on which it will be scheduled. In addition to the task entries it is necessary to also have entries about the process's runtime and memory consumption. At the start of each process, the process allocates the memory it needs during its execution. This memory will only be deallocated after the process is finished[5]. The main PB scheduler has to be able to take into account that the process might have already allocated memory on that node. The required data to determine what could be extracted from the task entries. However, the extraction would be executed repeatedly and introduce more complexity to the prototype code. Therefore, saving the process entries is more sufficient. This design was chosen after creating multiple drafts of possible utilization model structures. They were carefully comparing while focusing on whether the structure enables the main PB scheduler to determine the total reserved memory for a core during a specific time frame.

With no further assumption, the utilization model would not take into account some crucial aspects. This is due to the fact, that the model only concerns the processes and tasks that are requested to be run. Thus, it would neglect the possibility of maintenance and monitoring being scheduled in-between tasks. It also would neglect the fact that the kernel PB scheduler itself needs to be scheduled onto a core from time to time to do the actual scheduling.

[5]The memory allocated for a process does get deallocated in case a process gets terminated. However, this case will not be further focused on in this thesis.

The master thesis [4] proposed a logic to tackle the issue of the difference between the model and reality. The logic works based on the assumption that the prototype is not aware of the full CPU time capacity. In short, the prototype only sees the CPU time intervals that are available to it. The rest is reserved for the necessary processes run on each CPU.

4.4 Memory Consumption Model

The memory consumption model holds an entry for each process that contains the maximal memory consumption that was observed during the execution of that process. There is a difference between the MemC model created by the current prototype and the structure that is assumed by the main PB scheduler. The created MemC model holds entries for every process that was running during the time the memory consumption extension was executed. Each process entry holds the maximal memory consumption that was observed during the run of the process. The MemC model structure that is expected by the PB scheduler differs as it assumes that the MemC model is created and collects data in an HPC system that is running the kernel PB scheduler. Therefore, it expects entries for each job (job entries), as well as, entries for each process (process entry) of a job that indicates to which job that process belongs. A process entry stores the job ID, process ID and the maximal memory consumption that was observed during the run of the process. A job entry stores the job ID and the total memory consumption. Thus, the sum over all maximal memory consumption of the processes of the job.

4.5 Kernel PB Scheduler

Due to the fact, that this prototype was not built and tested in a system that is actually running the main and kernel PB scheduler, some simplifications were made. My kernel PB scheduler prototype is not based on the PB scheduler by [4] and I do not extend it to obtain memory consumption. Instead, I based my kernel PB scheduler prototype on a Linux kernel code base[6].

The extension is located in the general kernel schedule module and is called as part of the scheduling framework of the kernel. Therefore, strictly speaking, this prototype part is not a scheduler but an extension of the Linux kernel schedule module. The Linux kernel does not work with processes and tasks from the PB context but with Linux processes and Linux threads. The following assumption was made: The prototype neglects the differences between the concept of tasks and threads and obtains the memory consumption of the threads instead of tasks. Also, the concept of jobs is neglected as the kernel does not know which process would belong to the same job. The difference between the processes and tasks used in PB schedulers and the Linux processes and threads are discussed in more detail in Section 5.2.

The following are the requirements for the memory consumption retrieval:

[6] https://github.com/torvalds/linux

1. The extension has to be run in the kernel (kernel-level processes).
   The reason for that is that the user-level processes do not have permission to obtain the memory consumption data.

2. The extension has to be called by the currently used scheduler.
   Otherwise, the collection of the memory consumption data will not be executed. If the system would run an actual PB scheduler the introduced kernel PB scheduler prototype logic should be located in that scheduler and would only be executed if the PB scheduler is selected. However, as this is not the case the prototype coding should be executed independently of which scheduler is currently selected.

3. It should be possible to distinguish between processes run at the kernel-level and the user-level.
   Otherwise, we collect and add memory consumption data for processes that are not part of the requested job. This means we would end up storing data of kernel-level processes, such as the processes triggered during scheduling itself. Data concerning such processes is not important to the main PB scheduler and is therefore not necessary to be stored.

4. It must be possible to distinguish between different processes.
   Otherwise, the memory consumption model cannot be created as the model holds entries for each process.

5 Implementation

This section connects the concepts mentioned in Section 4 to the implemented structures. Furthermore, the technical details of the schedule planning are discussed.

5.1 Main PB Scheduler

The main PB scheduler is implemented in Python3. The coding is located in the private `master-thesis-main-pb-scheduler` repository[7].

The scheduler initializes the utilization model before accepting any job requests. The job request handling consists of the following steps:

1. Job Request Data Processing
2. Job Validation
3. Reservation (=Schedule Planning)
4. Plan Building

The following subsection will discuss the utilization model, the implementation of all steps of the job request handling and the exception handling.

5.1.1 Utilization Model

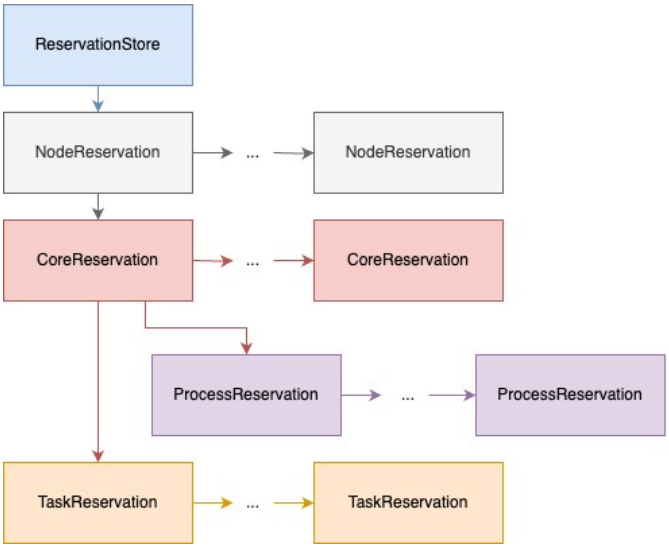

Figure 4: Reservation Store Class structure

The `ReservationStore` class implements the utilization model. Figure 4 shows a simple visualization of the Python class structure of the utilization model.

The main PB scheduler requires a file containing the machine details to initialize the Reservation Store. This file contains an entry for each node of the machine and is

[7] https://github.com/099111114097/master-thesis-main-pb-scheduler

located in the `data` folder in the prototype code source. Each entry contains the node ID, the names of each core of the node, the total memory shared across the cores and the tact frequency (`instructions_per_sec`) which is the same for all cores of the node. After the initialization stage of the main PB scheduler the Reservation Stores will contain the following.

There is one instance of the `ReservationStore` class. This instance points to the first element of a linked list containing `NodeReservations`. There is a `NodeReservation` instance available for each node of the machine. Every `NodeReservation` instance points to the first element of a linked list of `CoreReservation` instances representing the cores of the concerning node. In the following, the name of a Phyton class will be used as a synonym of its instances. This means a `ProcessReservation` refers actually to an instance of the `ProcessReservation` class.

No `ProcessReservation` or `TaskReservation` exists at this point. They get added during the schedule planning. The validation step temporarily adds `ProcessReservation` and marks them as inactive to differentiate them from `ProcessReservations` that are actually valid.

Table 2 states the most important attributes of each class of the utilization model. The purpose of the attributes will be explained during the job request handling steps. A Table of all attributes of each class can be found in the Appendix A.

It is important to mention that this specific implementation of the Utilization Model enforces a specific memory management. In the context of this master thesis, this memory management type will be called static memory balancing. When a `CoreReservation` is created, the size of memory that is available to it will be set. Each core of a node gets the same size of memory assigned to it. This means the memory is divided fairly between all cores of a node. Another memory management type is the dynamic balancing of the memory between cores. This would mean, that no core has a fixed size of memory available to it.

Imagen, there is a process ($process_p$) with a memory consumption (mem_cons_p) below the total memory of the node ($total\_mem_{node}$). $process_p$ can be run on a core of the node. When $process_p$ starts to run and is not yet finished, there is only $total\_mem_{node} - mem\_cons_p$ of memory remaining that is available on the core for other processes to reserve from when they start. Simply put, static memory balancing causes internal memory fragmentation and dynamic memory balancing results in external memory fragmentation. A visual example of the internal and external memory fragmentation can be seen in Figure 5 and 6.

Class	Attribute	Data Type
ReservationStore	node_head	NodeReservation
NodeReservation	shared_memory	Integer
	instructions_pre_sec	Integer
	core_head	CoreReservation
	next	NodeReservation
CoreReservation	memory	Integer
	instructions_pre_sec	Integer
	task_res_head	TaskReservation
	process_res_head	ProcessReservation
	next	CoreReservation
ProcessReservation	start	Integer
	end	Integer
	job_id	Integer
	process_id	Integer
	memory_to_res	Integer
	test_idel_used	Boolean
	active	Boolean
	next	ProcessReservation
TaskReservation	start	Integer
	end	Integer
	job_id	Integer
	process_id	Integer
	memory_reserved	Integer
	next	TaskReservation

Table 2: Important Utilization Model Class attributes

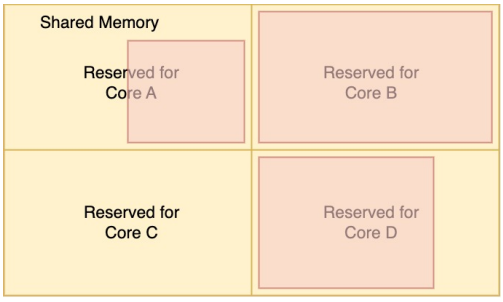

Figure 5: Static Memory Balancing Internal Fragmentation Example

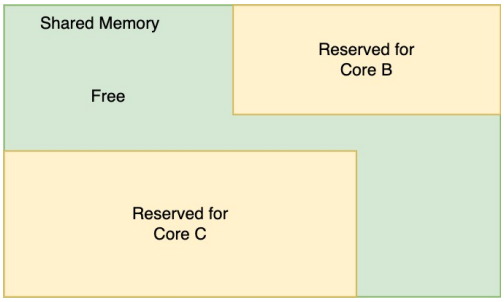

Figure 6: Dynamic Memory Balancing External Fragmentation Example

Dynamic memory balancing makes it possible to schedule processes with an even higher memory consumption as the memory limit of a dynamically balanced core is higher (as the memory is not divided by the number of cores). However, this is only true if the memory is not already reserved by another core and it increases the complexity of the reservation and the validation checks. This is due to the fact, that it is not sufficient enough to check the existing `TaskReservations` and/or `ProcessReservations` of one core to check whether there is enough CPU time and memory available during a specific time frame. Instead, all `TaskReservations` and/or `ProcessReservations` of the `ReservationStore` would have to be checked.

In the following, a `ProcessReservation` linked list refers to the list accessed from the `process_res_head` field of a `CoreReservation`. Similar to that, a `TaskReservation` linked list refers to the list accessed from the `task_res_head` field of a `CoreReservation`.

The classes of the utilization model are located in `reservation_static.py`.

5.1.2 Job Request Data Processing

During the processing of the job request data, the job details, process details and task net are read from the `data` folder located in the prototype code source[8]. The data is used to initialize instances of the `Job`, `Process` and `Task` Python classes. Similar to the utilization model, the `Job` class has only one instance which is pointing to a linked list.

These classes are introduced to build a more detailed task net (in the following PB task net [9]) that is easily readable by the scheduler during the schedule planning. Table 3 shows the important attributes of all classes and how they are connected. It can be seen that instances of the classes `Task` and `Process` are referenced in multiple ways. This is done to enable fast access without repeatedly iterating over linked lists.

A task can have one of the following action types: start, end, calculation, communication, fork, fork end, and join. The first two types can both only appear once in one PB task net as they indicate the start and end of the job. The communication type

[8] https://github.com/099111114097/master-thesis-main-pb-scheduler/tree/main/data

[9] The name is introduced to differentiate the detailed task net build out of Python class instances from the initial task net

Class	Attribute	Data Type
Job	start	Integer
	deadline	Integer
	total_runtime	Integer
	total_needed_memory	Integer
	head	Task
	processes	List of Processes
Process	process_id	Integer
	approx_runtime	Integer
	approx_needed_memory	Integer
	forked	List of Processes
	task_head	Task
Task	process_id	Integer
	task_id	Integer
	action	Integer
	needed_memory	Integer
	instructions	Integer
	child_process	Process
	comm_to	Process
	next	Task

Table 3: Important Task Net Class attributes

indicates that a message gets sent to another task. This thesis focuses on memory consumption and not on the complexity of scheduling communication tasks depending on each other. Therefore, our prototype schedules communication tasks with the assumption that the communication is asynchronous.

After the PB task net is fully initialized the PB scheduler can continue with the job validation step.

The code for the job request processing is located in `request_processig.py`. The classes for the PB task net are located in `job.py`.

5.1.3 Job Validation

The job validation step is used to filter jobs that are not possible to be scheduled on the machine. This is done to avoid continuing with the schedule planning when it is early on clear that the job cannot be successfully scheduled and, thus, no plan can be created for it. The validation step validates in three levels whether the job can be scheduled. If the validation fails at any level it will stop the validation and terminate the job request handling and inform the user (via terminal) that the job request is not possible.

Level 1
The first level is a general check. It is validated that the deadline is possible. A deadline that is too early will be rejected. That would be the case if the deadline

is earlier than the start and the total runtime summed up. The second part of the general check retrieves the total memory over all nodes and checks if the job requires more memory than the whole machine has available.

Level 2
The second level is the job check. This check verifies that enough memory is available to meet the estimated memory consumption of the job. The check is successful when 80% of the estimated memory consumption is available. The default value of 80% is a first rough estimation and not the result of statistical calculations over CPU runtime data of processes. With a value of 80%, it is assumed that the waiting time of a process is significantly smaller than the CPU runtime. Thus, when a process starts it is running the majority of its lifetime, which is a realistic assumption to make for jobs running HPC systems. The percentage can be adjusted by changing the constant `MIN_MEM_PERCENTAGE`. The estimation of the available memory consumption is done by roughly calculating the memory needed during the estimated runtime (`=want`) of the job and the memory not reserved across all nodes during the time between the start and the deadline of the job (`=left`). In the following, the time duration from the start until the deadline will be called maximal runtime. It is the upper bound of the job runtime. The check starts with calculating the memory needed during the time of the job. There are two versions provided by the prototype on how to calculate that. The Listings 1 and 2 show the calculation versions.

```
want = job.total_runtime*j.total_needed_memory
```

Listing 1: Memory Consumption Over Time Calculation Version 0

```
want = -1
for p in processes:
    want += p.approx_runtime*p.approx_needed_memory
```

Listing 2: Memory Consumption Over Time Calculation Version 1

Version 1 is assumed to be closer to the exact values of the memory consumption and runtime. However, that is only true if the approximated data is correct. Additionally, it might be beneficial to use a more rough estimation of the needed memory. Therefore, version 0 of the calculation is set as default during the calculations. It can be changed to version 1 by adjusting the constant `CAL_WANT_TYPE`. Thus the calculation version is not adjusted by users, but only the owner of the main PB scheduler.

The calculation of `left` is more complex. It starts with getting the total memory across all nodes and multiplying this with the maximal runtime. This results in the total memory available over the maximal runtime. The idea is to search for all `ProcessReservations` for which their time frame (the interval of their existence) overlaps with the time between the start and deadline of the job. For each `ProcessReservations`, that overlaps, that time interval of the overlap gets multiplied by the memory that is reserved by that process. This results in the memory reserved over the time of the overlap, also called *taken_memory_over_time*. The memory reserved over time of every `ProcessReservations`, that overlaps, will be added up. The sum is the total

```
def process_test_mapping(rs: re.ReservationStore, init_process: job.Process,
j: job.Job):
    res = rs.add_test_p_reservation(j.start, j.deadline, init_process.approx_runtime,
    init_process.approx_needed_memory, j.job_id, init_process.process_id)
    def test_mapping_forked_p(parent, res):
        if len(parent.forked) == 0:
            return
        for i in range(len(parent.forked)):
            p = parent.forked[i]
            res = rs.add_test_p_reservation(res.start+((i*FORK_TIME)+INIT_TIME),
            j.deadline, p.approx_runtime, p.approx_needed_memory, j.job_id,
            p.process_id)
            test_mapping_forked_p(p, res)
    test_mapping_forked_p(init_process, res)
```

Listing 3: Process Test Mapping Function

memory reserved over time. This will be subtracted from the total memory available over the maximal runtime. With that the calculations estimated `left`.

The last step is to divide `want` by `left` to see whether the memory requirements are fully or partly met and to which extent. Now the result can be checked against the accepted percentage (for that the result has to be multiplied by 100)

Level 3
The last level is the process check. Out of the three levels, this is the most complex one. This check verifies whether it is possible to find enough available CPU time and memory to map all processes of the job to specific cores that are free and have enough memory available at the specific time frames. The verification process is to create `ProcessReservations` for each of the `Process` instances using the approximated memory consumption and runtime. The goal is to find and temporarily map the `ProcessReservations` to the `task_res_head` linked list of a core that satisfies the necessary runtime as well as the memory requirements. Additionally, it has to be taken into account that the `ProcessReservations` of processes that were forked cannot be mapped to a time frame that starts earlier than the time frame of its parent `ProcessReservations`.

Listing 3 shows the test mapping function implementation.

The function will be called with the `Process` instance of the initial process. The mapping order will start with the initial process and continue with the processes that it forked. These can be found in the `forked` list of the `Process` instance. At this point, the mapping will be triggered recursively by mapping each process of the forked list as well as calling the `test_mapping_forked_p` again for the `forked` list of the currently mapped process. This ensures that the check will map all processes of the job while being able to map them time-dependently to each other. As seen in the code snippet above, the forked processes are reserved relative to the start time of the reservation of the parent process. During the explanation of this check, node, core

and process will be used as synonyms for `NodeReservations`, `CoreReservations` and `ProcessReservations`. For example, iterating over cores means that it will be iterated over the linked `CoreReservations` list.

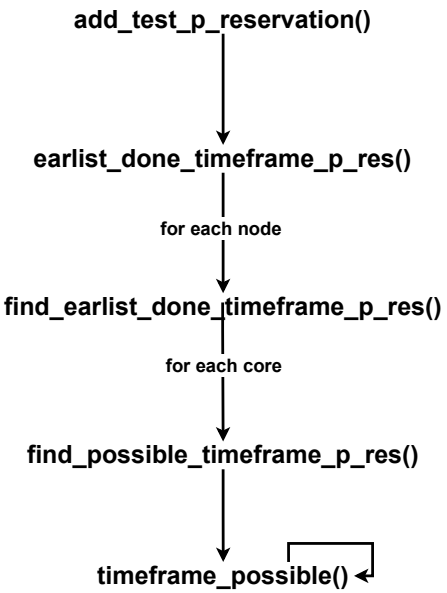

Figure 7: Test Mapping Function Call Overview

The function `add_test_p_reservation` starts the test mapping in the `ReservationStore`. Figure 7 shows the general flow of function calls for the test mapping from this point on. `add_test_p_reservation` first obtains all nodes that satisfy the memory requirements of that process. This means processes for which the cores do not have enough memory to execute the process at all are filtered out. If no nodes are found that satisfy the requirements, the check fails and the job will be declined.
In case, nodes with sufficient memory are found, the `earliest_done_timeframe_p_res` will be executed. This function itself goes through all the nodes with sufficient memory and executes the function `find_earliest_done_timeframe_p_res` to retrieve for each node the earliest time frame that satisfies the runtime and memory requirements. It returns the time frame with the earliest start. The called function, `find_earliest_done_timeframe_p_res`, iterates over all cores of the node and calls `find_possible_timeframe_p_res` to retrieve for each core the earliest possible time frame. This function, again returns the time frame with the earliest start.

To determine the earliest possible time frame available for a core the `timeframe_possible` gets called. Listing 4 shows the implementation of the function.

When `timeframe_possible` is called for the initial time for a core, it will start out with `process_res_head` of the core in question. The start is the earliest point in time in which this process is supposed to run. The set start ensures that the available time frame is not in a point in time where the process is not forked yet or the job is not supposed to start. The approximated runtime and approximated memory consumption are taken from the `Process` instance.

The recursion anchor of the function is the first two 'if conditions' (line 3-6). If the provided `head_start` is empty, there are no `ProcessReservations` at this point and the core is free from the provided start point. If the start of the `ProcessReservations` head starts later than $start + approximated\_runtime$, an available time frame before the `head_start` was found.

```
def timeframe_possible(self, head_start, start, approx_runtime,
approx_needed_memory)-> (int, int):
        if head_start == None:
        return start, start+approx_runtime
    if start+approx_runtime <= head_start.start:
        return start, start+approx_runtime
    else:
        if self.memory - head_start.memory_to_res >= approx_needed_memory:
            overlap = head_start.overlap(start, approx_runtime)
            approx_waiting = head_start.test_idel_time()
            if overlap <= approx_waiting:
                head_start.test_idel_used_set()
                return start, start+approx_runtime
            rest_overlap = overlap - approx_waiting
            if rest_overlap == approx_runtime:
                start = head_start.end
            if head_start.next == None:
                head_start.test_idel_used_set()
                return start, head_start.end+rest_overlap
            else:
                timeframe = self.timeframe_possible(head_start.next, start,
                rest_overlap, approx_needed_memory)
                if head_start.process_res_in_interval(timeframe[0], timeframe[1]):
                    head_start.test_idel_used_set()
                return timeframe
        else:
            return self.timeframe_possible(head_start.next, head_start.end,
            approx_runtime, approx_needed_memory)
```

Listing 4: timeframe_possible Function Implementation

Start	End	Reserved Memory (in GB)
0	4	0.3
6	8	0.4
14	16	0.7

Table 4: ProcessReservations Linked List Content Example

If both 'if conditions' are false, the function continues. If the provided memory reservation of the head `ProcessReservations` leaves not enough memory for the process that should be test mapped, the `timeframe_possible` functions get called with the next element of the `ProcessReservations` linked list as head and the start time will be the end field value of the currently checked `ProcessReservations` (line 26-28).

It is possible to schedule a `ProcessReservations` that overlaps partly or fully with another `ProcessReservations`. A full overlap means, in the context of this thesis, that one `ProcessReservations` starts before the other and ends after it. The partly and full overlap is possible due to two separate reasons.

Full Overlap

To understand why it is possible to have `ProcessReservations` overlap fully with one or multiple `ProcessReservations`, imagine the following.
A core $core_c$ has a `ProcessReservations` linked list. The core is assigned 1 GB of the shared memory of its node, in the following $core\_mem_c$. For process $process_p$, the function `timeframe_possible` tries to find the earliest possible time frame. The process should start earliest at the start time: 2. Furthermore it has an approximated runtime of 6 and has an approximated memory consumption of 0.5 GB. Table 4 shows the content of the `ProcessReservations` linked list of the $core_c$.
The earliest possible start, while taking into account the desired start, will be 4. As another `ProcessReservations` starts at 6, $process_p$ could run from 4 until 6. This would not fully satisfy the approximated runtime. Instead of searching for another starting point, it is still possible to start at 4 in case the following `ProcessReservation` does reserve an amount of memory that does leave enough of $core\_mem_c$ so that the memory needed about $process_p$ can be reserved before, and can be kept allocated.
The `ProcessReservation` with the time frame from 6 to 8 reserves 0.4 GB which leaves 0.6 GB free. Therefore, $process_p$'s memory requirements can still be satisfied and the process can continue running at 8 and finish at 12. With that, the process would run 6 time units and be able to reserve 0.5 GB from 4 until 12 without blocking another process or being terminated.
Please note, that this check is used to filter out jobs that cannot be executed with the available resources. The filter works with rough estimations. Furthermore, it does not take into account that there may already be `ProcessReservations` overlapping each other. Extending the check to actually take into account that

there might be multiple memory reservations at the same time is a topic of Section 7.

Partly Overlap

The partly overlap is possible as the following is assumed. A process does not run on a core without any interruption. A process can be blocked as it waits for access to a resource that might be shared among multiple processes and gets locked. A process that forked another process has to also wait until the forked process is finished in case it wants to join it with itself. This logic is the case for the jobs that we assume are requested and handled by the prototype. Another reason can be that the process waits for the arrival of a message. As mentioned before, in the context of this thesis, we only work with asynchronous communication. Therefore, this will be not the case for our prototype. However, it is mentioned for the sake of completeness.
To represent and take into account the fact that the time frame of `ProcessReservation` can include time units where the core is actually free, the check works with a general estimation. It is realized in the code with the constant `PROCESS_IDLE_PERCENTAGE` and the `test_idel_used` field of the `ProcessReservation` class. The constant indicates the portion of the time frame where the core is free, in the following also called idle time. The idle time of a process can be used by another process, as long as the memory reservations of the processes do not collide. This means if the `ProcessReservations` of the processes overlap in any way, the combined memory reservation cannot exceed the available memory of the core.

The function `test_idel_time` calculated the estimated idle time of a `ProcessReservation` based on the `PROCESS_IDLE_PERCENTAGE` and the value of the `test_idel_used` field. If `PROCESS_IDLE_PERCENTAGE` is set to 0 and/or `test_idel_used` is set to true the idle time is zero. The assumption here is that either, no idle time is assumed (`PROCESS_IDLE_PERCENTAGE`=0) or another `ProcessReservation` of the test mapping already overlaps with the `ProcessReservation` in questions (`test_idel_used`=true). This means the idle time of the process is already used by another process. As mentioned before, however, it does not take all situations of overlaps into account. If `ProcessReservations` overlap with each other and are not created during the test mapping, the check is not knowledgeable about the overlap.

If the value of `PROCESS_IDLE_PERCENTAGE` is a float f for which $0 > f > 1$ applies and `test_idel_used` is set to false, the idle time can be calculated. The following shows the calculation:

$$idle\_time = (end - start) * PROCESS\_IDLE\_PERCENTAGE$$

The start and end values are taken from the `ProcessReservation` instance. If the idle time satisfies the remaining time of the process, meaning the overlap time, the time frame (start, start+`approx_runtime`) can be used as the time frame of the new `ProcessReservation`. If the overlap is greater than the idle time, the next `timeframe_possible` function will be called with the next element of the `ProcessReservation`

linked list. The start variable gets updated before the function is called if the remaining runtime is equal to the approximated runtime. This means no time units were found before and during the currently checked `ProcessReservation`. Therefore, the start would be updated with the value of the end field of that `ProcessReservation` (line 15-16). After the recursively called function returns a time frame, it will be checked whether it overlaps with the already checked `ProcessReservation`. This happens when the idle time of that process is used. If the time frames overlap the `test_idel_used` field gets updated (line 23-24). In the end, the found time frame will be returned.

The functions call in-between the `timeframe_possible` and `add_test_p_reservation`, see Figure 7, iterate over the returned time frames and return the time frame with the earliest end value. Thus, the scheduler takes the time frame for which the task would be done at the earliest possible point in time. Finally, the time frame can be used to create the new `ProcessReservation`. After it is checked that the end value of the time frame does not exceed the desired deadline, the `ProcessReservation` gets added to the `ProcessReservation` linked list of the specific core.

If the deadline is exceeded an exception would be raised to indicate that the test mapping failed and the job should be declined. If all processes of a job are successfully test mapped and do not exceed the deadline, the check is successful and the `clean_test_mapping` function removes all `ProcessReservations` that have been added during the test mapping. Without the clean-up of the inactive `ProcessReservations`, the prototype would take them into account in the actual schedule planning as well as the test mapping for the next job.

The job validation step is finished and the prototype continues with the actual reservations/schedule planning.

5.1.4 Reservation

The reservation step creates reservation resources for the job and adds them to the `ReservationStore`. Reservation resources refer to `ProcessReservations` for processes and `TaskReservations` for tasks. In the following, the expression of doing the reservation for a process/task means that the prototype finds a time frame in the reservation resource linked list of a specific core and uses this to create the reservation resource and adds it to that linked list. Doing the reservations for processes/tasks of a job also realizes the schedule planning for that job. This is due to the fact, that the resulting `ProcessReservations` and `TaskReservations` state when and on which core the processes/task should run.

The reservation logic has to satisfy a number of requirements to ensure that the reservations do represent a possible schedule plan in the end. The requirements are the following:

1. None of the reservation resources can exceed the deadline.
   This means that the value of the end field of any reservation resource of the job cannot be higher than the deadline.

2. The first task of a child process can only start after the task that forked that child finished.
3. A parent process can only start its join task after the last task of the child process that will join is finished.
4. `TaskReservation` instances cannot overlap.
   A core can only execute one task at a time. It executes a job successfully a task cannot be planned to be interrupted after it is started.[10]
5. The `ProcessReservations` of parents and higher have to be taken into account when finding a time frame for the children processes/tasks.
   It is important as the prototype does add the `ProcessReservations` instance of a process to the linked list of a core after the end field is set. However, the end field can only be set when all child processes have joined back and the last task of the process has a `TaskReservation` instance that has been added to a core.
6. All `TaskReservation` instances of a process have to be added to the linked list of the same core.
   This is necessary due to the implementation of the `ProcessReservation` as it only can apply to one core. Furthermore, all tasks of a process running on the same core has also an advantage when looking at the cache management.

The reservation logic, implemented in the prototype, consists of going through all processes of a job and doing the reservation for all tasks of each process while satisfying the requirements. The process reservation resource is done based on the first and last task reservation resources. They indicate the start and end of the process reservation. In the following, the reservation logic will be discussed in more detail.

The reservation step starts with the call of the recursive function `add_reservation_for_process`. Listing 5 shows the implementation of the function.

To understand the reservation step implementation it is useful to view the recursive `add_reservation_for_process` function in more detail.

For the first part of the explanation, we assume that the reservation process is executed successfully and no exception will be raised. Afterwards, the logic of the reservation retry will be discussed which enables the prototype to try different ways to plan the scheduling, in case the first try fails.

The function `add_reservation_for_process` gets called for the first time with the necessary job details (job ID, start and deadline), the first `Process` instance of the job, an empty list[11] and `schedule_try`=0.

Line 4-12 prepare variables that will be needed during the process of the recursive process reservation. The function name indicates that it does the reservation of one process. However, the function gets called recursively after the reservation for a fork task is made. Therefore, the initial call of the `add_reservation_for_process` triggers the reservation for all processes and tasks of the job.

[10] This means, the overlap logic of the `ProcessReservations` do not apply to `TaskReservations`.
[11] No additional `ProcessReservations` to take into account.

```
def add_reservation_for_process(self, job_id, deadline: int, process: job.Process,
start, additional_p_res: [ProcessReservation], schedule_try=0)
-> (TaskReservation, int):
    curr = process.task_head
    actual_process_start = -1
    actual_task_start = start
    latest_res = None
    pref_core = None
    p = None
    add_p_res_for_child = additional_p_res
    child_join_to_task = {}
    join_in_task = -1
    while curr != None:
        if curr.process_id != process.process_id and curr.action == 3:
            join_in_task = curr.task_id
            break
        if curr.action == 3 and curr.process_id == process.process_id:
            if curr.task_id in child_join_to_task:
                last_child_res = child_join_to_task[curr.task_id]
                actual_task_start = last_child_res.end
        latest_res, pref_core = self.add_reservation(pref_core, actual_task_start,
        deadline, curr.instructions, curr.needed_memory, job_id, process.process_id,
        curr.task_id, additional_p_res, schedule_try)
        if actual_process_start == -1:
            actual_process_start = latest_res.start
        actual_task_start = latest_res.end
        if p == None:
            p = ProcessReservation(pref_core.node_id, pref_core.core_id,
            actual_process_start, -1, job_id, process.process_id,
            process.approx_needed_memory)
            add_p_res_for_child.append(p)
        if curr.action == 2:
            tmp_res, tmp_join_task = self.add_reservation_for_process(job_id,
            deadline, curr.child_process, latest_res.end, add_p_res_for_child,
            schedule_try)
            child_join_to_task[tmp_join_task] = tmp_res
        curr = curr.next
    if latest_res == None:
        raise ReservationException("No process with actual task to map provided")
    p.end = latest_res.end
    try:
        pref_core.add_process_res(p)
    except Exception as e:
        raise ReservationException(str(e))
    return latest_res, join_in_task
```

Listing 5: add_reservation_for_process Function Implementation

The function iterates over all tasks of the process and handles the reservation based on the action type of the task. Tasks with the action type fork (=2) or join (=3) trigger an additional handling that will be executed during the reservation process. For all tasks, the general reservation logic is relevant.

5.1.4.1 General Reservation Logic

The general reservation logic starts in line 13 with the iteration over all tasks of the process.

In the loop, the logic continues in line 21-23 with the function call of `add_reservation`. That function handles the actual reservation for a task. This includes finding the time frame which will ensure that the task finishes as early as possible, creating the `TaskReservation` based on that time frame, and adding the reservation resource to the linked list of the specific core. If it is the reservation for the first task of a process, all cores will be checked to find the time frame for which the task would finish the earliest. Otherwise, the prototype will search for the earliest time frame possible on the core that was used for the previous task of the process. Possible, in the context of this thesis, means that the resource requirements can be satisfied and that the time frame is at a later point in time than the previous task of the process.

In line 26 the desired start for the next task gets saved. In the next iteration, this value will be used to find a time frame that starts at the time or later. Earlier time frames are not seen as possible time frames and are ignored.

For each process line 27-31 handles once the creation of an incomplete `ProcessReservation`. Incomplete, means that the resource reservation is missing the actual value of its end field. The start field will be set with the start value of the first `TaskReservation` of the currently handled process, which was saved into a variable in line 24-25.

In line 38-39 a customized exception gets raised in case the `add_reservation` function got called without an actual `Task`. This should never happen and was just added to ensure that the prototype works as assumed.

In line 40 the end field of the `ProcessReservation` gets set. The value is the end value of the `TaskReservation` of the last task of the process. And in line 42 the, now completed, `ProcessReservation` will be added to the `ProcessReservation` linked list of the core.

5.1.4.2 Additional Reservation Logic for Join Tasks

In case the task currently handled is a join task, the prototype has to distinguished between the following cases.

Case 1

The task found is not part of the process that is currently handled by the function. This is the case when the prototype iterates over all tasks of a child process. When creating the PB task net, the last element of the `Task` linked list (iterated by going step by step through the next field of the `Tasks`) of a forked process is the join task of its parent. In that case, no actual reservation will be made as the task does not

belong to the currently handled process. Instead, the join task ID will be stored in `join_in_task` (line 14-16) and will be returned with the `TaskReservation` last (line 45 and 36). The iteration loop will be stopped and the function continues with line 38.

Case 2
If the join task belongs to the currently handled process, the reservations for the child process have been done and `child_join_to_task` contains the ID of the current task as a key. The value for that key is the `TaskReservation` of the last task of the child process. The key and value get added in line 36 when the `TaskReservation` and the task ID of the join task gets returned as mentioned in **Case 1**. The end value of the `TaskReservation` will be used as the earliest desired start for the join task (line 17-20). This ensures that the join task will only be scheduled after all tasks of the child process are finished. Otherwise, the join task of the parent might be scheduled too early and the parent process gets blocked which is not expected in the schedule plan.

5.1.4.3 Additional Reservation Logic for Fork Tasks

For fork tasks, the additional handling is done in line 32-36. `add_reservation_for_process` gets called after the reservation for the fork is finished. The function call triggers the reservation logic for the child process that was forked in the currently handled task. After the reservation of the last task of the child process is done the `child_join_to_task` gets updated as it has been discussed in the join task handling. Interesting to note are the parameters that are used for the `add_reservation_for_process` function call. The new reservation handling gets executed with the same job ID, deadline and `schedule_try` value. The `Process` instance provided to the function is one of the child processes. Furthermore, the provided start is the end value of the `TaskReservation` of the fork task. This ensures that the tasks of the child process do not get scheduled before the fork task is finished. `add_p_res_for_child` is not an empty list anymore. As explained before in the general reservation logic, in line 31 the list gets updated. It contains now the incomplete `ProcessReservation` of the parent of the child process. The `ProcessReservation` states to which core it belongs as it has the fields `node_id` and `core_id`. The list enables the prototype to take into account the reservations of the parent process (and higher[12]).

The recursive reservation logic for the child processes ensures that the reservation logic of the parent will have the necessary data concerning which child process will join the parent in each join task and the earliest desired start for the join task.

[12]Higher processes refer to processes that are higher in the "family tree". It does not refer to kernel-level processes.

```
def earliest_start(self, possible_start, instructions, add_p_res, needed_memory)
-> (int, int):
    def additional_p_res_mem(additional_p_res):
        if len(additional_p_res) == 0:
            return 0
        res_mem = 0
        for p in additional_p_res:
            res_mem += p.memory_to_res
        return res_mem
    add_p_res_mem = additional_p_res_mem(add_p_res)
    if (self.memory - add_p_res_mem) < needed_memory:
        return (INVALID_START, self.core_id)
    runtime = self.runtime(instructions)
    curr = self.reservation_head
    if curr == None:
        return (possible_start, self.core_id)
    if curr.next == None:
        if possible_start+runtime < curr.start and
        self.too_much_memory_res_by_p(possible_start, possible_start+runtime,
        add_p_res_mem, needed_memory):
            return (possible_start, self.core_id)
        return (max(curr.end, possible_start), self.core_id)
    while curr.next != None:
        if curr.next.start - max(curr.end, possible_start) >= runtime:
            potential_start = max(curr.end, possible_start)
            if self.too_much_memory_res_by_p(potential_start,
            potential_start+runtime, add_p_res_mem, needed_memory):
                curr = curr.next
                continue
            return  (potential_start, self.core_id)
        curr = curr.next
    return (max(curr.end,possible_start), self.core_id)
```

Listing 6: earliest_start Function Implementation

5.1.4.4 Task Reservation Logic

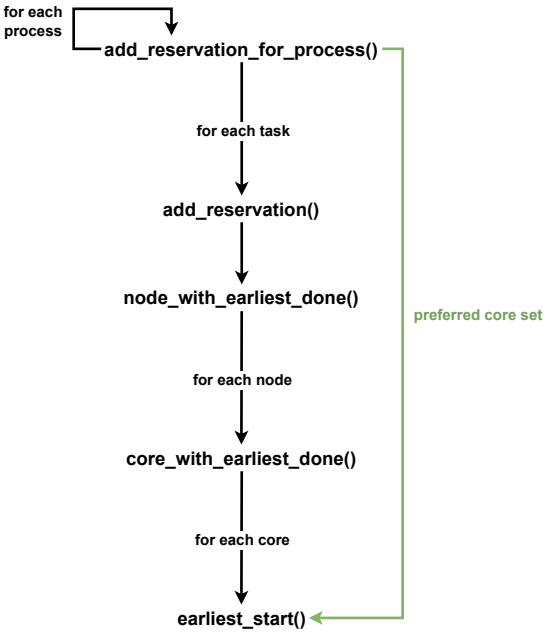

Figure 8: Reservation Function Call Overview

The function `add_reservation_for_process` handles the order in which the process and task reservations are triggered and sets the desired start times to ensure that tasks and processes do not run in an invalid order. The `add_reservation` function handles the reservation for every task of a process. It works similarly to the test mapping logic.
In case the currently handled task is the first task of a process, the core to which it should be scheduled is not fixed yet. The function requests a list of all `NodeReservations` with `CoreReservations` that have access to enough memory to generally satisfy the necessary memory reservation of the process. Afterward, the prototype iterates over all `NodeReservations` and their `CoreReservations` and calls the function `earliest_start` to find the earliest possible time frames for each core. The runtime of a task depends on the node/core to which it is scheduled. This is because the cores of different nodes can have a different tact frequency. Therefore, the final test frame that will be chosen is not necessarily the time frame with the earliest start, but the one with the earliest end. Then it will use that time frame to create the `TaskReservation` and add it to the `TaskReservation` linked list of the core from which it retrieved the chosen time frame.
If the task currently handled is not the first task of the process, the prototype has handled the reservation of a task of the process before. The `add_reservation` function returns the `TaskReservation` that was created and the `CoreReservation` to which it was added. The prototype stores the `CoreReservation` and will strictly prefer it in the task reservation of all following tasks of the process. In the context of this prototype, no other core will be taken into account if a preferred core is stored for the process. The prototype calls directly `earliest_start` for the preferred core to find the earliest possible time frame in the core and uses it to create the task reservation resource. Figure 8 displays the function call flow with and without a preferred core set.

The function `earliest_start` finds the time frame for a specific core. The implementation of the function can be seen in Listing 6.

It is important to note that `add_p_res` is a subset of `additional_p_res`. The functions `node_with_earliest_done` and `core_with_earliest_done` filter out all `TaskReservations` which are not relevant for the `CoreReservation` that is currently checked. `add_reservation` does the same in case a preferred core is set and the `earliest_start` gets called directly in that case. Therefore, `add_p_res` is a list of `TaskReservations` that are currently relevant. Relevant `TaskReservations`, are those that concern the core that is currently checked. Furthermore, a `TaskReservations` that was made for the process that is currently handled will also be removed. Otherwise, the reservation would be seen as made for another process.[13]

First of all, a check is executed to verify whether parent (or higher) processes of the job have already reserved memory for that core. To determine the total memory reservations created for the parent (and higher) process, the sum of the memory reservations of all `TaskReservations` in `add_p_res` is calculated (line 3-10). The calculated total memory reservation value (*total_mem_res*) is subtracted from the total memory available to the core ($total\_mem_c$). If the result is smaller than the necessary memory requirement of the task, the task cannot find a suitable time frame in that core. This is due to the fact that the parent and higher processes will finish after their children processes have finished and therefore no memory reservations that were used in the calculations will be freed on that core during the lifetime of that child process. `INVALID_START` is returned as a start point to indicate that no valid start was could that specific core. If there is enough memory remaining the function continues with calculating the runtime. The runtime has to be known to determine how long the time frame has to be (line 13). The runtime calculation is done by multiplying the `instructions_pre_sec` field value of the `CoreReservation` with the number of instructions the task consists of.

If there is no the `TaskReservation` linked list is empty, it can be concluded that no `ProcessReservation` exists for that core as well. The possible start time provided in the function call gets returned with the core ID to indicate to which core the returned start time belongs (line 15-16).

In line 17-22 the scheduler handles the case that only one `TaskReservation` is found in the linked list, in the following $task\_res_1$. In that case it will be checked if it is possible to make the reservation for the task before $task\_res_1$. If `possible_start` is greater than $task\_res_1.start$ it is not possible to do a reservation before $task\_res_1$. In that case, the returned start time is `max(curr.end, possible_start)`. This ensures that the start time is a point in time after the last existing `TaskReservation` of the list and that the possible start time is taken into account. If it is possible to do the reservation before $task\_res_1$ it has to be verified that the available memory of the core satisfies the memory requirement. If that is the case the possible start time gets returned. The function `too_much_memory_res_by_p` checks whether there is enough memory available while taking into account the existing `TaskReservations` in the linked list of the core that apply to the potential time frame as well as the `TaskReservations`

[13] Based on the implementation of the prototype, these `TaskReservations` should not be part of the `additional_p_res` list. However, cases were observed where that was the case. Therefore, the filtering was made stricter to remove these `TaskReservations` too.

from `add_p_res`. The function subtracts *total_mem_res* and the memory reservation that overlaps with the potential time frame from $total\_mem_c$ and checks if that leaves enough memory space for the reservation of the currently handled task. If there is not enough memory available `max(curr.end, possible_start)` is returned.

In line 23 the scheduler reaches the loop that iterates over the `TaskReservations` of the linked list. The first check in the loop is whether the time gap between the two observed `TaskReservations` is long enough to potentially run the task during that task. Additionally, the potential start (= `max(curr.end, possible_start)`) has to be equal to or greater than the possible start time. If both is the case the scheduler checks whether the memory requirements are also satisfied by calling `too_much_memory_res_by_p` (line 25-27). If the memory requirements are satisfied the potential start start time can be returned. Otherwise, the loop continues. If the end of the loop is reached, no suitable start time has been found yet. The greater value between the end value of the last element of the `TaskReservation` linked list and the possible start time get returned.

5.1.4.5 Retry Logic

The retry logic of the main PB scheduler prototype is kept rather simple. The prototype reuses most of its reservation logic. No separate scheduling/reservation technique is introduced. Instead, the difference between the first reservation try and the following tries is that the scheduler no longer determines the time frame that ends the earliest. The scheduler picks one of the provided time frames at random.

Imagen the reservation step fails on its first try. A ReservationException is raised during the reservation step. The customized exception gets caught and a clean-up is triggered. The clean-up removes all reservation resources from the `ReservationStore` that were added during the failed reservation try. Afterwards, it increases the `schedule_try` counter. If the `schedule_try` counter does not exceed the retry limit (`MAX_SCHEDULE_TRIES`) the scheduler re-triggers the reservation step with the increased `schedule_try` counter. The functions `core_with_earliest_done` and `node_with_earliest_done` return in the first try the time frame that would finish the earliest. Both functions request the time frames found for the node/core and get a list of possible time frames by node/core.

To randomize the pick in both functions, a random index is generated that points to one of the time frames in the given list. If the time frame that the index points to is a valid start time it will be used. Otherwise, the function continues as usual with determining the time frame that ends the earliest.

The topic of improving the retry logic will be discussed further in Section 7.

If the reservation step was successful in one of its reservation tries, the scheduler continues with the plan building step.

5.1.5 Plan Building

During the plan building step, the main PB scheduler creates a CSV file that contains an entry for each task of the currently handled job. To be able to create these entries,

```
{<node_id>:
        {'<core_id>':
                [{'start': <start>, 'end': <end>, 'task_id': <task_id>,
                'mem_consumption': <mem_consumption>},...]
                , ...
        }
        ,...
}
```

Listing 7: earliest_start Function Implementation

```
node_id;core_id;task_id;start;end;mem_consumption
5003;andi;0;0;1;400
...
```

Listing 8: CSV File Plan Content

the scheduler needs the start time, end time and memory consumption of the task. Furthermore, the node ID and core ID of the core to which the task should be scheduled have to be provided. All this data can be retrieved from the `TaskReservations` that were created during the reservation step. Therefore, the scheduler requests all `TaskReservations` of the job from the `ReservationStore`. The following describes the plan building logic in more detail.

The prototype calls `get_res_description_by_job` with the job ID as the `job_id` parameter value. In the following, we assume that the value of `job_id` is job_id_j. The function can access the reservation resources. `get_res_description_by_job` iterating over all `NodeReservations` and over all its `CoreReservations`. When a `CoreReservation` is reached, the prototype goes through the `TaskReservation` linked list and creates a dictionary for each `TaskReservation` for which its `job_id` field is equal to job_id_j. All dictionaries created for a `CoreReservation` will be stored in a list and added later on into a dictionary structure that is sorted by node and core ID. Listing 7 shows the final dictionary structure that will be returned from `get_res_description_by_job`.

If the returned dictionary is empty, an unexpected error has appeared and the plan build fails. Otherwise, a `plans/plan_<plan_id>.csv` file is created and the data of the dictionary is written to it. The result is a CSV file with a format that should look similar to Listing 8.

The presented main PB scheduler is a prototype so it simply creates a CSV file with the plan. When implementing a main PB scheduler that is supposed to run in a working HPC system, it would be more sufficient to use a database and send the plan content to it. It must be noted that the database should be accessible from the request server and the machine. This enables the machine to request up-to-date plans for each of its nodes.

5.1.6 Main PB Exception Handling

The main PB scheduler prototype defines three customized exceptions: `StructureException`, `ValidationException` and `ReservationException`. A `StructureException` is raised in case the initialized `ReservationStore` is unexpectedly missing resources. Currently, the exception is only raised when no `NodeReservations` exist. The other two exception types are introduced to enable the prototype to determine in which step the job handling failed. Furthermore, the prototype can have a separate exception handling for each exception type. This is a crucial part of the reservation retry logic, which was discussed in Paragraph 5.1.4.5.

In case the validation check fails, explained in Section 5.1.3, a `ValidationException` gets raised that contains a message with the specific reason why the check fails. A possible message could be that during the check the desired deadline was exceeded. If the prototype catches an exception that is not a `ReservationException` of if the retry limit was exceeded, the scheduler prints in which step the job handling failed and the exception message. This is done to notify the user that the requested job was declined.

5.2 Kernel PB Scheduler

The kernel PB scheduler prototype is the second part of the prototype. It is implemented as an extension of the kernel schedule module of the Linux code base, as mentioned already in Section 4.5. The C coding is located in the private `master_thesis_linux_kernel` repository in branch `model-draft`[14]. The prototype was not implemented in a working HPC system that is running the main and kernel PB scheduler. The used Linux code base does view programs in job, process and task definition that were introduced with the PB scheduler concept. However, a Linux kernel does view programs in processes and threads[15]. In the context of the kernel PB scheduler prototype, Linux processes are seen as PB processes. The closest concept, in the Linux context, to the PB task is the Linux thread concept. Linux threads do not have only one purpose like PB tasks. A PB task has only one action type, for example, communication and contains only the instructions used for the set action type. This means a task of action type communication does not contain calculation instructions, but only instructions to send e.g. a message. A Linux thread is created by a process and can be run in parallel to other threads of the process. Thus, Linux threads and PB tasks do not necessarily behave similarly. However, as there is no Linux concept that is more similar to PB tasks, I decided to neglect the differences between Linux threads and PB tasks for the implementation of the prototype.

Furthermore, it is important to note that Linux threads are implemented as lightweight processes that share their memory space with threads of the same Linux process. The difference between Linux processes and Linux threads (lightweight processes) is that Linux processes all have a process ID, in the following pid, that is unique to each other. Linux threads do also have a pid, but the value of the pid is the ID of the

[14] https://github.com/099111114097/master_thesis_linux_kernel/tree/model-draft

[15] Note that Linux kernel also has a concept of tasks which are a part of Linux thread. The task of a thread is the executable code. Thus, the definition of PB tasks and Linux tasks is quite different.

process that they belong to. Still, the Linux kernel itself sees no difference between Linux processes and Linux threads.

Details on how the development environment was set up for the kernel PB schedule prototype are presented in Appendix A.1.

5.2.1 Memory Consumption Model

The schedule handling in a Linux kernel is triggered by the time interruption that occurs periodically. The time interruption is based on the tact frequency of the Linux kernel. In the following, an execution of the schedule handling will be called a schedule session. The kernel PB scheduler prototype retrieves the total memory consumption that can be observed during each schedule session and updates the maximal memory consumption data of the concerning process in the MemC model. The scheduling describes the situation that a kernel scheduler interrupts the thread/process running on a core, determines which process should be assigned to the core and executes the new mapping. The improvement to extend the prototype to store the memory consumption peak of each task is discussed in Section 7.

The MemC model is implemented as a C struct, called `mem_model`. The struct points to the head of a linked list of process entries. As process entry is a C struct, `process_entry`, that contains the process ID, the maximal memory consumption and a pointer to the next process entry. This struct gets initialized once as a static struct. Initialized as a static struct, it is ensured that the struct will get reinitialized which would override its data.

All kernel scheduler types call the `__schedule` function that is located in the Linux kernel scheduler module[16]. The model gets initialized at the beginning of `__schedule`. I chose this place as the memory consumption data retrieval is also executed in that function. It ensures that the model is already initialized before the memory consumption data of a process is retrieved for the first time and that the model will be accessible.

5.2.2 Memory Consumption Data Retrieval

The memory consumption data for a process gets retrieved when the scheduler has determined the next Linux task that should be run on a core. This means it determines the Linux thread that should be assigned to a core. When both, the previous and the next task are known, the prototype requests the `mm_struct` of the previous task. This is all happening in the `__schedule` function. The struct `mm_struct` holds the information of the process memory space. That includes the `total_vm` field which states the total number of pages mapped by the Linux task in the current thread execution. It is live data that can change over time. Additionally, it is not easily possible to determine if the previous running thread/task was the last thread/task of a process. Thus, it is not possible to request the `mm_struct` only once when a process is about to finish. Therefore, the prototype retrieves the data and updates the model during each schedule session. This ensures that the data are as up-to-date as

[16] https://github.com/torvalds/linux/blob/master/kernel/sched/core.c

possible. The `total_vm` field is used as the maximal memory consumption that was found during the current execution time of the thread.

The `mm_struct` is only valid if the previous running Linux task/thread is not part of a kernel-level process. Otherwise, the struct is empty. All processes scheduled by the PB scheduler are user-level processes. Therefore, there is no issue with `mm_struct` being only valid for user-level processes.

5.2.3 Memory Consumption Data Update

When the maximal memory consumption data was successfully obtained, the function `update_process_mem_c` gets called. This function has to be called with a pointer to the `mem_model`, the memory consumption value and the pid. The function can access the model and go through all process entries. If no process entry was found with the requested pid, a new process entry gets initialized and added to the linked list. Otherwise, a process entry for the requested pid does exist and the stored and provided memory consumption value gets compared. If the provided value is greater, the stored value gets replaced by the provided value. The prototype prints out a message if a process entry gets added or when the stored value gets updated. After `update_process_mem_c` is finished, the `__schedule` function continues with the assigning of the next task.

,

6 Evaluation

The evaluation is divided into two prototype parts: the main PB scheduler and the kernel PB scheduler prototype. Both are evaluated based on the criteria defined in Section 2.2.

As the prototype parts are not built to run in the same system, no measurements, such as benchmark testing, were made. Therefore, the evaluation is based on the evaluation criteria which do not include analyzing measurements, such as performance data.

6.1 Main PB Scheduler

In this subsection, the functional and nonfunctional evaluation criteria for the main PB scheduler prototype are discussed.

6.1.1 Executable

The prototype does satisfy the executable criterion. It is a Python program that can be executed and provides a Command Line Interface that accepts job requests. In case all steps of the job handling are successful, the prototype produces a scheduling plan and stores it in a file in CSV format.

6.1.2 Flexible

The memory consumption prediction model data is taken into account by the prototype. It is assumed that the data is combined with the task net. The prototype retrieved the memory consumption data from the resulting PB task net. The prototype does not cache any memory consumption data for the following iterations. Therefore, the prototype can, theoretically, handle the scheduling of a job that has been scheduled before. This is due to the fact, that it processes the concerning PB task net every time a job request is received. Even though there is no issue with handling the memory consumption data, the prototype currently does not accept a job request for the same job more than once. This is because the job and process data as well as the PB task net are currently stored in the prototype itself. As a result, the same start time and deadline time would be set and the user does not provide new data. It is also necessary to note the PB task net is currently not updated after each job execution as this is out of the scope of this thesis.

Furthermore, if a job is scheduled twice, the logic in the `ReservationStore` would not be compatible as the reservation data of the first schedule still exists and there is no logic to differentiate between the reservation resources of different job executions of the same job. The issue in the `ReservationStore` could be simply resolved by adding a unique suffix to the job ID for each new job execution of the same job. Thus, the prototype does show flexibility to a degree that makes it efficient enough as a prototype. However, to implement a functional main PB scheduler based on this prototype some improvements have to be made.

6.1.3 Tolerable

The reservation handling has a retry logic in place to satisfy the tolerable criterion. It can be argued that the reservation logic for retries is not very sophisticated. There is room for improvement in form of potentially introducing additional scheduling techniques that replace the random pick functionality. However, it has to be noted, that the earliest done logic used in the reservation is sufficient in the manual tests done during the development. Furthermore, the scope of the prototype did not include the introduction of multiple scheduling techniques. Therefore, the implemented prototype does satisfy the tolerable criterion.

6.1.4 Filtering

Every job that was requested to be scheduled on the machine has to pass a 3-level check. This check filters out jobs that have no valid deadline set or that require in general an amount of resources that cannot be satisfied by the machine (level 1). Furthermore, the filtering logic checks the currently available resources during the potential run time of the job and verifies if they can roughly satisfy the estimated memory requirements (level 2). The most complex part of the check is level 3. This level executed a test mapping of the processes of the job while taking into account aspects of process scheduling. This includes the aspect that child processes cannot start earlier than its parent processes and processes allocate the memory needed at the beginning of its execution and only deallocate them at the end of their lifetime. The test mapping of level 3 is close to the actual reservation handling. Therefore, it does give a good estimation of whether the processes could be executed based on the currently available resources. Therefore, it can be stated that the prototype successfully satisfies the filtering criterion. An important assumption that is necessary, to support that argument, is that the approximated data used in the 3rd level of the check are close enough on the actual runtime and memory consumption data.

Additionally, it is recommended to set the value of `PROCESS_IDLE_PRECENTAGE` rather small. The functionality of estimating the idle time of processes and using it is an interesting concept. However, if the `PROCESS_IDLE_PRECENTAGE` is high, it assumes that a process is barely running which, in general, should not be the case in HPC systems.

6.1.5 Modular

The majority of the prototype is implemented in a modular way. The job data are located in a separate folder and can be requested based on the job ID. The PB task net and exception classes are implemented in separate packages. The processing of data and validation are located in their separate packages as well. An exception is the `ReservationStore` and the majority of the reservation logic. As the reservation logic needs to access `ReservationStore` and the connected class instances, I decided to implement most of the reservation logic as class methods to get the necessary access to reservation resources. The decision helped with keeping the code understandable and simplifying the implementation as it was possible to use the `self` type of a resource

instead of providing every function each function call with the concerning reservation resource pointer. The one shortcoming is that the introduction of an additional reservation/scheduling technique does make it necessary to adjust some parts of the reservation coding in the `ReservationStore`.

Still, I argue that the prototype is implemented to a sufficient degree of modularity. The potential adjustments are within reasonable limits. Additionally, the coding does make sure that the class methods do not exceed their reasonable access limits. For example, a class method of the `NodeReservation` class does not directly access `TaskReservations` but comply with the class structure. This also ensures that the replacement or extension of the reservation logic is possible and kept simple enough.

6.1.6 Stability

All exceptions, expected and unexpected, are caught by the prototype. A raised exception does not break the execution of the prototype. It does terminate the current job request handling as an exception indicates that the job is not suitable to be executed. The one aspect of the exception handling that could be improved is the exception handling for `StructureException`. It would be reasonable to terminate the prototype in case the initialized structure is flawed. However, that is a small potential improvement. The exception message of a raised exception will be printed with some additional information that states, if possible, in which step of the job handling the exception occurred. This is done to provide the user with a reason why their job was declined. Based on the implemented exception handling that does catch also general exceptions, the prototype does show a stable behavior in its execution.

6.2 Kernel PB Scheduler

In this subsection, the evaluation criteria presented for the kernel PB scheduler are discussed.

6.2.1 Executable

The prototype does satisfy the defined executable criterion. The coding is executable and will be called during each schedule session. It is ensured that the prototype code will be executed as it is called in the `__schedule` function of the Linux kernel scheduler module. The `__schedule` is part of the general schedule framework of the kernel. The function calls `pick_next_task` which will continue with triggering the currently prioritized kernel scheduler. This means, that code introduced in `__schedule` will be called for each schedule session no matter which kernel scheduler is used during the session. The coding will once create the model. If the model provided to the update function does not exist, a notification will be printed, but no exception will be raised. This is done to not influence the kernel and the processes/threads running on the cores. The memory consumption module and the retrieval of the data is a functionality that is useful for the main PB scheduler. However, an error in its execution should

not break the kernel or any currently running program. Furthermore, the prototype code ensures that the module is initialized before it is updated.

6.2.2 Modular

The coding is modular. The MemC model is implemented as a C struct and methods are introduced to find, update, and create process entries. Introducing another model can be done easily. The only potential improvement is that the introduced coding could be moved to a separate file and a header file could be introduced. This would improve the code structure and make the interfaces even more easier to understand. Since the introduced struct definitions and functions are short and clearly implemented, as well as, the master thesis is created on a time limit, I chose to not spend more effort into moving the code.

6.2.3 Stability

The prototype is the extension of a kernel module and is part of the scheduling framework of the kernel. A kernel is a complex program that does hold the risk of running into one or multiple issues due to different root causes. It has to be noted that the prototype coding does not make the kernel more stable. If an issue occurs that will break the initial Linux kernel, the same will happen with the kernel that contains the prototype code. However, increasing the stability of the kernel was not the focus of the prototype. The focus is to ensure that the introduced code is stable and does not introduce issues. During the creation of the MemC model and the process entry structs memory is allocated for each of them. Furthermore, the functions check whether the to be accessed struct instances are valid and can be accessed. They ensure that no null pointers are accessed. Otherwise, this can result in memory access errors as the memory address that the prototype tries to access is not valid. In short, the coding ensures that it is stable and safe to be executed.

7 Future Work

The presented main PB and kernel PB schedulers are prototypes. This means they are not meant to be the final solution. Their purpose is to test concepts that could potentially improve the current concept of the PB scheduler. During the previous sections of this thesis, simplifications, shortcomings and potential improvements were mentioned. This section will present and discuss the collection of potential improvements. Furthermore, the adjustments that should be made to make the newly introduced concepts applicable for actual HPC systems running the PB schedulers are discussed.

7.1 Improvements

This subsection presents a collection of all potential improvements that can be done to improve the main PB or kernel PB scheduler prototype. The improvements are organized in the following list. All items, except the last, are concerning improvements that can be done in the main PB scheduler prototype. The last item concerns an improvement that would make it necessary to adjust both prototypes.

1. PB Task Net Validation
   The current job validation logic checks in its 1st level that the job has a possible deadline and that the memory requirements can generally be satisfied with the capacities of the machine. The validation could be improved by extending the 1st level of the check to go through all `Tasks` of the PB task net and verify that the structure is correct. A PB task net with a valid structure has for each fork task a fork end task (belonging to the forked child process) and a join task. Additionally, the net must have exactly one start and end task. In the current state of the prototype, a job with an invalid task net would fail the reservation handling as the prototype as either a general exception gets raised as the prototype tries to access empty pointers or because it cannot find tasks to schedule. This additional check would stop the job request earlier for jobs that have no valid task net and could provide a clearer exception message.

2. Test Mapping
   Overlapping of active `ProcessReservations` are not taken into account during the test mapping. This could be improved using a similar logic used in the reservation handling which uses the `too_much_memory_res_by_p` function to check if enough memory is available.

3. Idle Time Calculation
   Currently, the concept of idle time in `ProcessReservations` uses only a flag that is set during the test mapping. This neglects the possibility that there are already tasks mapped during the time that the process of the `ProcessReservations` is not running. This means, another `ProcessReservations` already overlaps with the `ProcessReservations` in questions. Thus the idle time is already used in the reservation/scheduling. To improve the test mapping in the aspect of idle time estimation, a `idel_used` flag could be introduced besides the `test_idel_used` flag. The new flag would focus on actual `ProcessReservations`. If the

`idel_used` is updated for a `ProcessReservations` it should also update `test_idel_used` accordingly. This ensures that the idle time estimation will be closer to reality.

4. Reservation - Memory Reservation Estimation
   Currently the function `too_much_memory_res_by_p` sums up all `ProcessReservations` that overlap with the potential time frame checked during task reservation handling. It does not take into account that the `ProcessReservations` could overlap with the time frame while not overlapping with each other. This means, that not all memory reservations overlapping with the time frame must exist at the same time. Therefore, the calculated total memory reservations value might be too high and a later time frame gets returned even though the time frame is actually suitable.

5. Additional Reservation Scheduler
   It would be interesting to introduce additional reservation/scheduling techniques that would be executed in the retries or even replace the current reservation schedule logic of picking the earliest done time frame. If the main PB scheduler prototype concepts are introduced to an actual HPC system running the PB schedulers, it would be interesting to observe the actual performance of multiple scheduling techniques. Note that all added scheduling techniques should take the memory consumption data into account.

6. Memory Consumption Model
   Instead of storing the maximal memory consumption of a process the kernel PB scheduler prototype could be extended to additionally store the memory consumption peak of each task. This could improve the estimation value of the maximal memory consumption data for each process.

7.2 Adjustments for HPC systems

The prototype developed within the scope of this master thesis is a tool to educate oneself with possible concepts that can be used to tackle the topic of memory consumption in the context of the PB scheduler in HPC systems. The prototype does realize possible concepts that can be used to implement a PB scheduler that takes the memory consumption of processes into account. However, due to the nature of prototypes, it comes with limitations. Still, the current implementation of the prototype parts are fully applicable implementations of the main PB and kernel PB scheduler. This subsection focuses on the necessary adjustments or extensions that should be done to make them functional in a real-life HPC system.

7.2.1 Communication

The main PB and kernel PB scheduler prototypes are currently isolated from each other. To make them applicable for an actual HPC system they would have to communicate with each other. To realize direct communication, it is important to make sure that the communication is safe. The data could be encrypted before it is sent. Additionally, it is important that the data can be received by the machine. Therefore,

TCP is recommended. The main PB scheduler would directly send the created plans and the kernel PB scheduler would send the updated MemC model values to the main PB scheduler. A solution for the indirect communication option is to set up a database that can be accessed from the request server and the machine. The main PB scheduler prototype would send the created plans to the database and would request the current MemC model of a job from the database when the job is again requested by a user. The kernel PB scheduler would periodically request the plan for the specific nodes and send the updated MemC model data of a job after the job is finished. Assuming that a job will not be instantly requested again right after it has finished, there would be no inconsistencies between the MemC model data temporarily stored in the main PB scheduler and the data stored in the database.

7.2.2 Data Storing

All data concerning the machine and the jobs[17] are currently stored in files under the folder `data`. In an actual HPC system, this causes multiple issues. The local storing of data is hard-coded. If the machine details change, due to, for example, new nodes being added to the machine or the shared memory being improved, the main PB scheduler prototype has no knowledge about the new details. They would have to be manually updated. In the case of the job details it is even more crucial. Currently, it is assumed that the data of a job is already available in the prototype source code before it is requested. However, in reality, the data should be sent by the user and not stored in the prototype itself. A possible improvement would be to re-use the database from 7.2.1 or create one in case it was done to tackle the topic of communication. This would also simplify the process of updating the PB task net with the new data retrieved from the kernel PB scheduler. Currently, it is simply assumed, based on the scope of the master thesis, that the prototype is not responsible for updating the PB task net. Additionally, the machine details could be requested from the machine and either be directly sent to the database or to the request server which will send the updated details to the database.

7.2.3 Time Formats

The data type integer was used as a time format in the main PB scheduler to indicate time points, such as the start and end of time frames. This simplified the building of the concepts as well as calculations that are made based on time points. In real-life applications, the relative time points are not sufficient. Actual timestamps can be introduced. However, the consequence is that most of the calculations made in the job valuation and reservation step would have to be adjusted. Another solution is to introduce a converter that converts the relative time points into absolute ones, meaning timestamps. Thus, the main PB scheduler could work with relative time points and the time points would be converted either during the plan building or in a newly introduced step. Furthermore, deprecated reservation resources should be removed either way to not take up more memory than necessary.

[17] In the current context, the job details also include the process details, as well as, the PB task net.

7.2.4 Memory Units

The memory unit used in the main PB scheduler is never specified. I implemented the main PB scheduler assuming the the memory date in the machine details and the memory consumption data are in the same format. The memory consumption data retrieved by the kernel PB scheduler is in the format of the number of memory pages. This is not a common memory unit to state the capacity of a machine. Therefore, a converter logic should be introduced that converts the memory consumption data (and potentially the machine detail data). This shall ensure that all data are in the same format. The chosen format has to be suitable to indicate the shared memory available and the memory consumption of processes and tasks without running into issues such as integer or floating point overflows. An overflow could occur if a memory data type is chosen as the format for which the machine detail data is too large. For example, the shared memory value of a node could be larger than the maximum possible value that can be assigned to a variable of the chosen data type.

7.2.5 Kernel Scheduler

The kernel PB scheduler prototype of this master thesis does not implement an actual kernel PB scheduler. Therefore, it would be necessary to combine the implementation of the prototype with an actual kernel PB scheduler and make sure that the coding is only executed if that scheduler is currently selected. The research and development of the PB schedulers is still in progress. However, the prototype coding of this thesis could be combined with the prototype of [4] to further investigate the potential of the newly introduced concepts of this thesis.

8 Conclusion

Good morning, eager young minds

A Beautiful Mind (2001)

This thesis dealt with the memory consumption of processes/tasks and how it can be taken into account by PB schedulers in the context of HPC systems. The main value of the thesis is the concepts and design choices that resulted in the main PB and kernel PB scheduler part of the prototype. One of the greatest questions was how the resource reservation can be sufficiently tracked. Additionally, the prototype also opens up questions about potential improvements that were presented in Section 7. I acknowledge that the introduced concepts and the created prototype does work with several assumptions and some simplification. However, all assumptions and simplifications do not cause the prototype to be designed for an unrealistic system. The prototype is simply created for a scope of cases. This means the prototype can handle a job, as long as the job meets the assumptions. An example of jobs that are not compatible is for example jobs using synchronous communication. A simplification that was made for the prototype is the static memory management in the Utilization model. The prototype was designed in a modular way. Therefore, it can be extended to support other utilization models, for example, one that uses dynamic balanced memory management. Furthermore, the support for synchronous communication can be introduced as well. Due to that, the prototype can be seen as a successful implementation of a main PB scheduler and kernel module extension that can be helpful in the research topic of PB scheduling. The prototype realized the first steps presented in Section 2 and can be further developed or combined with for example prototypes of the area.

Finally, the topic of taking the memory consumption of process/task into account is a promising area of PB scheduling and HPC itself. It has the potential to improve HPC schedulers in general, as long as the jobs running on the HPC system are running repeatedly. In the context of PB scheduling, it can improve the job validation as well as the scheduling. Furthermore, it can prevent the situation that the created plans cannot be executed as desired, because the memory available to cores is not satisfying the necessary memory size. In summary, the topic of the master thesis does show importance for the research of PB scheduling and will also be relevant in the future. The topics of efficiency and utilization are costly and complex challenges in HPC. If the PB scheduling concept is introduced in HPC systems, also outside of the university institute, the importance of this thesis topic will also increase in the future.

Bibliography

[1] Hirbod Behnam. Compiling the Linux kernel and creating a bootable ISO from it. https://medium.com/@ThyCrow/compiling-the-linux-kernel-and-creating-a-bootable-iso-from-it-6afb8d23ba22. (accessed: 25.02.2024).

[2] Max-Georg Debbert. *Communication between a Plan Based Scheduler and User Space Processes in Linux*. Bachelor's thesis, Freie Universitat Berlin, DE, 09 2021.

[3] Matthias Diener. *Automatic task and data mapping in shared memory architectures*. PhD thesis, University of Illinois, USA, 01 2015.

[4] Kelvin Glaß. *Plan Based Thread Scheduling on HPC Nodes*. Master's thesis, Freie Universitat Berlin, DE, 03 2018.

[5] Fabian Kovacs. *Modelling Communication behaviour of Parallel Programs - Überwachen und Strafen*. Master's thesis, Freie Universitat Berlin, DE, 09 2018.

Glossary

Asynchronous Communication is a non-blocking communication style that is not waiting until a reply is received..

Memory Page is a virtual memory block of a fixed length.

Cache Flush describes the situation that all data gets removed from the cache.

A Appendix

A.1 Kernel Development Environment

The environment used to verify the kernel PB scheduler prototype during development consists of the following setup. The Virtual Machine Manager[18] is used to run a VM locally. QEMU[19] emulates a running kernel in the VM. I followed the instructions from [1] to run QEMU with the extended kernel. To make QEMU emulate the extended kernel it is provided with a kernel image and a filesystem. The kernel image is created by compiling the extended Linux kernel code. The filesystem is created via BusyBox[20]. The following command emulates the kernel using an explicitly set kernel image `arch/x86/boot/bzImage` and filesystem `initramfs.cpio.gz`:

```
qemu-system-x86_64 -kernel arch/x86/boot/bzImage -initrd initramfs.cpio.gz
```

A.2 Tables of Class Attributes

[18] https://virt-manager.org/
[19] https://www.qemu.org/
[20] https://busybox.net/

Class	Attribute	Data Type
ReservationStore	node_head	NodeReservation
NodeReservation	node_id	Integer
	shared_memory	Integer
	instructions_pre_sec	Integer
	core_ids	List of Strings
	core_head	CoreReservation
	next	NodeReservation
CoreReservation	core_id	Integer
	node_id	Integer
	memory	Integer
	instructions_pre_sec	Integer
	task_res_head	TaskReservation
	process_res_head	ProcessReservation
	next	CoreReservation
ProcessReservation	core_id	Integer
	node_id	Integer
	start	Integer
	end	Integer
	job_id	Integer
	process_id	Integer
	memory_to_res	Integer
	test_idel_used	Boolean
	active	Boolean
	next	ProcessReservation
TaskReservation	start	Integer
	end	Integer
	job_id	Integer
	process_id	Integer
	memory_reserved	Integer
	next	TaskReservation

Table 5: Utilization Model Class attributes

Class	Attribute	Data Type
Job	start	Integer
	deadline	Integer
	total_runtime	Integer
	total_needed_memory	Integer
	head	Task
	processes	List of Processes
Process	process_id	Integer
	approx_runtime	Integer
	approx_needed_memory	Integer
	forked	List of Processes
	task_head	Task
Task	process_id	Integer
	task_id	Integer
	action	Integer
	child_process	Process
	child_process_id	Integer
	comm_to	Process
	comm_to_id	Integer
	needed_memory	Integer
	instructions	Integer
	next	Task
	next_id	Task

Table 6: Task Net Class attributes