SPECIAL ISSUE PAPER OPEN ACCESS

Toward a Dynamic Allocation Strategy for Deadline-Oriented Resource and Job Management in HPC Systems

Barry Linnert[1,2] | Cesar Augusto F. De Rose[3] | Hans-Ulrich Heiss[2]

[1]Institut für Informatik, Freie Universität Berlin, Berlin, Germany | [2]Fakultät für Elektrotechnik Und Informatik, Technische Universität Berlin, Berlin, Germany | [3]Escola Politécnica, Pontifícia Universidade Católica Do Rio Grande Do Sul (PUCRS), Porto Alegre, Rio Grande do Sul, Brazil

Correspondence: Barry Linnert (barry.linnert@fu-berlin.de)

Received: 25 March 2024 | **Revised:** 17 September 2024 | **Accepted:** 1 October 2024

Funding: This work has been partially supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES).

Keywords: application runtime behavior | deadline-oriented policies | dynamic resource allocation | high-performance computing (HPC) | quality of service (QoS) | resource and job management system (RJMS)

ABSTRACT

As high-performance computing (HPC) becomes a tool used in many different workflows, quality of service (QoS) becomes increasingly important. In many cases, this includes the reliable execution of an HPC job and the generation of the results by a certain deadline. The resource and job management system (RJMS) or simply RMS is responsible for receiving the job requests and executing the jobs with a deadline-oriented policy to support the workflows. In this article, we evaluate how well static resource management policies cope with deadline-constrained HPC jobs and explore two variations of a dynamic policy in this context. As the Hilbert curve-based approach used by the SLURM workload manager represents the state-of-the-art in production environments, it was selected as one of the static allocation strategies. The Manhattan median approach as a second allocation strategy was introduced as a research work that aims to minimize the communication overhead of the parallel programs by providing compact partitions more than the Hilbert curve approach. In contrast to the static partitions provided by the Hilbert curve approach and the Manhattan median approach, the leak approach focuses on supporting dynamic runtime behavior of the jobs and assigning nodes of the HPC system on demand at runtime. Since the contiguous leak version also relies on a compact set of nodes, the noncontiguous leak can provide additional nodes at a greater distance from the nodes already used by the job. Our preliminary results clearly show that a dynamic policy is needed to meet the requirements of a modern deadline-oriented RMS scenario.

1 | Introduction

In recent years, high-performance computing (HPC) has become a much-used tool in many application domains, such as science and development, as well as business and industry[1] [1]. Many new applications in this broad range of fields are being developed to take advantage of the ever-increasing number of nodes present in the fastest HPC systems available, also known as supercomputers. This challenge of implementing new approaches to parallel programming that make meaningful use of the enormous computing power is accompanied by the need to embed these applications into high-level scientific workflows (the process for achieving a scientific goal detailing intermediate tasks and their dependencies) [2–4].

This use of supercomputers, as well as the integration into specific workflows, is familiar from science, where deadline-driven workflows are common, but is also important for many other application scenarios in development and industry. For example, when developing a new type of airplane engine, data from current engine versions are incorporated alongside objectives such as reducing noise and fuel consumption [5]. Simulations implemented as finite element method (FEM) therefore have to provide the results for optimizations of the engine blades or combustion chamber in time for the engineering team so that the prototype and subsequently the final product can be built as planned previously.

In this context, the resource management system (RMS), which is sometimes also referred to as resource and job management system (RJMS), is responsible for allocating HPC resources efficiently to incoming jobs to improve throughput and also has to guarantee job deadlines in order to provide the quality of service (QoS) requested by the users [6, 7]. Therefore, in a QoS-oriented environment, the RMS has to decide if a job request submitted to the HPC system can be accepted, taking into account the deadline of the job, when and where on the machine the job is to be executed [8]. Only when all questions are answered positively can QoS be guaranteed, and the job can be scheduled and afterwards executed on the HPC system. This is the only way to ensure that the deadline specified by the workflow in which the compute job is embedded is met.

This is a challenging task because the final execution time of a job depends on several factors that are hard to predict, such as resource demand and availability throughout the execution, their specific communication patterns, and how well these patterns match the machine's memory hierarchy and network topology [9–11]. The latter is especially true for any system that implements a multilevel network topology and is even more important in the context of the fastest supercomputers, such as the machines in the TOP 500 list (like Fugaku and Titan, respectively numbers 4 and 73 in the current ranking) [12]. When users are asked to give the execution times for their jobs, they generally fail to give good estimates because they have no running history or technical knowledge to make predictions or want to trick the system to get ahead of other users [13].

This work is an extension of a conference paper accepted at the High-Performance Computing Systems Symposium (former WSCAD now SSCAD) [15] that evaluates how well static resource management policies in traditional RMS systems cope with deadline-constrained HPC jobs and proposes two variations of a dynamic policy to meet the requirements of modern deadline-oriented scenarios. Building on our previous work, we have now investigated the influence of the shape of the partitions and the various forms of communication in more detail, implementing and evaluating a second approach for static allocation strategies called the Manhattan median approach, which results in more compact partitions than the Hilbert curve approach, reducing the communication overhead of parallel programs. It now focuses on the impact of compactness and the shape of the partitions in order to support different runtime behavior types (see Section 2). This leads to a deeper understanding of the relationship between allocation policy and runtime behavior in deadline-oriented scenarios.

Specifically, our contributions to these challenges in this article are as follows:

- We present an evaluation on how a state-of-the-art RMS like Slurm (with a static allocation strategy based on the Hilbert curve [14]) copes with deadline-oriented scenarios.
- We implement and evaluate a second approach for static allocation strategies, the Manhattan median approach, which implements more compact partitions than the Hilbert curve approach, reducing the communication overhead of parallel programs.
- We implement a dynamic resource allocation policy to the RMS and evaluate how a contiguous and a noncontiguous variant compares to the static policies.
- Based on the analysis of results obtained through an extensive number of simulations under different workload conditions, we demonstrate the advantages of dynamic policies over static ones in deadline-oriented scenarios.

2 | Parallel Applications and Load of HPC Jobs

Because the range of application domains that make use of HPC is enormous, there are different approaches to creating parallel applications and, consequently, different types of resource requirements for these programs [16]. In many cases, the parallel programs use message passing to distribute and collect data to be processed by the various processes that make up the running program—the compute job. In the era of multicore node architectures, the use of shared memory provided by these nodes also comes into play. Since the characteristics of the different programs resulting from the different programming approaches play an important role in providing efficient support for these parallel applications, a closer look at the runtime behavior of the programs is needed.

Parallel programs, implemented as so-called Monte Carlo applications, consist of a fixed number of processes processing randomly generated data. This is characterized in Figure 1, where light gray circles represent start and end nodes of the program, and light green circles represent the execution of instructions (computations) of the processes. The processes of the job run independently and do not exchange intermediate data. The only utilization of network resources occurs when input data is distributed to the processes. The results are received in order to be evaluated, and output is provided. This type of application is used primarily when statistical evidence is needed, such as for discrete event simulations that simulate technical entities like HPC or Grid environments. However, this type of

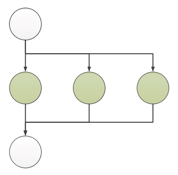


FIGURE 1 | Runtime behavior of a Monte Carlo-like program.

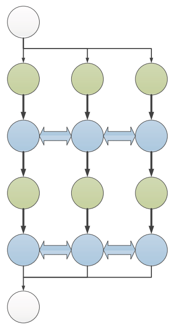


FIGURE 2 | Runtime behavior of a BSP-based program.

application is also used for implementing various approximation approaches and simulating physical processes. Nowadays, HPC systems are also used to train machine learning (ML) models. These types of applications also exhibit this type of runtime behavior.

Another type of parallel programs is also based on a fixed number of processes, but the processes do not run independently as characterized in Figure 2. Instead, they exchange data during processing in order to provide intermediate data or results to other processes and obtain new input data for the next step of computation itself. One of the most commonly used models for building such a parallel program is the Bulk-Synchronous Parallel Model (BSP) [17].

In recent years, the introduction of multicore CPUs on the one hand and the development of new features for the message passing interface (MPI) leading to the new standard MPI-2, as well as the increased use of OpenMP in combination with message passing approaches on the other hand, have led to the emergence of new types of parallel programs with new runtime behavior (see Figure 3). In order to utilize multiple cores of a CPU or the processors of a node, OpenMP is used to launch additional light processes (threads). The creation of additional processes can be based on the design of the application, so that the number of processes to be started is known before the job is started, or the start of additional processes can be controlled by the program at runtime. This creation of additional processes based on runtime decisions, for example, about the amount of work to be processed, has already been introduced in approaches such as the bag-of-task or the manager-worker approach but is increasingly used in applications implementing finite elements methods. Simulations of particles and molecules also benefit from this approach, which leads to the dynamic runtime behavior of the programs.

Figure 4 exemplifies that the creation of additional processes can be used in an even more flexible way when using the new features provided by the MPI-2 standard. With MPI-2, new processes can be created on different nodes and the start of the additional processes or threads is not limited to the local node, which is already used by some MPI processes of the job [18, 19]. The runtime behavior types presented here are only examples of very small programs. In real scenarios and during evaluation, the number of tasks reaches up to billions, especially for programs with dynamic runtime behavior [20].

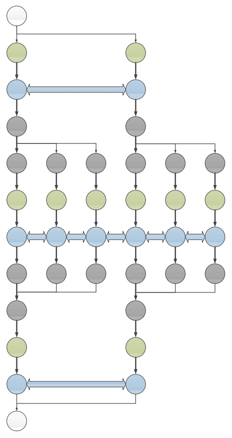


FIGURE 3 | Runtime behavior of a program using OpenMP.

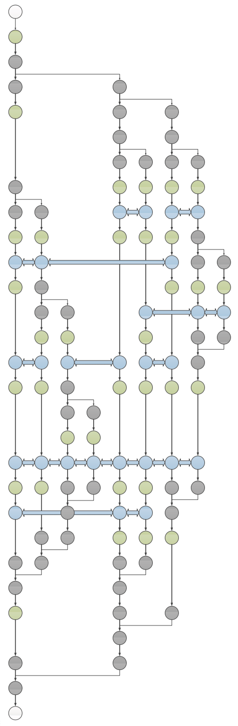


FIGURE 4 | Dynamic runtime behavior of a program using MPI-2.

Nevertheless, the RMS today—in addition to providing resources for the job—is more and more responsible for providing a certain level of QoS. When it comes to supporting higher-level workflows, this level of QoS is, as mentioned earlier, related to the reliable execution of the parallel program that is submitted to the HPC system as a job. In this case, reliable execution is often embedded in some form of contract—the Service Level Agreement (SLA). In a SLA, the service provider—the provider of the HPC resource—regularly guarantees the agreed QoS and would pay some kind of fee if the level of QoS is not achieved. Therefore, in these environments, the execution of the job must be completed before the deadline specified in the SLA. Hence, RMSs for HPC systems have to be examined to determine whether they currently meet these QoS requirements, and if not, new approaches need to be developed. Since there are different types of parallel programs running on the supercomputers, the examination has

TABLE 1 | Classification of parallel programs by runtime behavior.

Type	Creation of processes	Interprocess communication	Example
0	Static	No communication	Monte Carlo without communication
1	Static	Regular pattern	BSP programs
2	Dynamic on local nodes	Irregular pattern	FEM with mesh refinement using OpenMP
3	Dynamic on local and remote nodes	Irregular pattern	FEM with mesh refinement using OpenMP and MPI-2

to be performed with respect to these different types of applications. A summary of the different types of parallel programs is shown in Table 1.

3 | Static HPC Resource Management

When a parallel program such as described in Section 2 is submitted to the RMS by the user, current RMSs implement batch job-oriented, static allocation strategy assigning a set of computing resources for the entire execution of a job at start time. The parallel program then assigns the processes of the job to these resources at runtime, often with assistance from middleware and the RMS. To ensure that the job can be completed before a given deadline, the resources that the job requires for its execution have to be available at runtime, regardless of the resource demands of other jobs. With batch job-oriented RMS, the check for sufficient resources is performed at the time the job is selected to start. For deadline-oriented RMSs, it has been shown that reserving the required resources in advance is the basic method to provide the resources in time for the execution of the job [21]. Therefore, the check whether a sufficient number of computing resources are available in deadline-oriented systems is performed at the time the job request is submitted to the system. The request can then be accepted based on a successful reservation of these resources or has to be rejected by the RMS if there are not enough resources available to complete the job before the deadline.

3.1 | Hilbert Curve-Based Allocation

In addition to the scheduling decision, the performance of the local RMSs also depends on the mapping decisions. Various approaches for optimizing the mapping decision have been presented. One of these is the mapping approach of the SLURM workload manager [14], a queuing-based RMS widely used in the area of HPC that implements a batch job-based static strategy for handling job requests. In fact, it is the most popular RMS for the HPC systems summarized in the TOP500 list. The system can also be extended to support advance reservation [22] in order to provide a higher level of QoS needed to support deadline-oriented resource management.

Since the shape of the partition to which the job is assigned is important to minimize the communication overhead for many

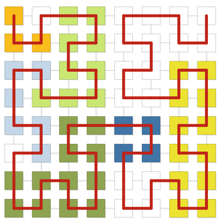


FIGURE 5 | Example of a mapping with Hilbert curve.

parallel applications and HPC systems, especially those implementing a mesh/grid or torus topology such as the previously mentioned Titan and Fugaku, the SLURM workload manager aims to provide compact, contiguous partitions to the jobs. For this purpose, the SLURM uses a shaping component that makes use of a Hilbert curve approach. Figure 5 exemplifies how a Hilbert curve, when used as a neighborhood relationship, which is the basis for creating compact partition shapes, is transferred from two- or more-dimensional grid or torus topology to a one-dimensional line of nodes (emphasized by a red line).

Based on this Hilbert curve, the SLURM workload manager uses a first fit approach (by following the red line) to find a partition with a sufficient number of computing resources for the job request and still form compact partitions that should help to reduce the communication overhead for the job. The partition formed using the Hilbert curve is then assigned to the job and can be used by the parallel program over the entire runtime, scheduled by the RMS. The SLURM workload manager uses the Hilbert curve approach primarily to support three- or more-dimensional grid or torus topologies. In order to investigate the approach, it has also been implemented for two-dimensional grid topologies.

Since the RMS has to implement advance reservation to support QoS based on SLAs, the Hilbert curve-based mapping—and therefore the SLURM manager itself—is extended to include an initial check for free computing resources and the booking of the number of resources assigned to a job, commonly called nodes in a HPC cluster. Figure 6 describes the steps that are performed when a job request is received. First, the number of free nodes available in the interval in which the job is to run is determined and compared with the number of nodes required by the job (line 0). If enough free nodes are available, a partition is searched using a first-fit strategy based on the list of nodes ordered by the Hilbert curve (lines 5–10). Only contiguous segments are considered. If a consecutive list of nodes is found that is available at the specified runtime of the job, these nodes are reserved for the job so that no other job will use the nodes in this time interval (line 11).

3.2 | Manhattan Median Approach

Another approach for mapping in HPC systems that focuses primarily on compact partitions in order to reduce the impact of other running programs and the communication behavior of the program itself is the Manhattan Median algorithm [23]. The approach was developed to reduce the overall Manhattan distance, and it has been shown to provide a $\frac{7}{4}$-approximation

```
  given is a list of nodes $nodes_{Hilbert\_curve}$ following the Hilbert curve
0  find valid time interval with enough resources with
     $t_{earliest\_start\_time} \leq t_{start} \leq t_{deadline} - t_{runtime} \wedge$
     $\forall t \in [t_{start}, t_{start} + t_{runtime}] : nodes_{sum\_of\_used\_nodes}(t) + nodes_{job} \leq nodes_{max}$
1  if valid time interval is found
2    $node_{start} := node_{first\_node\_of\_cluster}$
3    while $t_{start} + t_{runtime} \leq t_{deadline}$
4      while $node_{start} \neq node_{last\_node\_of\_cluster}$
5      select a set $nodes_{job}$ of nodes $nodes_{selected}$ based on $nodes_{Hilbert\_curve}$ at $node_{start}$
6        $\forall n \in nodes_{selected}$ :
7          if $\forall t \in [t_{start}, t_{start} + t_{runtime}] : free_{n(t)}$
8            save $n$ in $nodes_{map}$
9          else
10           $node_{start} := node_{next\_node\_of\_cluster}$
11       if number of $nodes_{map} = nodes_{job}$
12         return $(t_{start}, nodes_{map})$
13     get next valid time interval with $t_{start}$
14  reject request
```

FIGURE 6 | Overview of the Hilbert curve approach.

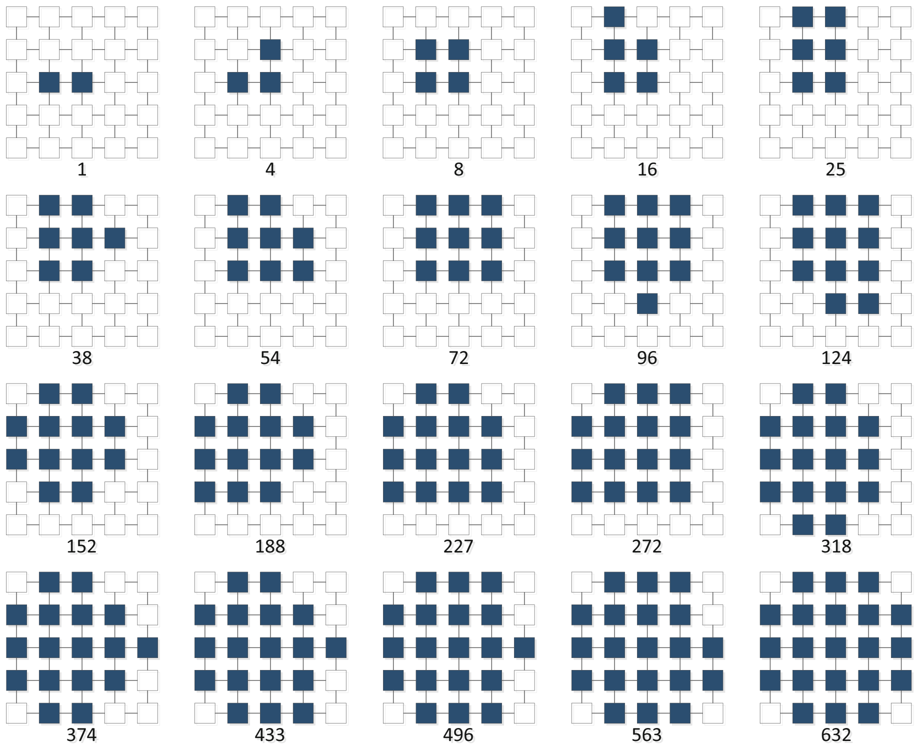


FIGURE 7 | Examples of partitions with minimum overall Manhattan distance with the value of the minimum overall Manhattan distance.

of the minimum Manhattan distance-shaped partitions on two-dimensional grid topologies. The basic algorithm can also be adapted to multidimensional grid topologies, with reasonable performance of the generated partition shapes.

With a large number of nodes, the shape that results in the minimum overall Manhattan distance is a circle, as shown in Figure 7. Therefore, the Manhattan Median approach focuses on generating circular partition shapes. To achieve this, the proposed algorithm follows a greedy approach. For a given set of nodes, all nodes that are within a certain area resembling the enlarged circle are determined and assessed. The nodes, which increase the overall Manhattan distance of all nodes in a minimal way, are added to the partition.

It must be emphasized that the originally proposed algorithm is based on the assumption that the partition is formed on an empty cluster system. This aspect and the requirement to support advance reservations make it essential to adapt the algorithm [24]. Like the Hilbert curve-based approach, the Manhattan Median approach begins with a check to see if enough resources are available in the interval between the earliest start time and the scheduled end time based on the runtime prediction before the partition is formed. If this initial check is successful, an initial start time is set and used to check all nodes.

The shaping of the partition itself begins—as exemplified in Figure 8—with the search for a first compute node that is available over the time interval and can thus be used for the job by

```
0  find valid time interval with enough resources
1  if valid time interval is found
2    $node_{start} := node_{first\_node\_of\_cluster}$
3    while $t_{start} + t_{runtime} \leq t_{deadline}$
4      while $nodes_{map} = \emptyset$
5        if $\forall t \in [t_{start}, t_{start} + t_{runtime}] : free_{n(t)}$
6          $nodes_{map} := node_{start}$
7          $nodes_{sum\_map} := 1$
8        else
9          $node_{start} := node_{next\_node\_of\_cluster}$
10       while $nodes_{sum\_map} < nodes_{sum\_req} \wedge nodes_{checked} = \emptyset$
11         select from the neighboring nodes of $nodes_{map}$ the nodes $nodes_{selected}$ that increase the
             overall Manhattan distance the least
12         $\forall n \in nodes_{selected}$ :
13           if $\forall t \in [t_{start}, t_{start} + t_{runtime}] : free_{n(t)}$
14             save $n$ in $nodes_{map}$
15             $nodes_{sum\_map}$++
16             $nodes_{checked} := \emptyset$
17           else
18             save $n$ in $nodes_{checked}$
19       if number of $nodes_{map} = nodes_{job}$
20         return $(t_{start}, nodes_{map})$
21       else
22         $nodes_{map} := \emptyset$
23     get next valid time interval with $t_{start}$
24 reject request
```

FIGURE 8 | Overview of the Manhattan Median approach.

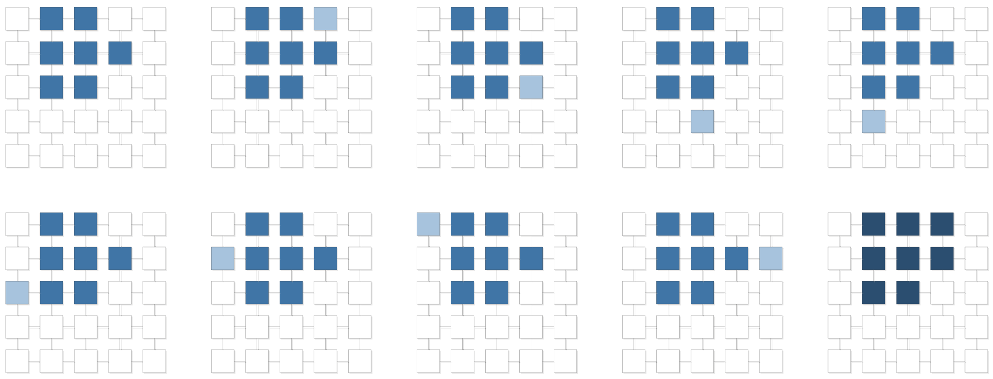


FIGURE 9 | Search for partition with minimum overall Manhattan distance.

checking one node after another following the specified order within the machine (lines 4–9). Starting from this first node, all neighboring nodes are examined to see if they are available in the same time interval from the start time to the scheduled end time (lines 10–18). All nodes that meet this condition are considered, and the overall Manhattan distance is calculated for each of these new nodes. The calculation of the overall Manhattan distance is performed for a node with respect to all nodes that are part of the partition—in the first step, this is only the first node. The node that would minimally increase the overall Manhattan distance of the partition is considered to be part of the partition. This step is followed by the search and examination of all neighbor nodes for all nodes of the partition (see Figure 9). This procedure is optimized in the way that only the nodes at the edge of the partition are considered as new neighbor nodes to be investigated.

These steps are repeated until the requested number of nodes has been determined and inserted into the partition or no more free neighboring nodes can be found. If the request cannot be satisfied with the available nodes by starting with the first node, the partition is released, the next nodes are checked to see if any of them can be used as a new first node, and the procedure is started with this new first node of the partition.

In case no partition with a sufficient number of nodes can be found, the start time is increased in the expectation that some of the examined nodes will be available later. Based on this new start time, the search for free nodes for the current request is started again. The start time is increased in increments determined by the change in the occupancy state of all other jobs. This is reasonable because additional nodes become available only after other jobs are finished, freeing up nodes. In addition, the start time is only increased if the time remaining until end time (deadline) is at least as long as the requested runtime.

Static resource allocation is designed to provide appropriate shapes of partitions at the time of submission but increases the chance of over-provisioning for programs with dynamic runtime behavior. This is because the approaches book the resources of the entire time interval the job is executed, regardless of the utilization of the node during the whole runtime. However, when the job is finished, or when the specified end time of the job is reached, the computing resources are released and marked as free to be assigned to the next job or set of jobs.

4 | Dynamic Allocation of HPC Resources

For parallel programs implementing a more static runtime behavior, such as Monte Carlo-like applications or BSP programs (types 0 and 1), the static allocation as provided by the current RMS, like the SLURM workload manager, should be well suited to support these types of runtime behavior even in a deadline-oriented workflow scenario. However, for the applications with dynamic runtime behavior (types 2 and 3), it can be assumed that static allocation strategies are not appropriate because the number of compute resources changes during runtime [25, 26]. For parallel programs with dynamic runtime behavior, the provision of a fixed number of nodes could reduce the utilization of the entire HPC system, as not all nodes are used by the program over the entire runtime. The nodes that remain free during parts of the runtime of the program, as only a reduced number of processes exist, could then be made available to other jobs. The consideration of the runtime behavior by the RMS should therefore increase the overall utilization of the HPC system and ensure that the deadlines of the jobs can be met with a high probability. Therefore, a more dynamic way of resource assignment should be beneficial. This means the resources required by the running job are assigned when the resource is needed and released immediately after use.

4.1 | Contiguous Leak Approach

In order to implement such dynamic allocation of HPC resources and support decentralized and distributed resource management, the leak approach was introduced [9, 27] and examined in detail [28]. In this work, we have adapted it to work with deadline-oriented RMSs.

In the leak approach, the parallel program resembles a liquid that is dropped into a partially occupied container (e.g., a parallel machine with partially allocated nodes). Due to its properties, the job occupies a free part of the container or displaces another job until an equilibrium is reached. Since parallel programs—especially those with dynamic runtime behavior—change their properties, such as the number of processes running in parallel, the equilibrium and thus the number of resources used by the different jobs can change over time. This liquid-like expansion of the number of nodes used by the parallel program ideally fits into a phase of reduced node utilization of other jobs. Thus, the jobs should complement each other in terms of resource utilization.

In contrast to the previously discussed allocation strategies, the RMS in the leak approach accepts all job requests and handles the resource requirements of the program at runtime. Thus, no advance reservation is performed at the submission time. In this way, the job is started at the earliest possible time. The algorithm in Figure 10 exemplifies this strategy. To start the first process of the parallel program, an empty compute node is searched for (lines 1–5). The distribution of the different jobs over the entire HPC system is supported by the use of different nodes from which the search is started. If the entry node is occupied and therefore unable to accommodate the new process, a next node is

```
0   in case of advance reservation find valid time interval with enough resources
1   $node_{start} := node_{one\_of\_start\_nodes\_of\_cluster}$
2   while $node_{start} \wedge node_{start} \neq node_{last\_next}$ is occupied
3     $node_{start} := node_{next\_node}$
4   if $node_{start}$ is free
5     save $node_{start}$ in $nodes_{map}$
6   else
7     reject request
8   select in $nodes_{selected}$ all neighboring nodes of $nodes_{start}$
9     while $nodes_{sum\_map} < nodes_{sum\_req} \wedge nodes_{selected} \neq \emptyset$
10      if $\forall n \in nodes_{selected} : free_n$
11        save $n$ in $nodes_{map}$
12        $nodes_{sum\_map}$++
13      select in $nodes_{selected}$ all unchecked neighboring nodes of $nodes_{map}$
14      if $nodes_{selected} = \emptyset$in case of non-contiguous leak search for next free node to put into $nodes_{selected}$
15  return $nodes_{map}$
```

FIGURE 10 | Overview of the Leak approaches.

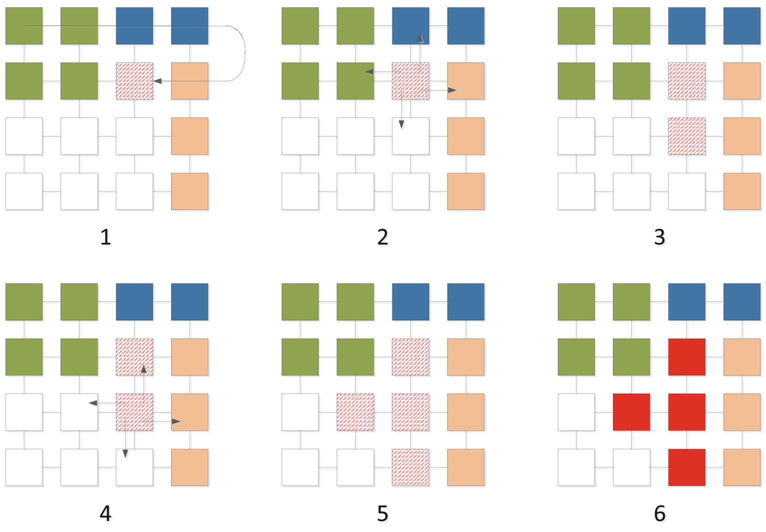


FIGURE 11 | Search for first nodes using the Leak approach.

examined (lines 8–12). The search for a first node for the new job is performed sequentially for all nodes of the same row for grid or torus topologies. Thus, the search sequence for a first node for the starting job resembles a snake. If no node is available for the job to be started, the job is rejected (line 7).

Figure 11 exemplifies the search for the first nodes for a new job. As the nodes in the first row are occupied by the green and blue jobs (see Tile 1), the search for a first free node in the second row is continued in the opposite direction. In this row, a node is occupied by the light red job, but the next node is free and marked as the first node for the new job. As the new job requires four nodes, further free nodes have to be found. This is done by examining the neighboring nodes, starting with the first marked node (see Tile 2). Since all of the node's direct neighbors except for the node in row 3 are used by other jobs, this free node is marked for the new job (Tile 3) and taken as the start for the next search step (Tile 4). In this search step, two free nodes can be found and marked (Tile 5). Finally, it is confirmed that all nodes are assigned to the new job, as the required number of nodes has been reached (Tile 6).

Some additional processes can be created while the parallel program is running. In connection with the creation of an additional process, there is a search for a node on which this process is to be started. The search for the node is performed by the RMS, starting from the node to which the creating process is assigned. First, all of the neighboring nodes are checked. If no node of these adjacent nodes are available, all other nodes used by the parallel program are used to check the neighbors of these nodes. In this way, a compact—dynamic—partition is created for the job. In contrast to the static partition approaches, the leak approach introduces requests for additional nodes—except for the first node—at runtime and not at the start time of the job. Therefore, checking for additional nodes cannot be used to reject a job at submit time. If no additional node can be found near the partition with the contiguous leak version, the new process is started on the node of the creating process (parent process).

In Figure 12, the example from Figure 11 is continued (Tile 1), and one node is released because the process of the red job assigned to this node has been completed (see Tile 2). This node is then used by a new job (yellow) and can no longer be used by the red job if a new process is created (Tile 3). However, as a new process is now created by the red job and the previously used node is occupied, a search is performed from all nodes used by the job (Tile 4). Two free nodes are found, and one of them is used for the new process of the red job (Tile 5).

4.2 | Noncontiguous Leak Approach

Due to the dynamic runtime behavior of the various jobs running on the HPC system, in some cases no free resource can be found for a new process in the neighborhood of the partition of a job. Starting the new process on the same node as the creating parent process contradicts the intention of the runtime behavior and thus also the intention of the programmer, since the new process is created to use an additional compute resource. The start on the same node and the preemption of the parent process would in many cases slow down the execution of both of the processes and thus reduce the probability that the job can be completed before the deadline. It therefore makes sense to extend the search space and also considers nodes that are not direct neighbors of the nodes of the partition. However, this can increase the communication costs.

Nevertheless, to overcome the restriction of the continuous leak version, a noncontiguous version of the leak approach was introduced. The noncontiguous version of the leak approach works in the same way as the contiguous version: search for the start node, search for additional nodes in the neighborhood of the node to which the parent process is assigned, or the nodes of the partition. Only in case no free node from the set of neighboring nodes is available to satisfy the current request, the search for other nodes is performed like the search for a start node as described before (see Figure 11), resulting in several disjoint subpartitions of contiguous nodes. For further processes, these additional nodes are then treated as a node of the partition, so that all regions used by

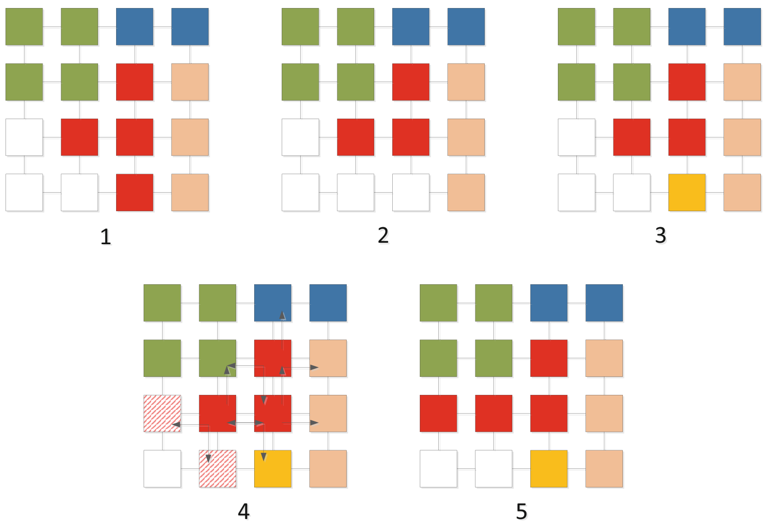


FIGURE 12 | In the Leak approaches, nodes can be released, and additional nodes can be requested at job runtime (breathing partition). The search for additional nodes starts by examining the neighboring nodes (shaded red) of the nodes in the partition (solid red).

the job can leak in the aforementioned way. As with the contiguous version, the nodes are released once the process is completed and can then be used for other requests of the same or other jobs.

In the example shown in Figure 13, after a first partition has been searched for and used (Tiles 1 and 2), additional processes are created, and free nodes are searched for. Since there are no free nodes available in the neighborhood of the used nodes, the noncontiguous leak performs the snake-like search and can find a free node in the second row (Tile 3). From this node, others in the neighborhood are searched for and found (Tiles 4–6) and used by the red job (Tile 7). During the runtime of the job, a node is released (Tile 8) and used by another job (Tile 9). The creation of a further process then leads to an additional search, shifting the set of nodes further away from the original node (see Tiles 10–12).

5 | Evaluation

Based on the observations presented earlier and with respect to the different types of parallel programs, the examples of static and dynamic resource allocation were examined. In order to evaluate the different approaches of RMSs in terms of their efficacy to provide the required QoS (respecting jobs deadlines) and ensure the completion of the jobs for a given workload, we used the ApplOXSim simulation environment [24, 29] focusing on the impact of data structures and static allocation strategies. This simulator is capable of simulating both the RMS tasks as well as the execution of the parallel programs. Thus, the discrete event simulation controls the entire machine as well as the individual computing resources and interconnection network. Events are introduced to notify the arrival of an HPC job, such as the start and the end of a job, and the termination of the job in case the deadline has been reached (canceled job). In addition, events are created and handled for each part of the parallel program that leads to a resource utilization, such as the start of a computation (computation task) or the transmission of a message (communication task) as well as the start of a new process (fork) or the joining of some processes when the child process finishes its work (join). Therefore, in addition to the two static allocation strategies for resource management for HPC systems, the two variants of the leak approach (contiguous and noncontiguous) were implemented in this simulation environment to evaluate their performance. For the leak approach, the search for a new CPU core is performed when a process invokes a fork system call and the CPU core is released when a join call is performed by the process.

Furthermore, the simulator is able to support different configurations of HPC machines. The configurations used for the evaluation presented in this article are based on the performance values of the supercomputer HLRN-II, which was in operation at the Zuse Institute Berlin (ZIB) comprised of 128 compute nodes connected by a grid topology (totalising 512 CPU cores). To provide a realistic load for the RMS to handle, the workload generator by Feitelson [30] was used, and the resulting traces were extended to include the type (model) of the runtime behavior of the parallel programs as well as the earliest start time and deadline for the job. Since deadlines are defined by the users and the higher-level workflow in which the job is embedded, it is reasonable to assume that the deadline leaves some leeway for the management system. This slack time is often related to the importance and the size of the job, as the user would want to reduce the risk of canceling the job if the results are important. The workload generator can be configured to provide different load levels in the form of arrival rates. It is based on Feitelson's workload generator, which generates new traces based on an archive of real traces executed on parallel supercomputers, clusters, and grids and using a normal distribution [30]. In addition to the recommended and common load, which is referred to as normal load and is determined with an interarrival factor of 1500, two other load situations were examined. The very heavy load (interarrival factor of 1) is determined by the fact that on average, a new job request is submitted to the RMS every second. Between the normal and very heavy load, a heavy load situation was investigated with an interarrival factor of 150. The traces provide 1000 job requests over a simulation time of about 65 days for the heavy and very heavy load configurations and about 74 days for the normal load traces.

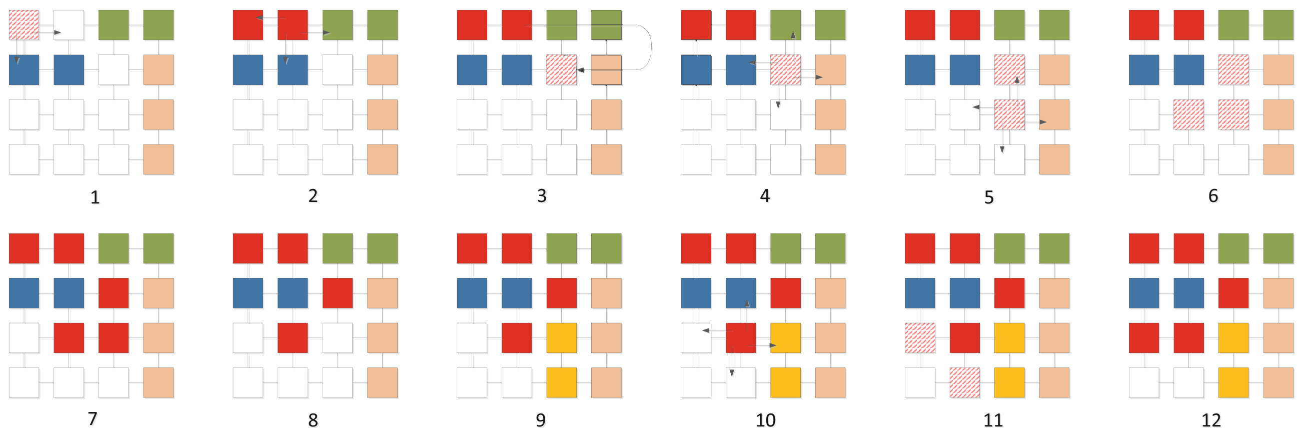


FIGURE 13 | Search for nodes using the noncontiguous leak approach. After the initial partition is created in the first row (red nodes), nodes are released in the second row during execution, and new nodes are assigned when new processes are created.

The running parallel programs were derived from the different runtime behavior models and adapted to the specifications as given in the job description of the individual trace file. However, the jobs are adapted to be able to run the program using a compact partition of an empty HPC system at the given runtime, and thus, they should be able to finish the execution before the deadline in case no other interferences occur.

In addition to the simulation of traces with jobs following the same runtime behavior type (models 0–3), a mixture of the types was used according to a uniform distribution of the behavior types (model uniform) to reproduce a more realistic scenario in a shared production machine (with different users using different models). For each configuration—application type, communication behavior, and load—320 different traces were generated and used for simulation in order to obtain significant results for the performance of the different approaches. In total, about 38,000 different simulation runs were performed to obtain the results resented in this article [31]. Since the simulator tool developed for this work is open source, it can from now on be used by related work to compare their scheduling and mapping results to our results.

5.1 | Request Acceptance

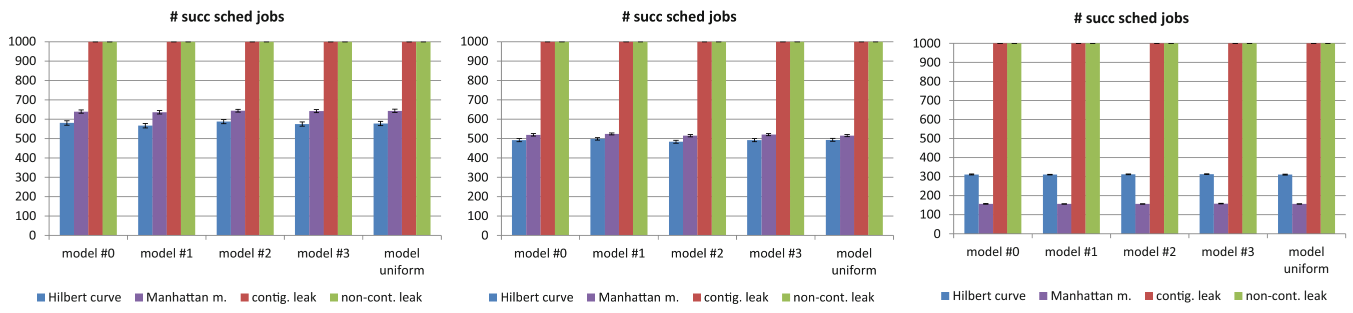
As might be expected, the approaches perform differently on the performance metrics. First, the impact of checking the number of resources when the jobs are submitted can be clearly seen in the acceptance rate (see Figure 14). While the static allocation strategies perform such a check at submit time for the total number of computing resources, with the leak approach, the check is performed only for one node. Thus, nearly all of the jobs are accepted by the RMS, which implements one of the versions of the leak approach.

If only the first step of the resource allocation is considered, that is, the number of jobs that could be accepted after checking the number of nodes (line 0 in the static procedures), a number of 510, 400 and 260 jobs are accepted for normal, heavy, and very heavy load. In comparison, taking the mapping into account by using the Hilbert curve-based approach (SLURM), the average numbers of successfully scheduled jobs are increased up to more than 120% for the heavy load situation. This is due to the mapping decision since a partition has to be found for the job at submit time. Jobs are rejected if such a partition is not available for the time, the job is to be executed. The impact of the mapping decision is even more obvious with a look at the Manhattan median approach, where the average number of accepted jobs is slightly higher than for the Hilbert curve-based approach at normal load, about half of the average number at very heavy load. With increased load, the chance to find a sufficient number of free nodes reduces, and the chance to find contiguous free nodes is even more reduced due to external fragmentation (external fragmentation may occur after the release of previously allocated partitions, when free resources are scattered at separate locations in small discontinuous portions being unusable as a whole for contiguous policies). The effect of the external fragmentation is reduced by the Hilbert curve-based approach, which implements a first fit search for free nodes. The special shapes of the partitions provided by the Manhattan media approach lead to even greater external fragmentation. However, this has no significant impact on the leak approaches, which still accept almost all job requests due to their dynamic allocation.

Since the partition provided by the static allocation strategy is assigned to the job over the entire runtime and the check for free resources is independent of the runtime behavior, no significant difference can be seen with regard to the runtime behavior types of the jobs (see Figure 14). Same holds true for the communication pattern used and also for the dynamic allocation strategies. Even though the number of accepted jobs is only part of the performance of the approaches, it can serve as a lower measure for further consideration. The more important is how many jobs are completed before the given deadline or canceled because they cannot meet it.

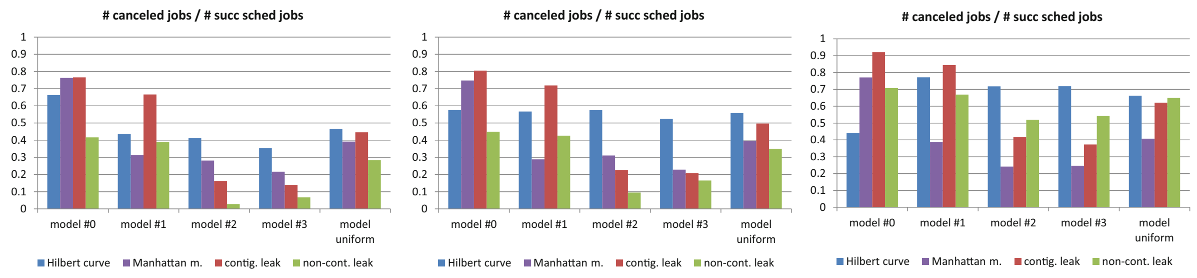
5.2 | Job Canceling Rate

The comparison of the different approaches implementing static and/or dynamic resource allocation in terms of the number of successfully scheduled jobs at submit time suggests that not all jobs can be completed successfully. Especially for the leak



(a) Normal load (b) Heavy load (c) Very heavy load

FIGURE 14 | The numbers of successfully scheduled jobs under normal load show the difference between static (blue and purple) and dynamic (red and green) assignments with respect to the first checks for free nodes.



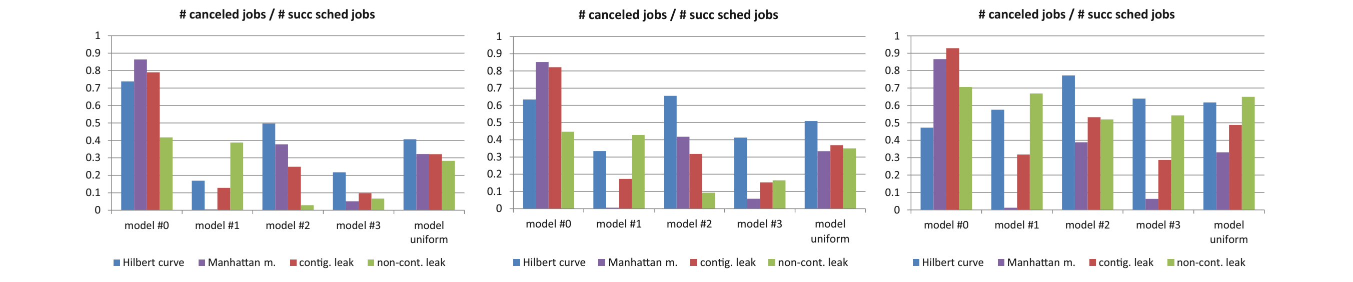
(a) Normal load (b) Heavy load (c) Very heavy load

FIGURE 15 | The canceling rate depends on the combination of load, behavior model, and approach, but in almost all configurations, the noncontiguous leak approach (green) outperforms the Hilbert curve-based approach (blue) and the Manhattan median approach (purple), which performs better than the other static allocation strategy (here for asynchronous communication).

approach, which accepts almost all jobs, it can be assumed that the overbooking at heavy and very heavy load will lead to a significant number of canceled jobs. The results for the leak approach confirm this assumption for both the contiguous and the noncontiguous version of the leak approach (see Figure 15a) even under normal load. However, a significant number of canceled jobs out of the number of accepted jobs is also observed for the Hilbert curve-based approach used by SLURM and the Manhattan median approach. Since the static Manhattan median approach and to a certain extent also the Hilbert curve-based approach implemented by SLURM perform better than the dynamic allocation strategies for the jobs running parallel programs with rather static runtime behavior type (model 0 and 1), the picture is reversed for the programs with dynamic runtime behavior. The Manhattan median approach reduces the communication overhead and provides compute resources of the entire runtime, but this is not sufficient to ensure the finishing of the job before deadline even for the jobs with static runtime behavior. Nevertheless, the approach performs best of all of the four approaches with respect to the jobs with a static number of communicating processes (BSP-like jobs—runtime behavior type 1). The dynamic resource provisioning of the leak approach supports the dynamic runtime behavior, as it can be seen in the results for the runtime behavior types 2 and 3. Especially with the noncontiguous leak approach, the share of canceled jobs is much smaller than for the static allocation strategies, and the version performs best for the mixture of runtime behavior types (model uniform) under normal load.

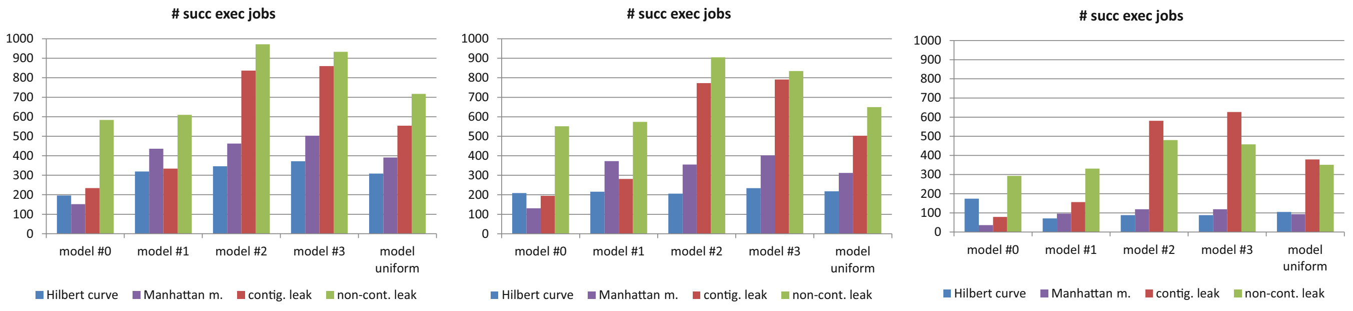
As expected, the share of canceled jobs increases with load in the dynamic allocation strategies (see Figure 15c). This is due to the overload situation in the network compared to the static allocation strategies, and it becomes even worse if no free computing resource is found at the time a new process is created for the dynamic strategies. The last is even more relevant when the number of nodes to be examined is reduced since only nodes in the neighborhood of the existing partition are considered, as it is with the contiguous leak approach. Nevertheless, the results for the very heavy load show that the increased communication overhead associated with a more scattered assignment, due to the need to use more distant nodes in the case of noncontiguous leak, and a higher load on the network links themselves results in worse performance for the noncontiguous leak compared to the contiguous leak for the job with dynamic runtime behavior. In this load scenario, the reduction of the communication costs that comes with the Manhattan median approach pays off, so that this static approach performs best for almost all configurations. Only the assignment of the Hilbert curve approach supports the runtime behavior of jobs with a static number of noncommunicating processes (Monte-Carlo-like jobs—runtime behavior type 0) better than the Manhattan median approach.

While for the dynamic assignment approaches, the share of canceled jobs increases with the load and, for the static Manhattan median strategy, the share of canceled jobs changes only slightly with the load. This is different from the Hilbert curve-based



(a) Normal load (b) Heavy load (c) Very heavy load

FIGURE 16 | The canceling rate for jobs implementing synchronous communication pattern is similar, but here the compact shape of the partitions is more conducive to the Manhattan median approach, especially for runtime behavior types 1 and 3.



(a) Normal load (b) Heavy load (c) Very heavy load

FIGURE 17 | Due to the high acceptance rate and the canceling rate, the leak approaches (red and green) perform better than the Hilbert curve-based approach (blue) and the Manhattan median approach (purple) in terms of successfully executed jobs, especially for the jobs with dynamic runtime behavior (here for asynchronous communication).

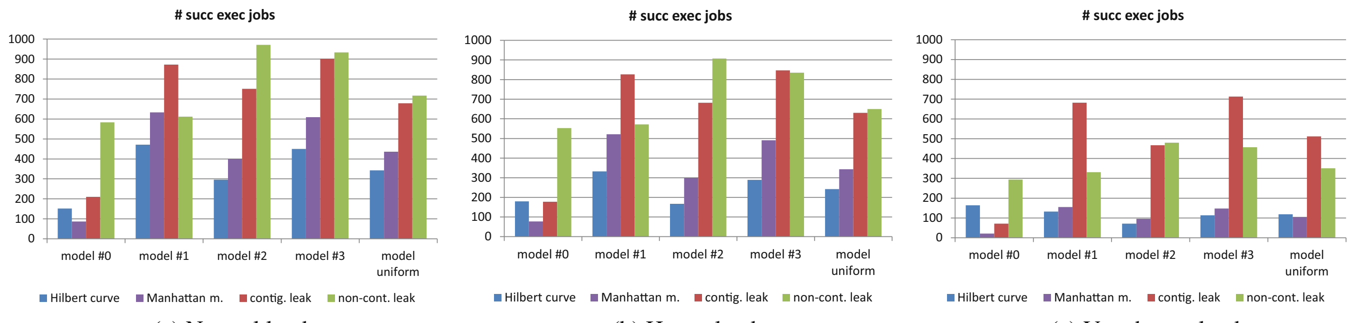
approach with respect to jobs with a static number of noncommunicating processes (Monte-Carlo-like jobs—runtime behavior type 0). Since communication is concentrated at the start and end of the runtime with this runtime behavior type, communication overhead is not as important to the runtime as it is for other types of runtime behavior. Moreover, the higher load—which comes with a reduction in interarrival time—in case of static assignments reduces the advantage of smaller jobs. If all job requests arrive at the same time and the decision to schedule some of the jobs is based on the number of available computing resources, all jobs have an equal chance of being selected, and the preference for smaller jobs that fit into some holes in an already existing schedule is negligible. Since the larger jobs are associated with higher slack time, these jobs have a higher probability to get finished before the specified deadline. However, due to the better support of the runtime behavior of the jobs, especially those with dynamic runtime behavior, the dynamic allocation in most scenarios performs better than the static allocation strategies based on the Hilbert curve or Manhattan median approach when it comes to the share of canceled jobs.

This also holds for other communication patterns than the widely used asynchronous send in combination with a synchronous receive operation. The results for jobs performing synchronous send operations only are depicted in Figure 16. The difference in communication patterns is visible, especially with the jobs with a static number of communicating processes (BSP-like jobs—runtime behavior type 1). Because of the exchange of intermediate results between all of the processes, the allocation of a compact partition has a great influence on the number of canceled jobs.

5.3 | Overall Performance

The unexpected high number of canceled jobs in combination with the rejection of jobs due to the allocation of static partitions provided by the Hilbert curve-based SLURM approach as well as the Manhattan median approach leads to reduced performance in terms of the number of successfully executed jobs. This holds true for the jobs with dynamic runtime behavior, where only half the number of jobs can be completed before the deadline compared to the leak approaches under normal load (see Figure 17a). The results are even more evident for the other load situations (see Figure 17b,c). However, also for the jobs, implementing a static runtime behavior using the static partition strategy is beneficial only for Monte Carlo-like applications at very heavy load compared to the contiguous leak approach, while the noncontiguous leak performs better even for this type of applications. Thus, the characteristics of the static partition strategies, which come with a guarantee to the job that there will always be enough computing resources available to start the processes, and the shape that reduces communication overhead do not compensate for the specific requirements associated with more sophisticated new programming approaches implementing a dynamic runtime behavior.

The fundamental results observed for the programs with asynchronous communication generally hold for the jobs with



(a) Normal load (b) Heavy load (c) Very heavy load

FIGURE 18 | The comparison between the strategies for synchronous communication is similar to that for asynchronous communication, and the noncontiguous approach in particular delivers better performance in almost all configurations than the Hilbert curve-based and the Manhattan median strategy.

synchronous communication patterns, although there are some increases for certain configurations. For the SLURM RMS and the contiguous leak approach, the synchronization of the processes through synchronous communication leads to a decrease in the number of canceled jobs and thus to an increase in the number of successfully executed programs compared to the job with asynchronous communication (see Figure 18). This can be seen particularly well when looking at the results for the jobs implementing a BSP-like runtime behavior (model 1) in combination with the contiguous leak approach. In this case, the occasional mapping of the child process to the node of the parent process, since no free node is available in the neighborhood of the partition, reduces the communication overhead and can at least partially compensate for the overload situation on the respective node.

6 | Related Work

In recent years, there are several contributions to the field of HPC management [1, 15, 32, 33]. First, there is great interest in the research and development of novel applications implementing dynamics in different ways, such as load balancing in particle simulations [34] or dynamic runtime behavior based on different computational phases of the parallel program [35], mesh refinements [20, 36–38], and frameworks to support programming such applications with dynamic runtime behavior to overcome the limitations of static applications, such as the need for internal load balancing and running multiple versions to deal with different data sets [19].

Secondly, the need to support an increasing variety of applications also leads to the development of new features for the RMS. Therefore, research in the field of RMS is also driven by the emergence of complex parallel applications with dynamic runtime behavior. An example of this is the extension of the SLURM workload manager [14] to support dynamic runtime behavior by providing malleable job allocation [39]. This also includes current work to support dynamic resource management, for example, by implementing the PMIx standard for the SLURM workload manager [40].

Furthermore, the use of ML techniques to improve the reliability of runtime predictions required to optimize the schedule [41, 42] is a promising approach that could also be considered in combination to dynamic allocation strategies. Other approaches focus on evaluations of runtime behavior at runtime [43] to optimize the schedule. While this classification can support the resource allocation, as our results with the different types of runtime behavior clearly show, additional online scheduling techniques, as introduced with the leak approach, are needed to meet the requirements of individual jobs. The utilization of the HPC system can further be increased by introducing coallocation [16]. Our investigations of the leak approach support this finding and additionally provide a mapping approach that supports jobs with dynamic runtime behavior. Almost all of these approaches and research activities focus on optimizing the utilization of HPC systems in combination with increasing the reliability of job execution.

In addition, the requirements that the job must be completed before a certain deadline have also been taken into account in recent years. This can be seen as a further increase in reliable job execution or a higher level of service quality. However, meeting deadlines for jobs is a challenging task, especially for batch job-based scheduling. Nevertheless, different approaches are presented for these systems. One approach [8] aims to optimize the schedule considering the deadlines for the jobs given by the users, while another approach [6] additionally considers further challenges such as heterogeneous systems and different resource requirements or constrains. Since this article focuses on the scheduling part of the allocation problem, there are other approaches [44] that deal with the mapping of the individual processes or tasks. Therefore, a more sophisticated model, a directed acyclic graph (DAG), is used to describe the runtime behavior of the parallel program running as a job.

The combination of all these research works and our results clearly shows that all of these aspects and requirements have to be considered in order to develop a reliable solution for the next generation of HPC RMSs. However, current research efforts, as described here, have not yet considered all aspects necessary for ensuring deadlines in the field of HPC, although we have listed the most similar ones here. Nevertheless, to the best of our knowledge, there is no other work that has explored and analyzed the impact of a dynamic RMS strategy in the context of deadline-oriented execution scenarios—and its impact on scheduling, mapping, application performance, and throughput—with respect to the different types of runtime behavior.

7 | Conclusion and Outlook

In this article, we evaluate how well static resource management policies cope with deadline-constrained HPC jobs and explore two variations of a dynamic policy in such scenarios. Deadline enforcement is crucial in environments where a higher level of QoS is implemented through the use of SLAs in order to support advanced workflows.

The Hilbert curve-based approach used by the SLURM workload manager, the widely used RMS, and the more research-based Manhattan median approach are prominent examples for static allocation strategies for HPC systems that focus on compact partitions. Since the compactness of partitions is considered the basis for reducing the runtime of a parallel application, these approaches were used as a reference for evaluating the leak approach, which implements dynamic resource management. Therefore, the static allocation strategy should provide sufficient computing resources for the running job and reduce the communication overhead by providing compact partitions. This should at least correspond to the static runtime behavior of the parallel program types according to models 0 and 1.

However, our results clearly show the impact of the runtime behavior of the parallel applications running on these HPC systems on the results, as previously indicated [6]. The static policies are not able to meet the requirements of a modern deadline-oriented RMS scenario, especially in terms of resource utilization and the number of jobs that are successfully executed as agreed in the SLA. It is important to emphasize that the optimization of the partitions performed with the Manhattan median approach reduces the number of canceled jobs only at the cost of rejecting a large part of the job requests, and the Hilbert curve-based approach performs best in terms of the share of canceled jobs only for the Monte Carlo-like jobs at overload situation. Rather, the results for the dynamic assignment approaches—the contiguous approach and even more so the noncontiguous leak approach—show the need to support dynamic runtime behavior of the jobs in the form of providing additional computing resources at the time a new process is created. The neighborhood relationship between the nodes of the partition is important in reducing the communication overhead associated with the message passing of the parallel applications but has less impact than the timely provisioning of additional compute power. Since this effect may be of minor importance in queueing-based RMSs where the runtime of the job can be extended without major impact (except for optimizations implementing backfilling [45]) and the reduction in utilization associated with static partitions can be accepted, any aspect that threatens the deadline has to be omitted in the area of SLA-based RMS.

However, the results for the dynamic allocation—the leak approaches—also show that the resource requirements of parallel programs with the same runtime behavior type can match in such a way that one program can use the currently freed resources that were previously used by another program. This effect can greatly increase the number of jobs executed on the HPC system. However, the mixture of parallel applications with different runtime behavior types can reduce this effect.

Since the results presented in this article were obtained through an extensive number of simulations, some aspects of real-world systems are not considered or can only be examined in productive systems, such as estimations of runtime by the users and specified deadlines. Nevertheless, a rough estimate of the technical transfer and its effects can be given. It would be reasonable to assume that the effort required for dynamic resource management is higher than for the static allocation strategies, especially for jobs with dynamic runtime behavior. The impact of the resource management activities, such as searching for additional nodes, maintaining the current state of resource allocation and monitoring the overall resource utilization, can initially be reduced due to the decentralized nature of the leak approach. This also promotes the parallelization of the tasks, as the requests are sent to more than one node. Thus, the decentralized dynamic approach should be able to scale much better than centralized, static allocation strategies. Especially with respect to the partial shift of the resource allocation overhead from the start time to the runtime of the job. In addition, the caching of the states of resource allocation of neighboring nodes can also be used to reduce the resource management overhead associated with dynamic policies. Since the spawning and initialization of new processes in MPI environments come with a certain amount of overhead regardless of the resource management approach, resource management can easily be integrated into this part [40]. However, the dynamic strategy shows its strengths when using dynamic applications. For parallel programs with static runtime behavior, resource management at the start of the job is similar to the resource allocation of the static allocation strategies. For the programs with dynamic runtime behavior, the dynamic resource allocation significantly increases the utilization of the system, as the transition is made from internal fragmentation to externally available resources, which only count as external fragmentation if no further jobs are accepted or other jobs do not require resources.

Overall, our preliminary results clearly show that RMS should adopt dynamic policies, but more detailed information about the—future—runtime behavior of the parallel program should be incorporated in the scheduling and mapping decisions performed by the RMS. This holds for the information about creating and joining processes as well as for the communication patterns used by the parallel program. Future work will address this feature and pursue a resource management approach that is capable of providing a reliable service for the execution of parallel programs with deadlines by taking into account the runtime behavior and resource requirements of the HPC jobs in order to support SLAs.

Data Availability Statement

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Endnotes

[1] https://www.nhr-verein.de/research.

References

1. D. Reed, D. Gannon, and J. Dongarra, "HPC Forecast: Cloudy and Uncertain," *Communications of the ACM* 66, no. 2 (2023): 82–90.

2. M. Bux and U. Leser, "Parallelization in Scientific Workflow Management Systems," Arxiv Preprint Arxiv:1303.7195. 2013.

3. J. Schneider, "Grid Workflow Scheduling Based on Incomplete Information," PhD Thesis, Technische Universität Berlin. 2010.

4. R. F. Da Silva, H. Casanova, K. Chard, et al., "A Community Roadmap for Scientific Workflows Research and Development," *2021 IEEE Workshop on Workflows in Support of Large-Scale Science* (*WORKS*) (2021), 81–90.

5. J. D. Mattingly, *Aircraft Engine Design* (Reston, VA: AIAA, 2002).

6. Y. Fan, "Job Scheduling in High Performance Computing," *Horizons in Computer Science Research* 18 (2021), 1–15.

7. Y. Fan, Z. Lan, P. Rich, W. Allcock, and M. E. Papka, "Hybrid Workload Scheduling on HPC Systems," *IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*) (2022), 470–480.

8. T. H. Le Hai, K. P. Trung, and N. Thoai, "A Working Time Deadline-Based Backfilling Scheduling Solution," *International Conference on Advanced Computing and Applications* (*ACOMP*) (2020), 63–70.

9. H. U. Heiss, *Prozessorzuteilung in Parallelrechnern* (Mannheim, Germany: BI-Wiss.-Verlag, 1994).

10. T. N. Minh and L. Wolters, "Using Historical Data to Predict Application Runtimes on Backfilling Parallel Systems," In *IEEE Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing* (2010), 246–252.

11. M. Soysal, M. Berghoff, and A. Streit, "Analysis of Job Metadata for Enhanced Wall Time Prediction," in *Job Scheduling Strategies for Parallel Processing* (Heidelberg, Germany: Springer, 2018), 1–14.

12. E. Strohmaier, J. Dongarra, H. Simon, M. Meuer, and H. Meuer, "TOP500 List," accessed February 9, 2024, https://www.top500.org/.

13. D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling Using Runtime Predictions Rather Than User Estimates," *School of Computer Science and Engineering, Hebrew University of Jerusalem, Technical Report TR* (2005).

14. A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple Linux Utility for Resource Management," 9th International Workshop on Job Scheduling Strategies for Parallel Processing (2003), 44–60.

15. B. Linnert, C. D. Rose, and H. U. Heiss, "Impact of a Dynamic Allocation Policy for Resource and Job Management Systems in Deadline-Oriented Scenarios," *WSCAD* (2023).

16. A. Frank, "Reducing Resource Waste in HPC Through Co-Allocation, Custom Checkpoints, and Lower False Failure Prediction Rates," *PhD thesis*, *Johannes Gutenberg-Universität Mainz* (2022).

17. L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM* 33, no. 8 (1990): 103–111.

18. J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé, "Improving the Integration of Task Nesting and Dependencies in OpenMP," *2017 IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*) (2017), 809–818.

19. J. Aguilar Mena, O. Shaaban, V. Beltran, P. Carpenter, E. Ayguade, and J. Labarta Mancho, "OmpSs-2@ Cluster: Distributed Memory Execution of Nested OpenMP-Style Tasks," Euro-Par 2022: Parallel Processing: 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22–26, 2022, Proceedings (2022), 319–334.

20. N. Nangia, N. A. Patankar, and A. P. S. Bhalla, "A DLM Immersed Boundary Method Based Wave-Structure Interaction Solver for High Density Ratio Multiphase Flows," *Journal of Computational Physics* 398 (2019): 108804.

21. W. Smith, I. Foster, and V. Taylor, "Scheduling With Advanced Reservations," *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS* (2000), 127–132.

22. R. P. Becker, "Entwurf und Implementierung Eines Plugins Für SLURM Zum Planungsbasierten Scheduling," Bachelor's Thesis. Freie Universität Berlin (2021).

23. M. A. Bender, D. P. Bunde, E. D. Demaine, et al., "Communication-Aware Processor Allocation for Supercomputers: Finding Point Sets of Small Average Distance," *Algorithmica* 50, no. 2 (2008): 279–298.

24. B. Linnert, J. Schneider, and L. O. Burchard, "Mapping Algorithms Optimizing the Overall Manhattan Distance for Pre-Occupied Cluster Computers in SLA-Based Grid Environments," *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (*CCGrid*) (2014), 132–140.

25. D. Álvarez, K. Sala, and V. Beltran, "nOS-V: Co-Executing HPC Applications Using System-Wide Task Scheduling," *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (New York, NY: IEEE, 2022), 312–324.

26. J. Li, G. Michelogiannakis, B. Cook, D. Cooray, and Y. Chen, "Analyzing Resource Utilization in an HPC System: A Case Study of NERSC Perlmutter," arXiv Preprint arXiv:2301.05145 (2023).

27. C. A. De Rose, "Verteilte Prozessorverwaltung in Multirechnersystemen," PhD thesis. Universität Karlsruhe (Technische Hochschule) (1998).

28. C. A. De Rose, H. U. Heiss, and B. Linnert, "Distributed Dynamic Processor Allocation for Multicomputers," *Parallel Computing* 33, no. 3 (2007): 145–158.

29. J. Schneider and B. Linnert, "List-Based Data Structures for Efficient Management of Advance Reservations," *International Journal of Parallel Programming* 42, no. 1 (2014): 77–93.

30. D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience With Using the Parallel Workloads Archive," *Journal of Parallel and Distributed Computing* 74, no. 10 (2014): 2967–2982.

31. "Curta: A General-Purpose High-Performance Computer at ZEDAT," *Freie Universität Berlin*, https://doi.org/10.17169/refubium-26754 (visited February 09, 2024).

32. S. Shilpika, B. Lusch, M. Emani, et al., "Toward an In-Depth Analysis of Multifidelity High Performance Computing Systems," *2022 22nd IEEE International Symposium on Cluster*, *Cloud and Internet Computing* (*CCGrid*) (2022), 716–725.

33. S. R. Alam, J. Bartolome, M. Carpene, K. Happonen, J. C. LaFoucriere, and D. Pleiter, "Fenix: A Pan-European Federation of Supercomputing and Cloud E-Infrastructure Services," *Communications of the ACM* 65, no. 4 (2022), 46–47.

34. H. Qiu, C. Xu, D. Li, H. Wang, J. Li, and Z. Wang, "Parallelizing and Balancing Coupled DSMC/PIC for Large-Scale Particle Simulations," *IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*) (2022), 390–401.

35. L. L. Nesi, L. M. Schnorr, and A. Legrand, "Multi-Phase Task-Based HPC Applications: Quickly Learning How to Run Fast," *IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*) (2022), 357–367.

36. C. Schepke, N. Maillard, J. Schneider, and H. U. Heiss, "Why Online Dynamic Mesh Refinement is Better for Parallel Climatological Models," *2011 23rd International Symposium on Computer Architecture and High Performance Computing* (*SBAC-PAD*) (2011), 168–175.

37. M. Buschmann, J. W. Foster, A. Hook, et al., "Dark Matter From Axion Strings With Adaptive Mesh Refinement," *Nature Communications* 13, no. 1 (2022), 1049.

38. K. C. Tang-Yuk, X. Mi, J. H. Lee, H. D. Ng, and R. Deiterding, “Transmission of a Detonation Wave Across an Inert Layer,” *Combustion and Flame* 236 (2022): 111769.

39. M. Chadha, J. John, and M. Gerndt, “Extending Slurm for Dynamic Resource-Aware Adaptive Batch Scheduling,” *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics* (*HiPC*) (2020), 223–232.

40. R. P. Becker, “Slurm PMIx Extension for Dynamic MPI Processes in a Plan-Based Environment,” 2024.

41. D. Nichols, A. Marathe, K. Shoga, T. Gamblin, and A. Bhatele, “Resource Utilization Aware Job Scheduling to Mitigate Performance Variability,” *IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*) (2022), 335–345.

42. B. Li, Y. Fan, M. Dearing, et al., “MRSch: Multi-Resource Scheduling for HPC,” *IEEE International Conference on Cluster Computing* (*CLUSTER*) (2022), 47–57.

43. S. Zrigui, R. Y. de Camargo, A. Legrand, and D. Trystram, “Improving the Performance of Batch Schedulers Using Online Job Runtime Classification,” *Journal of Parallel and Distributed Computing* 164 (2022): 83–95.

44. N. Ueter, M. Günzel, v. d G. Brüggen, and J. J. Chen, “Parallel Path Progression DAG Scheduling,” *IEEE Transactions on Computers*, 2023, arXiv Preprint arXiv: 2208.11830.

45. E. Frachtenberg, D. G. Feitelson, J. Fernandez, and F. Petrini, “Parallel Job Scheduling Under Dynamic Workloads,” *Workshop on Job Scheduling Strategies for Parallel Processing* (2003), 208–227.