Malleable Supercomputing Using Containers

Bachelor Thesis

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
High Performance Computing Group

Advisor and Examiner: Prof. Dr. Florina M. Ciorba
Supervisor: Dr. Osman Seckin Simsek

Author: Tom Rodewald
Email: tom.rodewald@unibas.ch

August 4, 2025

University
of Basel

Acknowledgement

I would like to convey my appreciation to Prof. Dr. Florina M. Ciorba for providing me with the opportunity to explore such an engaging and interesting topic. I would also like to thank my supervisor, Dr. Osman Seckin Simsek, for his support, engagement, and adaptability that went into this research. His help was especially helpful when reaching dead ends and offering new perspectives.

Abstract

Traditional high-performance computing (HPC) systems rely on a rigid resource allocation model, where the number of compute nodes or CPUs must be statically defined prior to job execution. This inflexible model can lead to inefficient resource utilization when jobs allocate more resources than they can use. Malleability offers a solution; however, it remains largely unsupported in existing software solutions for HPC environments due to limitations in common parallelization frameworks, such as MPI and OpenMP.

This thesis investigates whether malleability can be achieved in containerized HPC workloads by retrofitting applications. The proposed framework leverages dynamic OpenMP thread pool manipulation in combination with checkpoint and restore operations for containerized environments using Docker and CRIU. Multiple hybrid benchmarks were containerized and evaluated under varying malleability configurations. Experiments show that this malleability approach offers performance improvements in environments with spare resources and preserves correctness across dynamically changing configurations, with only modest overhead introduced by checkpoint and restore operations in benchmarks that have a small memory footprint. The results demonstrate that practical malleability for thread-level parallelism can be achieved using userspace tools alone, offering a viable path toward more adaptive, efficient, and portable HPC execution models. This thesis contributes a flexible and reproducible framework for integrating malleable execution into existing HPC infrastructures.

Contents

Chapter 1

Introduction

High-performance computing (HPC) environments have, for decades, relied on a static and rigid allocation model, where jobs are declared at submission time with a fixed number of nodes or CPUs, and those resources remain allocated, regardless of whether the application can fully use them, until termination. This model simplifies scheduling and execution, yet it can lead to two problematic scenarios of wasted processing power. The first is when users overestimate the resources required, and a job is not utilizing all of its available resources, resulting in other jobs having to wait in the queue till this job has finished execution. The other scenario occurs when not all available resources are distributed across all running jobs, resulting in underutilization among the available resources.

A possible solution to this problem is malleability, the ability of a job to scale its resource usage during execution dynamically. This thesis investigates whether containerized HPC workloads is able to offer a solution for malleability within existing HPC applications.

1.1 Motivation

The growing scale of HPC workloads presents challenges related to resource utilization and rigid scheduling. Cloud environments solve a similar problem by utilizing elasticity to dynamically scale instances at runtime in response to the availability and demand of resources. Similar to the example of elastic resource management in cloud computing, there is a growing interest in adapting traditional HPC systems to improve system utilization.

Fulton et al. [2023] has investigated existing research in the area of containerization technologies in HPC, demonstrating the benefit of using containerized HPC workloads. However, the benefit of containers for malleability has not yet been explored.

1.2 Area of Interest

Malleability in HPC refers to the ability of applications, commonly referred to as jobs within HPC environments, to adapt to a changing environment with either more or fewer physical resources available without the need for termination or restarting of the application.

Malleability provides crucial benefits within HPC environments, offering flexibility and allowing more important jobs to take priority without requiring the termination of already running jobs or scaling up a job during a time when no other job is running. Providing malleability functionalities requires three interoperative parts in order to function. These include the scheduler, which determines when a change in available resources is necessary, a resource manager that provides more or less physical resources, and finally, an application or framework capable of adapting to these changes in the environment.

Most, if not all, HPC workloads are designed either with or around Message Passing Interface (MPI) and Open Multi-Processing (OMP), which is an execution model where the number of processes and their distribution across nodes is static, meaning neither offers malleability functionalities directly. Therefore, traditional malleability relies on techniques that offer task migration and dynamic load balancing.

In this thesis, we explore an alternative approach to resource adaptability through containers, which offer the portability of existing applications. Integrating parallelized applications into a containerized environment that supports malleability requires coordination or dynamic malleability at the MPI or

OpenMP (OMP) level, which is not currently present in the implementations of MPI or OMP. Within MPI applications, communicators must be handled in a way that allows processes to join or leave without violating collective operations. Although containers offer application portability and encapsulation, they introduce further complexities in terms of dynamic malleability. These complexities are mainly due to most container layouts rigidly defining resource allocations at startup. There is also a risk for potential dependency management issues between the HPC systems and containers that must be addressed before HPC environments can use them as foundations for malleable execution.

1.3 Thesis Contributions

The primary goal of this thesis is to explore and evaluate the use of containerized workloads to achieve malleability in HPC environments, aiming to determine whether dynamic resource scaling can be effectively implemented and maintained during application execution within containers while utilizing container runtimes. This thesis focuses on hybrid applications that take advantage of both MPI and OpenMP; however, it can also be adapted to pure OpenMP applications with minimal changes. An additional goal is to identify the practical limitations and requirements of current container technologies, evaluate their performance implications, and propose approaches for deploying malleable HPC applications.

The main focus is then to provide a framework for transparently offering malleability support to existing applications by utilizing thread management and core affinity, which simulates malleability. Therefore, this thesis proposes and evaluates a framework that utilizes containers to enable thread-level malleability in hybrid HPC applications and pure OpenMP HPC applications. The proposed implementation adjusts the OpenMP thread counts dynamically during execution without modifying the application's source code or relying on MPI malleability. The framework will then be evaluated and compared against static configurations.

By exploring how containers provide support for runtime scaling, this thesis contributes to the area of research focused at modernizing HPC infrastructure with dynamic resource management models. This thesis investigates whether containerization techniques are able to provide malleability in HPC workloads by supporting dynamic resource scaling during runtime and how this approach compares to static resource allocation in terms of efficiency, correctness, and overhead.

Chapter 2

Background

This chapter provides technical information that is relevant to the remainder of this thesis, offering some foundational knowledge that is needed to understand the concept and contributions of the thesis. It explains the two dominant parallel programming techniques in HPC, discusses the different job models, outlines container technology, and introduces checkpoint and restore tools that enable runtime malleability.

2.1 Parallelization Techniques

In the context of parallelized software, a parallel region refers to a segment of code that is executed concurrently by multiple threads or processes. In Open Multi-Processing (OpenMP), a parallel region is a block of code executed simultaneously by multiple threads that share the same memory region. Within Message Passing Interface (MPI), the entire application can be considered a parallel region, as each process (or more commonly referred to as rank) operates independently with its own memory region.

The first mentioned approach to parallelization is MPI, a paradigm that distributes work across multiple processes, typically assigned to nodes or CPU cores. Each MPI process manages its own memory region and program process, typically referred to as ranks, and maintains communication with the other MPI processes, resulting in difficulties in changing the count of processes during runtime. In our implementation, we exclusively use MPICH, a high-performance and widely adopted implementation of the MPI standard MPI. The MPI standard also provides bindings for various programming languages, such as Fortran, C, and C++. However, C++ is considered deprecated in more recent versions of MPI implementations. A major benefit of MPI is that it can span across multiple nodes and split the existing workload across multiple machines. Another benefit of using MPI is its widespread adoption in almost every existing HPC environment. Its main drawback, however, is the relatively more complex development effort required from developers, which involves manually partitioning the data, managing buffers, and handling necessary communications. It is also worth considering that MPI does not share its memory, which is a benefit in terms of security but also a drawback in terms of efficiency.

Another approach to parallelization is OpenMP, a shared memory model that divides work among threads, allowing regions of code to be executed by multiple threads in parallel. Thread creation is handled at runtime so it can be effected by libraries that alter the number of threads dynamically, offering a form of malleability. The largess benefit of OpenMP is its ease of use, as it requires minimal code adjustments when using annotation macros within loops. However, OpenMP's reliance on a single shared memory region, limits its functionality to a single system, or collection of systems that shares a memory region. Another drawback of OpenMP is its inherently nondeterministic approach to ordering parallel regions, which can introduce race conditions and lead to failures that are hard to reproduce and fix.

When developing an application that combines these parallelization techniques, it is referred to as a hybrid application. Hybrid applications are primarily developed for HPC systems and use both MPI and OpenMP to combine distributed and shared memory parallelism. Almost every application that will run inside an HPC environment will adopt at least one of the two introduced parallelization techniques, MPI and OpenMP, or may combine them in a hybrid design.

2.2 Job Models in HPC Scheduling

HPC scheduling techniques can be categorized into four application models based on their execution behavior regarding resource usage. The first is a rigid job model, which uses a fixed resource assignment, meaning that it requires a fixed number of nodes or cores specified at submission time, which remains unchanged throughout execution. In contrast to rigid jobs models, moldable job models have the ability to provide adjustments among different configurations just before execution begins, in other words, at start time, but do not allow changes during execution. Then, there are malleable job models, which introduce the ability for an application to dynamically expand or shrink its resource utilization in response to system availability. Lastly, there are adaptable job models that not only offer the ability to modify their resource environments during execution but can also change their underlying algorithm or communication patterns at runtime. The methodology proposed in this thesis falls within the malleable job model.

2.3 User space Kernel Space

In the area of HPC environments is the total address space split among two regions, namely user space and kernel space. Kernel space refers to the set of addresses that are maintained by the operating system and a small set of trusted, and therefore privileged, components. Code running within kernel space has the ability to execute privileged instructions or read protected memory locations. Applications running inside kernel space have the ability to interact with every defined component of the system and is therefore able to corrupt the entire system. Due to this ability to corrupt the entire system does kernel space access remain restricted to only the most relevant components. All other areas that do not require this level of access to the system are located within so called user space. User space encapsulates all application binaries, shared librares and other tools. This separation provides security and fault isolation for systems, by ensuring that a malicious or corrupted application is not able to overwrite critical system components, the kernel or scheduler itself, while also allowing the users the ability to run most software.

2.4 Containers

Containers are a lightweight virtualization technology that encapsulates an application's binary and its dependencies into a standalone environment. Unlike entirely virtual environments, which emulate entire operating systems and hardware, containers share the kernel of the host system on which they run. This results in less overhead and faster response times. These advantages make them particularly useful in HPC environments, where performance and resource efficiency are desired. Popular container runtimes, including Docker, Podman, and Singularity (renamed to Apptainer in a more recent stage), provide isolated environments for executing applications based on their defined environment.

A key limitation of malleability is that most runtimes assume the processing environment to rigid throughout execution. This thesis, therefore, treats containers as both a means due to their portability and dependency isolation, as well as a drawback due to the initial rigid resource allocation.

2.5 Checkpoints / Fault tolerance

This thesis only uses containerd, a lightweight container runtime that provides container management support on a single host and is based on the Open Container Initiative, and Docker for containerized environments, which are used not only to encapsulate HPC applications but also to facilitate resource malleability. By dynamically adjusting container CPU affinities and OpenMP thread counts, container runtimes are able to simulate changes in resource availability. As a result, containers are a favorable technology for runtime adaptation, particularly in shared or elastic environments.

One tool that offers checkpointing is Distributed Multithreaded Checkpointing (DMTCP), which operates entirely in user space to enable checkpointing for applications that utilize parallelization techniques. A benefit of offering a solution that operates in user space is that users within HPC environments can utilize these techniques without requiring elevated or more than already available permissions. DMTCP is able to capture the complete runtime state and then save it to the disk of an application, which enables restart and migration across different nodes.

Another commonly used tool is Checkpoint/Restore in Userspace (CRIU). It offers the ability to checkpoint individual processes and operates by freezing a process and saving its complete runtime state to disk, which then can later be restored. CRIU is integrated into Docker as an experimental feature and is particularly useful for containerized workloads. However, its support is limited to recent Linux kernels and system configurations.

An older tool that provides checkpoint functionalities is Berkeley Lab Checkpoint/Restart (BLCR), which was an earlier checkpointing system developed for HPC environments, which required kernel level access. It allowed for checkpointing of userspace processes and integration with job schedulers, such as SLURM. Although BLCR was influential in early fault tolerance research, it has seen reduced maintenance and adoption in favor of solutions that do not require applications with kernel level access in favor of user space solutions, such as DMTCP and CRIU.

Chapter 3

Related Work

Malleability within HPC environments has become a growing area of research aimed at optimizing available resources and enabling applications to utilize more resources, tackling different bottlenecks that prevent HPC jobs from resizing at runtime. This chapter surveys the most influential work in three different areas, explaining what has already been solved and where gaps remain.

3.1 State-Capture and Container-Centric Approaches

Theoretical models and simulations have addressed the topic of malleability, as demonstrated in Iserte et al. [2025]. However, practical realizations remain largely unexplored due to the rigidity of common parallel programming frameworks and the inherent complexity of runtime adaptation. This thesis, which explores malleable supercomputing in containerized environments, bridges these efforts, aiming to demonstrate that runtime reconfiguration and adaptation can be practically achieved without modifying the source code of existing applications.

Among the most directly related contributions is the work by Rodríguez-Pascual et al. [2019], who demonstrated the potential of job migration through checkpoint and restore operations using DMTCP. Their work is fundamental for understanding checkpoint and restart operations within user space in real-world environments. While DMTCP and CRIU differ in architecture, both offer transparent state capture and recovery, which this thesis uses within Docker containers to support runtime reconfiguration. Similarly, the work by Ansel et al. [2009] provided further operational insights into DMTCP, and their approach validates the effectiveness of checkpointing in enabling the flexibility of job execution.

Al-Dhuraibi et al. [2017] extended the concept of adaptability to Docker containers with ELASTICDOCKER. This system supports vertical scaling by dynamically modifying CPU and memory allocations using control groups and CRIU for dynamic resource adjustments. The work of Al-Dhuraibi on ELASTICDOCKER is closely related to the work presented in this thesis, which also relies on the Docker checkpoint and restore feature using CRIU. Their contribution to ELASTICDOCKER lies in providing data that supports the viability of integrating autonomic computing principles into container environments, which this thesis builds upon in designing runtime malleability mechanisms.

Process Checkpointing and Migration (PCM), as discussed by El Maghraoui et al. [2007], provided another area towards malleability. Their PCM framework captures the execution state of MPI applications and restores it on different nodes using merge and split operations. As a result, the adaptation of available resources and job migration are possible, and overall flexibility is increased. Although designed for cluster-level management, the underlying principles of state capture and recovery are directly applicable to this thesis and its use of CRIU to modify the runtime configuration of a containerized application. Likewise, BLCR (Berkeley Lab Checkpoint/Restart), presented by Hargrove and Duell [2006], showcased an early attempt to enable process-level checkpointing within Linux environments. Although BLCR lacks modern support, it serves as a historical foundation from which systems like CRIU have evolved.

Containerization in the HPC space was also explored by Fulton et al. [2023], who investigated how Docker and Singularity can enhance the deployment, reproducibility, and portability of existing applications. The proposed results regarding container runtime behavior and integration with existing job scheduling systems provided a basis on which HPC environments could use containers, reinforcing the premise of this thesis that Docker, along with CRIU, can enable adaptive supercomputing without requiring specific adaptation.

3.2 Programming-Model Malleability (MPI & OpenMP)

The challenge of integrating malleability in MPI-based applications is taken up by Martín et al. [2013] with FlexMPI. FlexMPI is an extension to the MPI-2 standard that enables the dynamic resizing of communicators, allowing MPI applications to expand or shrink the number of ranks dynamically. FlexMPI introduces an intermediate runtime layer, EMPI, to manage these changes. While conceptually powerful, FlexMPI requires adjustments to the application source code. In this thesis, FlexMPI was tested using a custom shim layer; however, practical integration with existing applications proved infeasible due to the significant assumptions and required adjustments at the application level. One major problem with FlexMPI's provided controller is that it assumes all existing applications are those provided by FlexMPI itself. Nonetheless, FlexMPI remains an important work in the domain of HPC malleability, illustrating both the potential promise and probable pitfalls of full MPI malleability.

Utrera et al. [2004] also explored MPI malleability through folding, where applications adapt the number of processes used during execution based on resource availability. While this approach does not appear to be compatible with containerized MPI jobs as explored in this thesis, it provides a reference for how the adaptability of jobs can be realized and the performance impact it can have on overall system throughput and resource efficiency.

Several papers work towards adaptability within OpenMP. The approach proposed by Chen and Long [2009] uses machine learning to optimize the parallelized loops within an OpenMP application based on runtime conditions using multi-versioning. Using their adaptive runtime improves efficiency across a range of platforms and workloads, reinforcing the potential benefits of runtime responsiveness. With a similar premise to Chen and Long [2009], Qawasmeh et al. [2015] presented APARF, which uses hybrid machine learning models and profilers to predict the optimal scheduling strategy for OpenMP applications. Despite their technique not introducing any form of malleability in the sense of available hardware changes, their contribution to the internal adaptability of the program complements the work of external runtime reconfiguration.

Scherer et al. [2000] also addressed adaptability within OpenMP applications, introducing a system that automatically adjusts the number of active processes based on runtime resource availability. Their solution inserted points around task boundaries, allowing for the external adjustment of application behavior. Despite their approach requiring a modified OpenMP runtime, their results support the feasibility of externally controlled applications in adapting to changing conditions, a concept this paper demonstrates through thread rebinding within containerized environments.

3.3 System-Level Frameworks and Scheduling Theory

The frameworks proposed by Buisson et al. [2005] supplemented both dynamic resource management and scheduling. With their adjustments of parallel components, they proposed mechanisms for runtime reallocation of parallel tasks in response to changes in the execution environment. Although this framework assumes support for malleability already exists, the structural runtime model they described helped to adapt the runtime logic designed in this thesis. They also introduced policies for multicluster systems on how to reallocate resources among existing executing jobs, aiming to improve overall system throughput. While their methods require already malleable applications, they validated the need for dynamic schedulers that can leverage the malleable behavior of these applications.

Finally, McCann et al. [1993] proposed and analyzed dynamic resource scheduling techniques to manage malleable jobs, with a focus on trade-offs between fairness and overall job-specific resource allocation. Hungershöfer et al. [2001] expanded on this by introducing a policy aimed at maximizing overall system throughput. These works highlight the general importance of providing schedulers who can take advantage of malleable execution, although it remains entirely theoretical.

Collectively, these works highlight the necessary adjustments that need to be made before implementing malleability into existing HPC environments. While some studies addressed malleability through application design, process state adjustments, or custom parallelization implementations, all aimed to achieve a similar ability for applications to adapt to changing environments. This thesis combines multiple facets of these approaches, primarily mandating checkpoint and restore operations, dynamic environment adaptation via containerization, and dynamic thread scaling in order to demonstrate that practical malleability in HPC can be achieved with minimal intrusion into application code and without replying on specialized or tightly coupled infrastructures.

Chapter 4

Methodology

This chapter presents the experimental methods and technical approaches used and explored in order to enable malleability in native and containerized HPC applications. It details the mechanisms that provided OpenMP malleability, discusses the limitations encountered with MPI malleability, and provides insight into a series of unsuccessful yet informative attempts conducted during the work. All of these efforts combined demonstrate both the progress made and the challenges in adapting HPC applications to dynamic execution environments.

4.1 Malleability approaches

4.1.1 OpenMP Malleability

The primary mechanism introducing malleability within this thesis is based on the dynamic adaptability of OMP thread pools. Since OMP statically determines the thread count at the program start by default, it prevented dynamic adaptation under standard execution. To overcome this limitation, a custom LD_PRELOAD library was developed to intercept and overwrite the environment variables used in thread initialization calls at runtime as described in Figure 4.1. This mechanism enables the thread count to be dynamically adjusted during execution without requiring modifications to the application source code. The means of adjustments are controlled through an external file located at ”/tmp/force_nthreads.txt”. However, this approach demands the application to create new parallel regions throughout the execution for the change to take effect as the ”force_nthreads.txt” file is read before each parallel region.

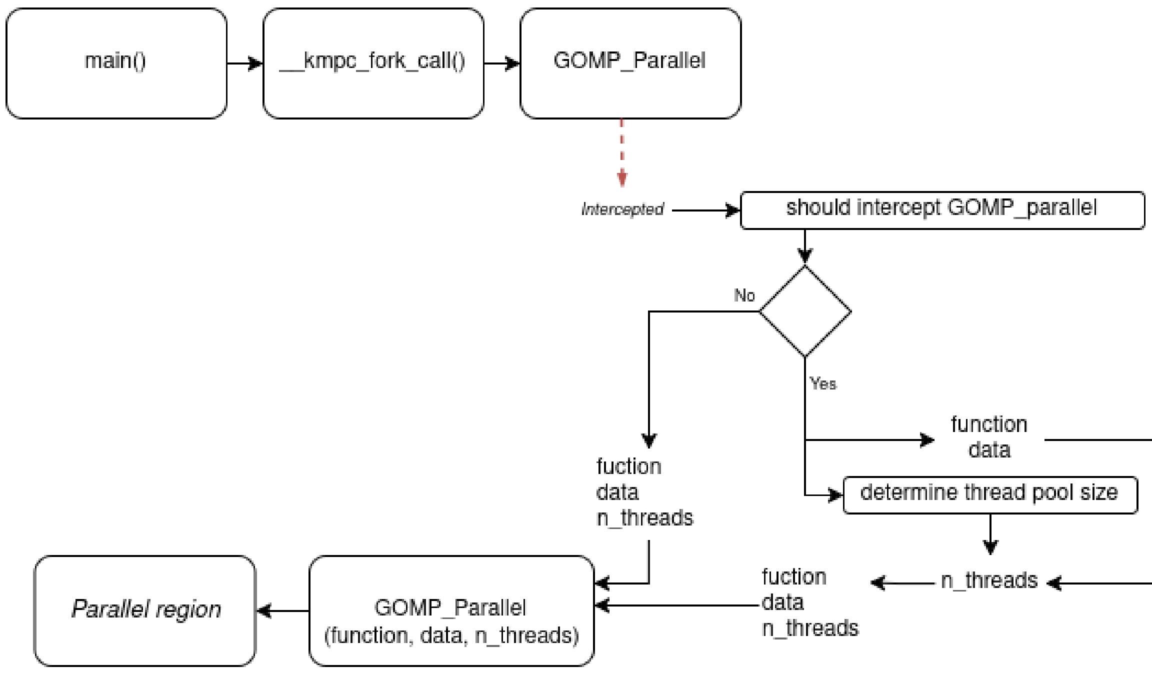

Figure 4.1: Visualisation of the process stack for OMP parallel application when using the LD_Preload library

To enforce hardware affinity aligned with thread count changes, the task set utility, which allows pinning specific processes to specific CPUs, was used in conjunction with the preload library. Such an approach allowed dynamic remapping of an application's thread execution to specific physical CPU cores, effectively simulating resource allocation changes during execution.

Thread malleability was evaluated using the NAS Parallel Benchmarks (NPB), LULESH, and AMG, where the OpenMP thread pool can be updated at runtime. Application output was monitored to track application progress and determine when thread count and core affinity adjustments should be triggered.

4.1.2 MPI Malleability

Similar to the approaches used to achieve malleability in OpenMP, an attempt was made to adjust the number of ranks using another LD_Preload library, focused on MPICH instead of OpenMP. However, due to the MPI implementation of ranks being more global rather than redefined for each parallel region, unlike OpenMP, this did not result in a successful implementation but instead caused direct crashes to the application.

Subsequent investigations then focused on FlexMPI, a runtime extension built on MPICH that leverages the MPI-2 dynamic process model, the second iteration of the MPI standard, in order to enable communicator resizing. FlexMPI is not interchangeable with existing MPI implementations, such as MPICH, despite being built upon them. As a means of circumventing this limitation of FlexMPI, a source-to-source transpiler (SHIM) was proposed in the initial stages of this thesis, which can be parsed as a compilation argument to transpile from established MPI function calls to the FlexMPI interface. This tool aimed to automatically translate standard MPI calls into their FlexMPI counterparts, effectively allowing legacy codebases to benefit from dynamic resizing without requiring direct modification. However, this approach proved to be infeasible in practice, as being compiled against FlexMPI does not provide immediate malleability. The programs were able to receive the function calls provided by FlexMPI but did not act on the malleability calls and were waiting for specific routines to be called that instantiate the malleability action.

Complicating matters further, many MPI applications, particularly the NAS benchmarks, are also written in Fortran. As a result, automated translation and linkage against the FlexMPI interface become significantly more complex, if even feasible at all. These challenges, combined with the added dependency overhead and the need to recompile applications from the source, ultimately led to abandoning the SHIM in favor of alternative approaches.

Therefore, it was concluded that FlexMPI requires substantial source code adjustments as it replaces standard MPI function calls with its own interface, meaning that any existing MPI application must be fully recompiled. Moreover, developers would need to implement malleability-aware function calls, which is impractical, especially considering that most applications are distributed as compiled binaries rather than source code available for direct compilation on the machine.

Alternative technologies, such as Adaptive MPI (AMPI), which is built on Charm++, were also briefly explored but abandoned due to a lack of existing benchmarking software and the probable future desertion of the work, as most HPC environments are designed around C applications rather than Charm++ ones.

In summary, despite having theoretical support in experimental runtimes like FlexMPI, the combination of tooling immaturity, invasive integration requirements, and limited support for typical benchmarking workloads rendered MPI malleability infeasible within the constraints of this study.

4.2 Checkpoint and Restore of Docker containers

An approach to achieve malleability is through checkpointing and adjusting the physical resources. While container environments offer portability and reproducibility, the integration of physical resource remapping during runtime creates challenges that must be overcome, especially for parallel applications, which this thesis focuses on.

Later stages of this work implemented containers as a means to remap physical availability, aiming to evaluate the practical implications of malleability on the university's HPC environment, MiniHPC. These tests were designed to expose both the potential and the limitations of runtime adaptation in existing HPC infrastructures, especially in the context of retrofitting to established solutions.

4.2.1 CRIU and DMTCP

A way to achieve this physical reallocation is by using Docker's experimental feature to enable checkpointing and restoring of containers. However, initial attempts conducted to show container checkpointing and migration failed, primarily due to the MiniHPC's outdated kernel (version 3.10). This kernel version lacks critical features required by CRIU, particularly those introduced in CRIU 3.16 and later, which are required to checkpoint parallel applications.

There were attempts made to install and test the experiments locally; however, this proved unsuccessful at first, as the local kernel version (5.1.3) was still outdated and did not include mandatory fixes related to the thread and cgroup handlers introduced in kernel version 5.6 and newer. There have been two updates to CRIU that are assumed to be the cause of why parallel applications cannot be restored without failing, namely CRIU version 3.16 and CRIU version 3.18. CRIU 3.16 expanded the support for restoring containers into already defined pods, while CRIU version 3.18 introduced fixes and improved the handling of parallel applications by dumping cgroup controllers for each thread. Despite these limitations, applications that used parallelization techniques such as OMP and MPI could successfully be checkpointed and restored, under the condition that there is at most one rank and one thread per rank, effectively not having parallelization, most likely due to information loss would be lost in older versions of CRIU when attempting to checkpoint more than one process at a time. In contrast, parallel applications consistently failed, with application hangs and incomplete state restoration. Notably, these failures produced no internal CRIU logs, which would indicate any specific error, indicating that the failures occurred during CRIU's initialization phase prior to launching any processes inside the container. The absence of logs reinforces the assumption that existing incompatibilities with older kernels are more likely than configuration errors.

Additionally, it is worth noting that even with updated CRIU versions further configuration changes are still necessary to restore parallel applications that were previously using the MPI protocol successfully. Mainly, the ghost_file_limit has to be increased beyond the total memory footprint of the existing MPI application, which does not require any additional permissions. However, it appears that CRIU must be executed as root; otherwise, the applications fail to restore.

To overcome CRIU's limitations, DMTCP was also evaluated as an alternative that could improve functionality for existing HPC environments. Unlike CRIU, DMTCP offers more direct support for parallel application checkpointing within the requirements that everything is without privileged access. However, attempts to combine DMTCP with Docker proved unsuccessful because of incompatibilities in signal handling, resulting in failed checkpoints and process deadlocks.

4.2.2 Experimental Platform

As a solution to the problems mentioned above with CRIU and DMTCP, all subsequent experiments were conducted on a modern local workstation, with topology visualized in Figure 4.2, running on a modern kernel (version 5.15), where full checkpoint and restore capability has been achieved and demonstrated successfully. Using a local system also provided access to higher privileges, which were necessary to fulfill other dependencies introduced by CRIU.

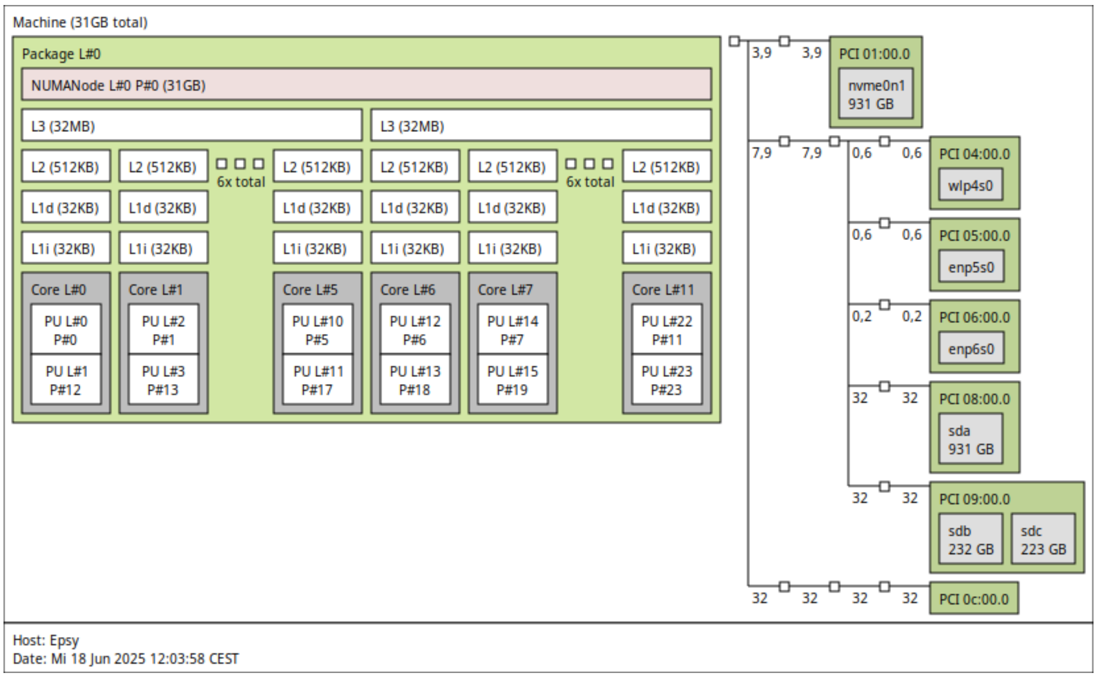

Figure 4.2: Topology overview of the local system used for malleability experiments

Figure 4.2 visualizes the topology of the local workstation machine. It consists of a single NUMA node with 12 Cores and SMT (Simultaneous Multithreading), which is a technology that allows one physical core to execute multiple, in this case two, threads concurrently.

Moving to a modern system allowed this thesis to have more control over resource management, as would typically be achievable when using HPC environment-specific software, such as Slurm, and software support. The method still presents a set of controlled and reproducible malleability experiments that are comparable to those run on an actual HPC environment.

4.2.3 Containerization and Execution Framework

All benchmark applications were containerized using Docker, with the base image set to debian:bookworm-slim, providing a minimal and consistent environment for reproducible and comparable experimentation. Each container was installed with MPICH and OpenMP, along with all other necessary components required for hybrid applications, as all benchmarks to be tested are using a hybrid parallelization approach. This setup enabled the execution of hybrid parallel workloads, composed of MPI ranks and OpenMP threads, within containerized environments. When attempting to run the docker image the flag, shm-size, must be explicitly set to a minimum of the memory constraints of the application. A means of modifying the file, force_nthreads.txt, is by parsing it as a volume bind mount using the following command to dockers run script, ”-v /location/of/mount:/tmp/force_nthreads.txt”.

The selected benchmarks, combined, provide a variety of computational patterns in order to test the shortcomings and benefits of malleability mechanisms in both growing and shrinking scenarios that would not exclusively benefit from raw compute or I/O speeds.

4.2.4 Conclusion about Contanierization

In conclusion, although containerized parallel workloads were executable, the implementation of runtime malleability via checkpoint/restore was unreliable and introduced additional system dependencies that are not easily adaptable to existing solutions. Such challenges highlight the current immaturity of malleability within containerized environments in existing HPC systems.

4.2.5 Resulting solution

This thesis explores different approaches to malleability and concludes with a working solution that operates by checkpointing applications to disk and utilizing the proposed LD_Preload library before a checkpoint occurs to modify the assumed size of available threads within the OpenMP thread pool. A visualization of this approach is also demonstrated in Figure 4.3 for applications that expand their available resources and Figure 4.4 for applications that decrease their available resources. The approach consists of four different regions, those being the previous configuration of available resources with the relevant number of MPI ranks and OMP threads per rank. The next stage involves checkpointing the entire process to disk, where the state of each rank is saved in a single checkpoint file. With the checkpoint file, the application can be moved to an environment with more available resources where it can be relaunched. The final phase occurs once the application has been restored inside the new environment, using a new set of available resources. This approach assumes that the application is utilizing multiple parallel regions and fits within the available memory of the host machine(s). It is also possible to use this approach, as shown in Figure 4.5, to allow overpopulation of the system resource, which can be useful in scenarios in which an application is waiting for other resources that are not immediately computable.

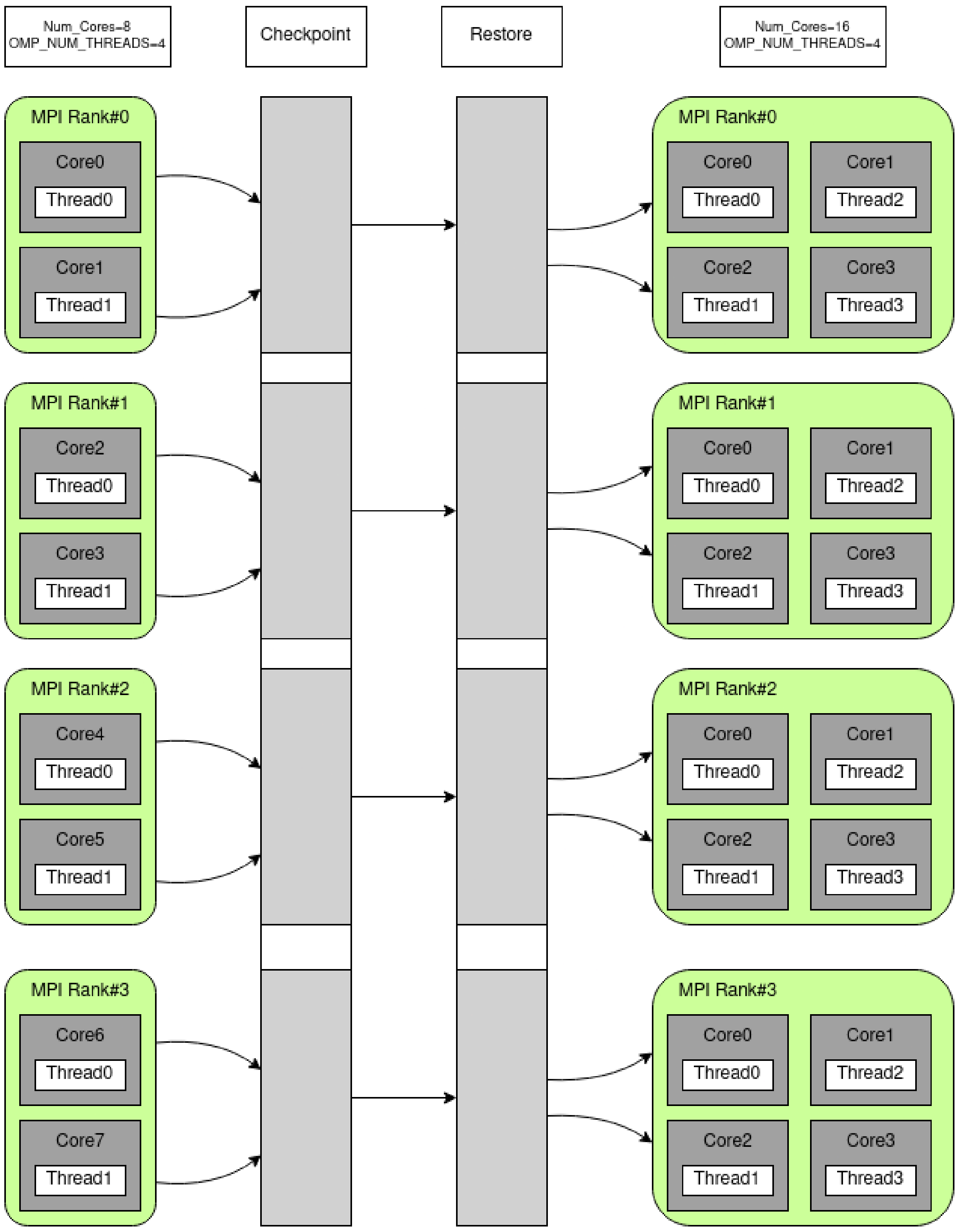

Figure 4.3: Malleability approach using containerized environments to increase the available resources from 4 ranks and 2 threads per rank to 4 ranks with 4 threads per rank

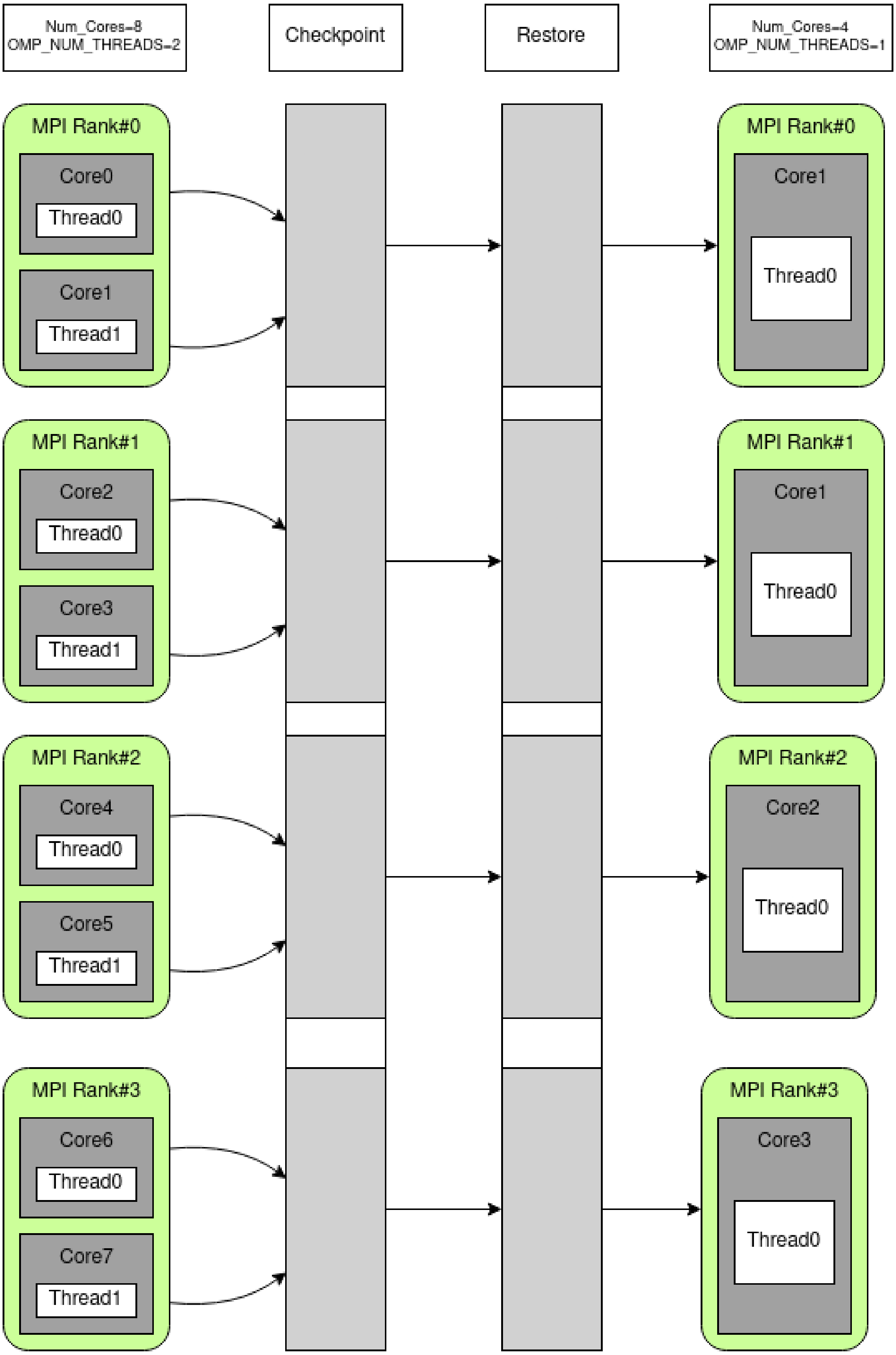

Figure 4.4: Malleability approach using containerized environments to decrease the available resources from 4 ranks and 2 threads per rank to 4 ranks with 1 thread per rank

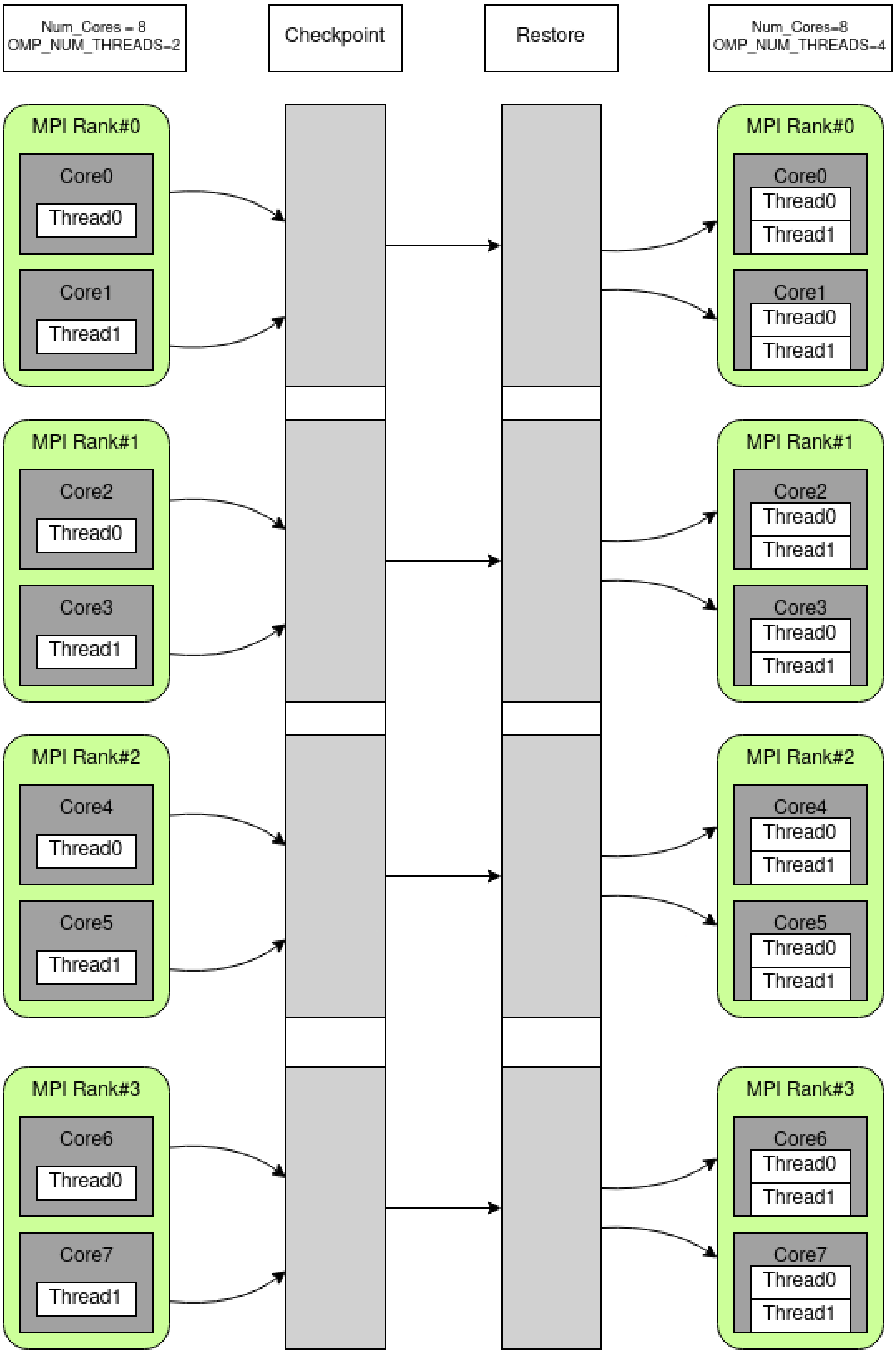

Figure 4.5: Malleability approach using containerized environments to decrease the available resources from 4 ranks and 2 threads per rank to 4 ranks with 4 threads per rank, but two threads share one available CPU core

4.3 Application insight

The experiments focus on three sets of benchmarking software, those being the "NAS Parallel Benchmark" (NPB), "AMG" and "Lulesh". The NPB suite is a collection of synthetic and widely adopted benchmarks developed by NASA Ames to represent key computational patterns in fluid dynamics and structured grid-based partial differential equation (PDE) solvers. These benchmarks model computational systems and applications typical in the realm of hydrodynamics and other scientific domains and are designed to evaluate the performance of parallel supercomputers.

From the NPB suite, we only considered benchmarks that offer multi-zone (MZ) variants, as they use hybrid parallelization. These include the Block Tridiagonal Solver (BT-MZ), which simulates compressible fluid flow using block-tridiagonal systems across multi-zoned structured grids and is primarily compute-bound. Next is the Scalar Pentadiagonal Solver (SP-MZ), which solves scalar pentadiagonal systems, placing high stress on memory access and point-to-point communication, and is also mostly compute-bound. The last is the Lower-Upper Symmetric Gauss-Seidel Solver (LU-MZ), which applies an iterative solver over multi-zoned domains and has strong synchronization and communication demands. It is compute-sensitive, meaning that while execution time is largely constrained by CPU speed, some portions are also memory-bound.

These benchmarks represent hybrid parallelism, combining MPI and OpenMP within zones, making them ideal for testing the impact of dynamic malleability. It is important to note that we exclusively use Class C problem sizes of the multi-zone variants. In the context of the NAS Parallel Benchmarks, a Class refers to the problem size and workload intensity, with Class C representing a larger and more computationally demanding test case designed for performance evaluation on medium to larger systems.

In addition to the NPB suite, two other applications developed by the Lawrence Livermore National Laboratory (LLNL) were selected, namely AMG and LULESH. The Algebraic Multigrid Solver (AMG) is a benchmark that consists of unstructured, sparse linear solvers, which exhibit irregular communication patterns and memory access, and is used in large-scale implicit PDE solvers. The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) software models a simplified hydrodynamics kernel commonly found in shock physics simulations and calculates a hydrodynamics model using a computational approach over a structured mesh in cubic form. It operates over unstructured 3D meshes, solving a Lagrangian form of the hydrodynamics equations, and is highly representative of production-scale finite element codes in terms of both computation and communication. Both are frequently used in Department of Energy (DOE) proxy applications that study and reflect more complex, realistic computation and communication characteristics than synthetic benchmarks.

Chapter 5

Results

This chapter presents the results obtained from a comprehensive set of factorial experiments, as defined in Table 5.1, evaluating the feasibility and performance implications of malleable HPC workloads executed within containers.

Table 5.1: Design of factorial experiments, total of 59 experiments.

Factors	Values	Properties
Applications	NPB LU-MZ	dT = 0.000100 | Mesh_size = 480 * 320 * 28 | Number of zones = 64 | Iterations = 400
	NPB SP-MZ	dT = 0.000670 | Mesh_size = 480 * 320 * 28 | Number of zones = 256 | Iterations = 400
	NPB BT-MZ	dT = 0.000100 | Mesh_size = 480 * 320 * 28 | Number of zones = 256 | Iterations = 200
	LLNL/Lulesh	Iterations = 500
	LLNL/AMG	N_x = 200 | N_y = 250 | N_z = 200 | Iterations = 22
Configurations (MPI Ranks ; OMP Threads)	NPB BT-MZ	1;1 | 1;2 | 1;4 | 1;8 | 2;1 | 2;2 | 2;4 | 2;8 | 4;1 | 4;2 | 4;4 | 8;1 | 8;2
	NPB LU-MZ	1;1 | 1;2 | 1;4 | 1;8 | 2;1 | 2;2 | 2;4 | 2;8 | 4;1 | 4;2 | 4;4 | 8;1 | 8;2
	NPB LU-MZ	1;1 | 1;2 | 1;4 | 1;8 | 2;1 | 2;2 | 2;4 | 2;8 | 4;1 | 4;2 | 4;4 | 8;1 | 8;2
	LLNL/LULESH	1;1 | 1;2 | 1;4 | 1;8 | 8;1 | 8;2 | 8;4
	LLNL/AMG	1;1 | 1;2 | 1;4 | 1;8 | 2;1 | 2;2 | 2;4 | 2;8 | 4;1 | 4;2 | 4;4 | 8;1 | 8;2
Scheduling techniques	Halfway malleability	after half of the iterations have been done will malleability adjustments occur
	Dynamic malleability	after a set time interval will the application change its available resources
Computing nodes	Home system	Ryzen 9 5900x (1 sockets, 12 cores each) P = 24 with hyperthreading, Pinning:`OMP_PLACES` = cores
Metrics	Metric 4	Total wallclock time (seconds)

5.1 Experimental Design and Evaluation Strategy

The experiments in this thesis were designed to evaluate malleability spanning multiple axes of variation. Five HPC benchmark applications were tested, including BT-MZ, SP-MZ, and LU-MZ from the NAS Parallel Benchmarks (NPB) suite, as well as AMG and LULESH, two mini-applications developed by Lawrence Livermore National Laboratory. Each application was evaluated under both static and dynamic scheduling conditions, incorporating techniques such as halfway malleation and time-based adjustments to simulate real-world resource variability. A diverse set of MPI and OpenMP configurations were explored, with careful control over thread counts and physical core assignments to assess the impact of different parallelization strategies. All experiments were executed on a Ryzen 9 5900X workstation featuring 12 physical cores and simultaneous multithreading (SMT), with CPU affinities managed using taskset to maintain consistent thread-to-core mapping throughout the tests, as illustrated in Figure 4.2. The primary performance metric was total wall-clock time, supplemented by checkpoint and restore overhead in malleable cases.

The NPB suite employs a system that prints the current iteration of the benchmark at every step to the standard output, allowing the script to have real-time data from the benchmark directly and to monitor current progress. All NPB benchmarks are designed to have exactly 200 iterations so the malleability approach can read the current iteration and mandate a malleability command at the 100th iteration. The LULESH benchmark employs a similar strategy, allowing the application to print its current iteration. However, this functionality has to be specifically enabled by the *-i* program argument. The total number of iterations for the application can also be controlled upon starting the application. The experiments mentioned in this thesis assumed a constant number of iterations, being 500. The AMG benchmark, however, does not offer functionality to print the current iteration, although the benchmark itself prints out that it will use 22 iterations in the configurations used within the experiments for this thesis. Therefore, this thesis attempted to estimate the average time before the halfway mark has been reached. Since the benchmark has an average runtime of over 20 seconds, the approach waits 10 seconds before introducing malleability requests, which this thesis refers to as Dynamic malleability, as shown in Table 5.1. All benchmarks provided their own means of validating the results that were provided, and they were validated successfully.

5.2 Performance analysis

5.2.1 Metrics

The five discussed applications were evaluated in static execution under various parallelization configurations. Additionally, there were three execution modes for each benchmark. The first and most straightforward approach is to execute the benchmark natively on the hardware without any containerization environment. This was then compared against the same benchmark being run inside a container with no malleability or resource availability adjustments. Finally, the primary research goal of this thesis involved running the application inside a containerized environment and using the malleability approach by reallocating available resources and core mapping. Depending on the benchmark, the method for choosing the malleability instruction changed, which will be discussed in the relevant part of the benchmark itself. The x-axis of the visualizations will represent the parallel execution configuration, specifically from a given MPI rank to another. MPI ranks will remain constant, as described in previous chapters. The results demonstrate the time taken for each configuration transition and provide insight into how effectively the malleability approach adapts resource allocation during run time. The benchmark execution time (in seconds) will be compared across five different execution modes: Native (number of OMP threads), Containerized (number of OMP threads), Malleable, Bare (number of OMP threads), and Container (number of OMP threads). Bare (number of OMP threads), for example, refers to the benchmark being run with a specific number of OMP threads per MPI rank. Moreover, the malleable bar references a benchmark being run inside a containerized environment while being malleated from the *initial_configuration* to the *later_configuration* at the specific point at which the benchmark has to be malleated.

5.3 NAS BT-MZ:C performance

5.3.1 NAS BT-MZ malleability with factor 0.25

The data shown in 5.1 shows that there is a significant overhead associated with the proposed malleability approach compared to both native and containerized execution. There is a general trend in which the containerized environment experiences a performance loss ranging between 5.6% and 10.1%, with an emerging trend indicating that configurations with more MPI ranks and fewer OMP threads per rank typically result in less overhead compared to configurations with higher numbers of OMP threads and fewer MPI ranks. This could be explained by differences in memory management, where OMP threads share a common memory space, leading to congestion and inefficiencies, whereas MPI ranks operate with separate memory regions, thereby creating the overhead associated with Docker.

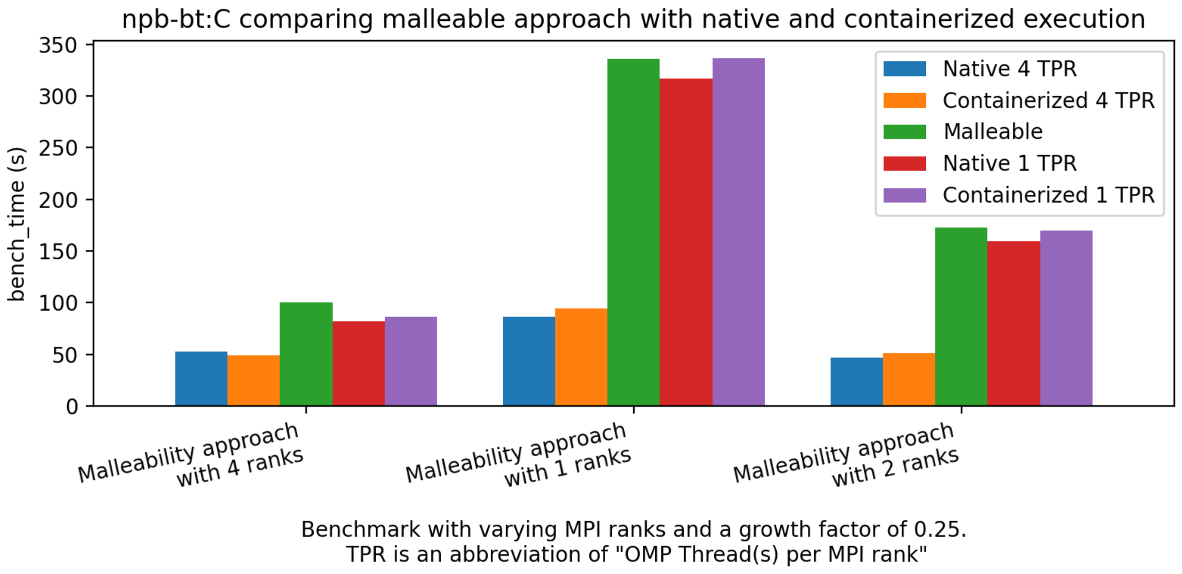

Figure 5.1: illustration of the performance results for the NAS BT-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 0.25, meaning that the number of OMP threads per MPI rank will be reduced by 75% halfway throughout execution.

Overall, this data highlights important performance implications when using the proposed malleability approach in combination with containerization, with a strong emphasis on carefully balancing MPI ranks and OMP threads based on application characteristics and available hardware resources. Figure 5.2 further reinforces this as it shows that the overhead introduced by checkpoint and restore operation is growing with the number of MPI ranks. The data also suggests that shrinking of available resources might result in a performance deficit but also validates that the concept behind malleability is possible in scenarios in which more important jobs get scheduled that are supposed to have priority without having to reschedule the currently running job outright, having to start at the beginning anyway with current implementations.

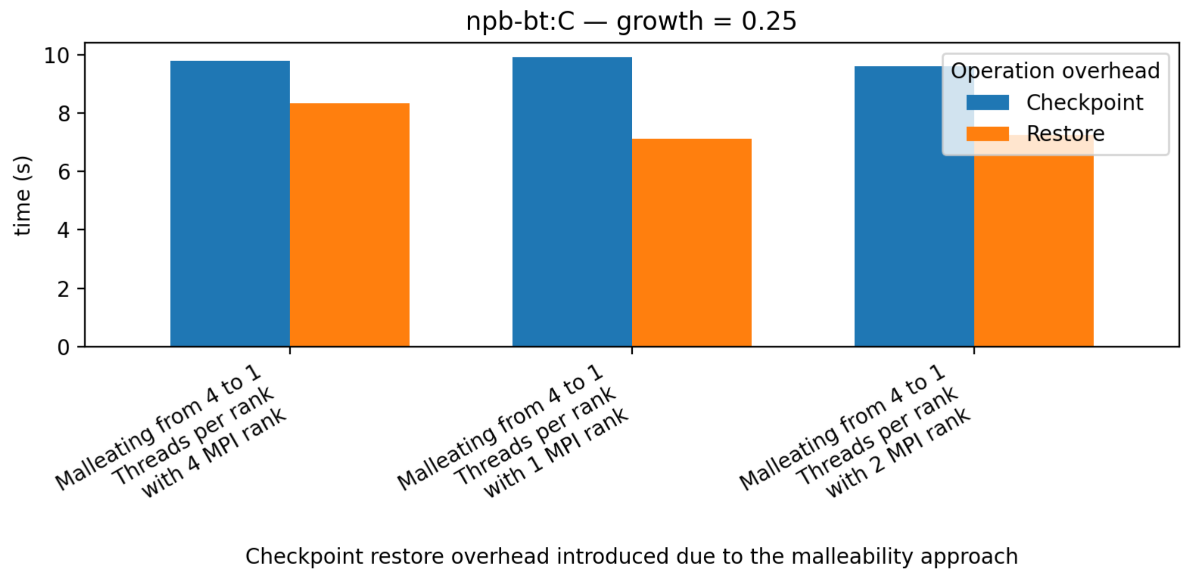

Figure 5.2: NAS BT-MZ experiments checkpoint and restore overhead

5.3.2 NAS BT-MZ malleability with factor 0.5

The data shown in 5.3 demonstrates a similar trend to that in 5.1 in which a significant overhead associated with the malleability approach can be seen compared to both native and containerized execution.

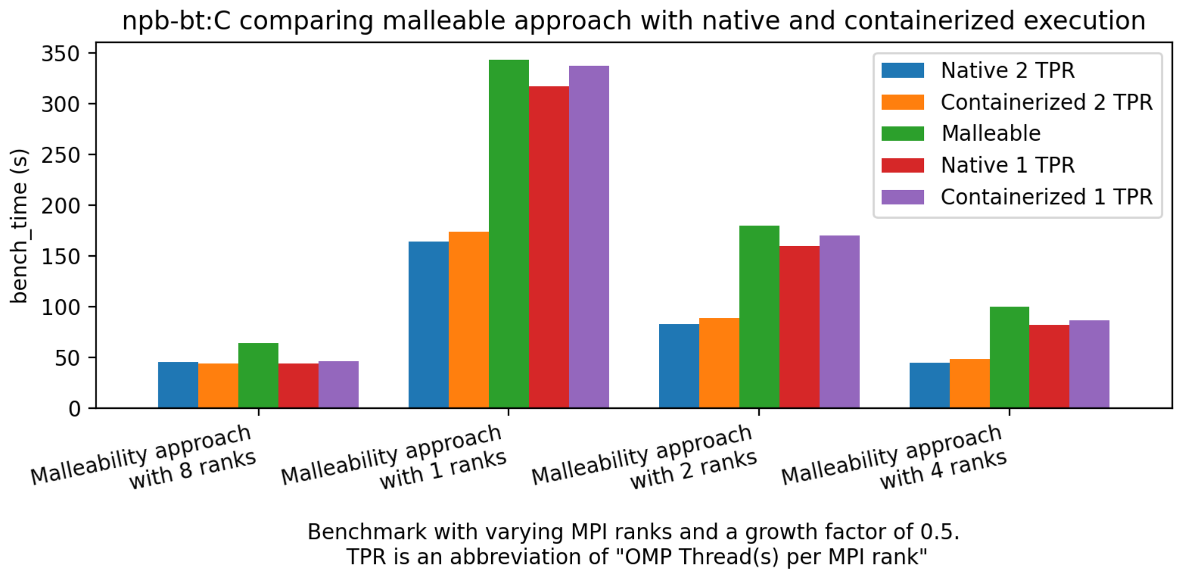

Figure 5.3: illustration of the performance results for the NAS BT-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 0.5, meaning that the number of OMP threads per MPI rank will be reduced by 50% halfway throughout execution.

Upon comparisons with the container overhead results from Figure 5.2, the data illustrates a performance loss ranging between 1.3%, with 8 MPI ranks, and 10.1%, with the same trend emerging, indicating that configurations with more MPI ranks and fewer OMP threads per rank typically result in less overhead compared to configurations with higher numbers of OMP threads and fewer MPI ranks.

Overall, this data highlights similar important performance implications when using the proposed malleability approach in combination with containerization, compared to Figure 5.1. The data suggests similar results, in which shrinking available resources results in a performance decrease. However, it also validates that the concept behind malleability is possible in scenarios where more important jobs are scheduled to have priority without requiring rescheduling, allowing the currently running job to be restarted outright. The overall checkpoint and restore operation overhead also remains similar to that shown in Figure 5.2 in terms of total overhead introduced.

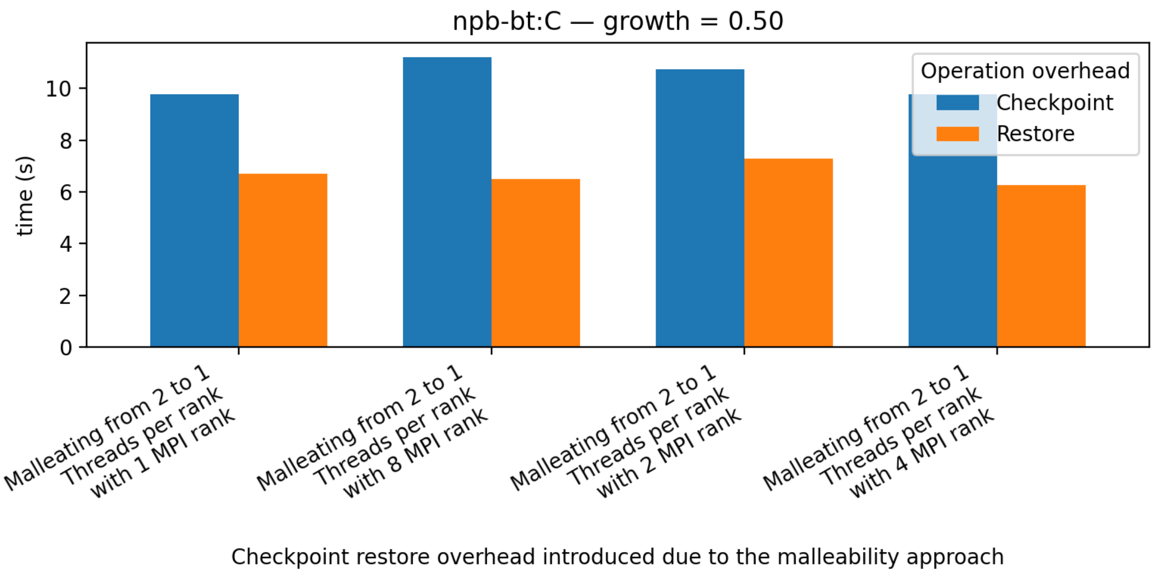

Figure 5.4: NAS BT-MZ experiments checkpoint and restore overhead

5.3.3 NAS BT-MZ malleability with factor 2.0

The results in Figure 5.5 illustrate the previously known overhead introduced by containerizing the benchmark, showing a modest increase in overhead compared to native execution. More importantly, under the same configuration, the completion time of the benchmark within a containerized environment outperforms the same benchmark in a native environment, exclusively in the configuration with 8 MPI ranks and 2 OMP threads per rank, with a performance increase of 2.3% in total completion time.

The observed performance increase could be attributed to two factors. The first factor could be that containerization enforces stronger resource isolation, improving cache affinity and simplifying thread management. However, the more probable reason involves regular run-to-run variances, where every benchmark run might be slightly different due to scheduling and core locality differences, as the overall performance increase is 2.3%. However, it is essential to note that this data represents an aggregated result from five consecutive benchmarks, which typically eliminates run-to-run variances.

More crucially, the malleability approach consistently outperforms the containerized version when using at most four MPI ranks. The data suggest that having more than four MPI ranks results in slower native execution time compared to the original configuration when running in a containerized environment. The checkpoint and restore overhead introduced to achieve the mentioned malleability approach causes this performance degradation. Therefore, it would not show a significant performance penalty for longer-lasting jobs, where the relative cost of this overhead diminishes over time.

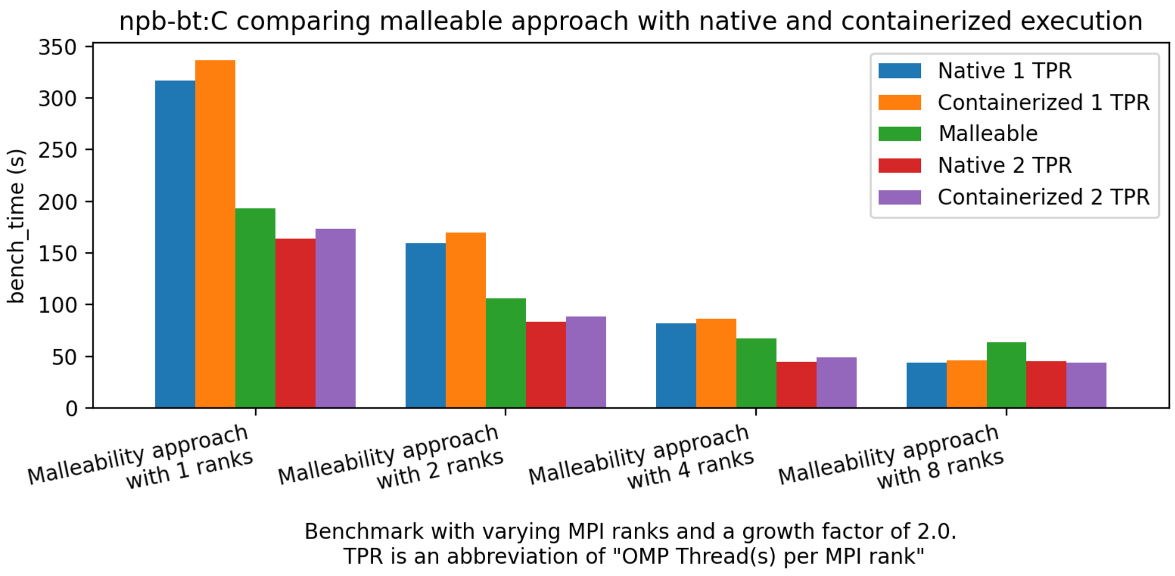

Figure 5.5: illustration of the performance results for the NAS BT-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 2.0, meaning that the number of OMP threads per MPI rank will be doubled halfway throughout execution.

The presented data shows that there is almost no overhead compared to running with a higher number of threads from the beginning in a containerized state. There exists an important result indicating that the completion time of the benchmark can, for smaller initial configurations, closely approximate the native execution time.

This data suggests that the performance gain achieved by leveraging malleability to expand an existing job can be beneficial in scenarios with resource abundance, enabling faster execution. It does, however, also point out that there is a limit to how much one application can scale when compared to the overall runtime of the job, considering the introduced overhead, as shown in Figure 5.6. When accounting for overhead, the overall execution time of the result with malleability enabled approaches to that of the containerized environment.

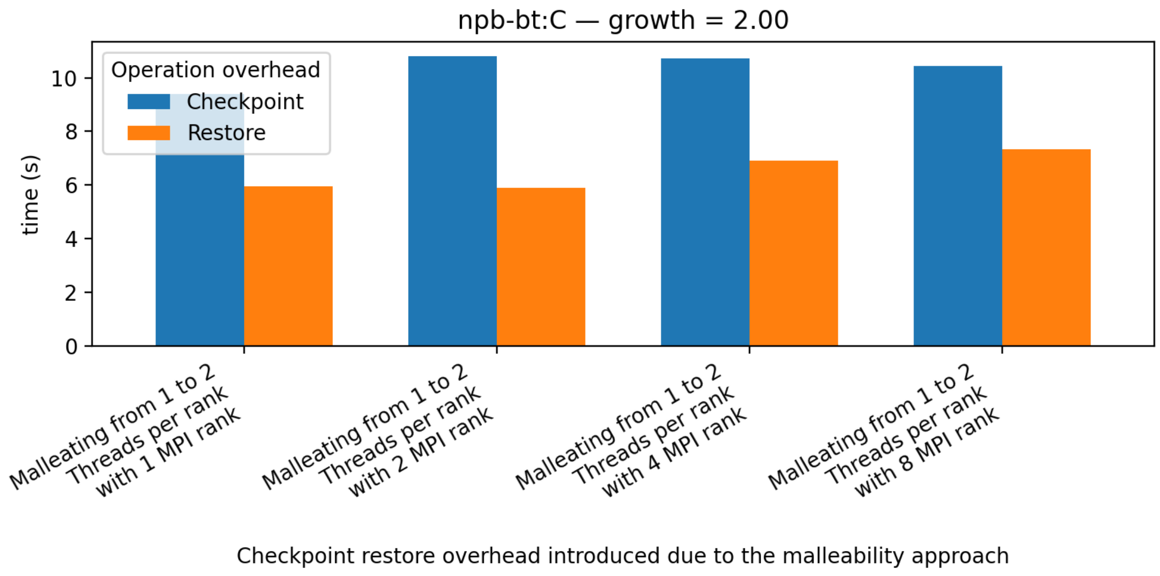

Figure 5.6: NAS BT-MZ experiments checkpoint and restore overhead

5.3.4 NAS BT-MZ malleability with factor 4.0

The results from Figure 5.7 show significant performance gains across all existing configurations, similar to Figure 5.5. This similarity in the result indicates that the completion time of the benchmark within a containerized environment outperforms the benchmark in a native environment, exclusively in the configuration of 4 MPI ranks and 4 OMP threads per rank, with a performance increase of 6.7% in total completion time. This observed performance increase could potentially be explained by the same two factors previously described. The first factor, which addresses resource isolation, appears less probable compared to the second factor for this data, given that the performance increase is only 6.7%, and thus could be explained by run-to-run variances. The impact of stronger resource isolation, however, cannot be underestimated, as it may partially explain the performance increase.

Additionally, the figure shows similar results regarding container overhead, as previously mentioned, with another notable exception in which the containerized benchmark outperforms the application running natively. This anomaly can be explained by the same assumptions as previously and will require a more in-depth analysis of the actual metrics.

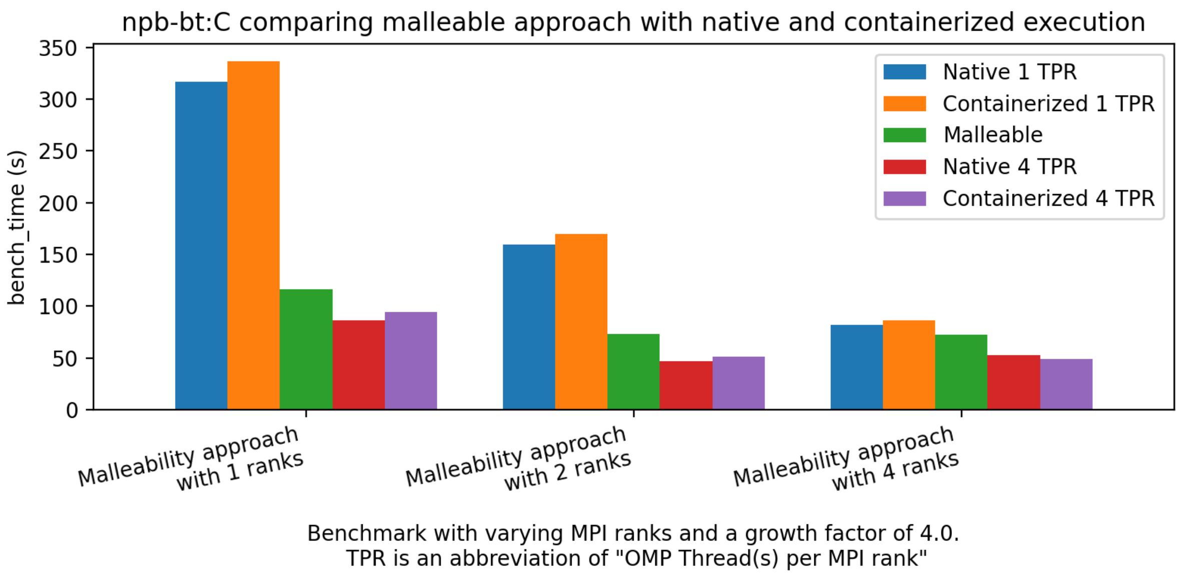

Figure 5.7: illustration of the performance results for the NAS BT-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 4.0, meaning that the number of OMP threads per MPI rank will be quadrupled halfway throughout execution.

We do, however, observe diminishing returns from 2 MPI ranks and 1 OMP thread compared to 4 MPI ranks and 1 OMP thread, with a growth factor of 4. As displayed in Figure 5.7, the overall performance improvement is only 66.30 seconds compared to 73.60 seconds, respectively, a performance increase of 1.1%. This marginal improvement is likely due to the job's overall short execution time, which is less than two minutes to begin with, and longer-running jobs could potentially achieve significant performance gains.

There is, again, a performance increase when running the benchmark with 4 MPI ranks and 4 OMP threads inside a containerized environment, which is most likely explainable by the same explanations as previously. The overhead introduced by having the checkpoint and restore operations also stays constant compared to the previous experiments, as shown in Figure 5.8.

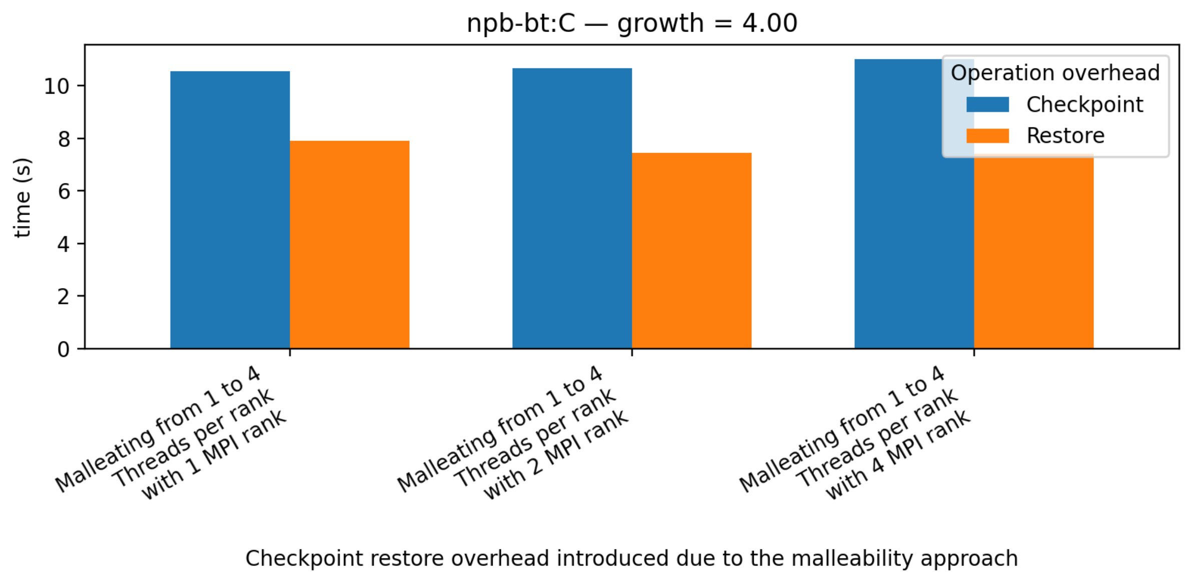

Figure 5.8: NAS BT-MZ experiments checkpoint and restore overhead

5.4 NAS LU-MZ:C Performance

5.4.1 NAS LU-MZ malleability with factor 0.25

The results from 5.9 show a similar trend compared to the NAS-BT benchmark, shown in Figure 5.1, in which the malleability approach introduces significant overhead when shrinking the application. Noteworthy is that, with an increasing number of processes, the number of ranks multiplied by the number of threads per rank, the application exhibits a similar execution time compared to the containerized environment with the same configuration, including post-malleability. The similarity suggests that the approach may offer performance benefits with sufficiently long job execution times.

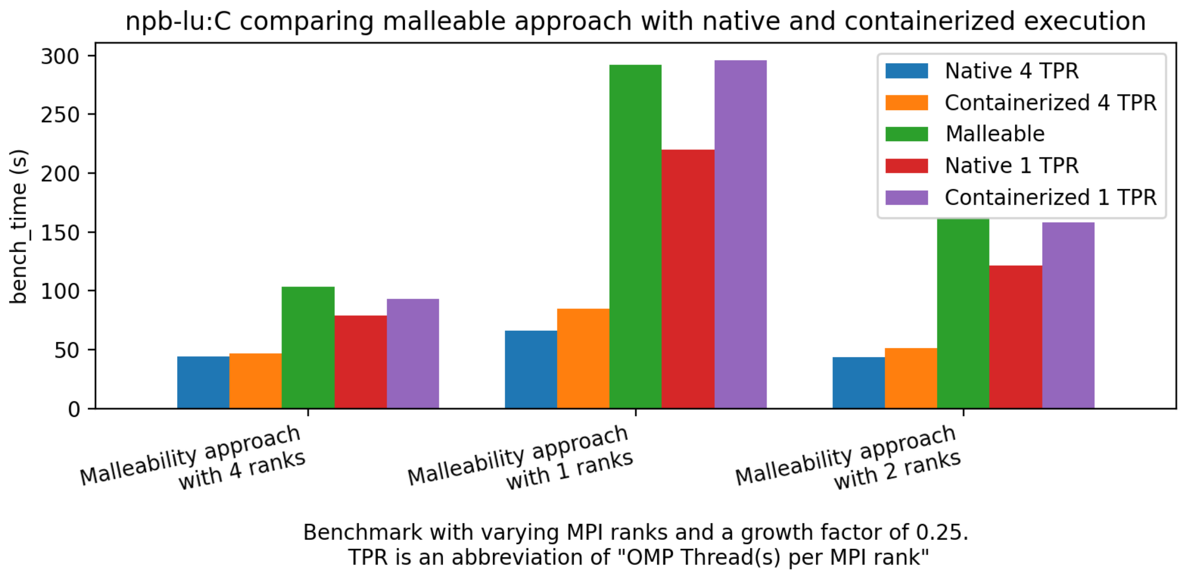

Figure 5.9: illustration of the performance results for the NAS LU-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 0.25, meaning that the number of OMP threads per MPI rank will be reduced by 75% halfway throughout execution.

This benchmark also does not reveal any anomalies regarding containerized environments having shorter execution times compared to native execution. Therefore, in this scenario, we can conclude that native execution is always faster than containerized execution, with a trend towards a higher number of total processes demanding less overhead than fewer processes.

There is no major difference when comparing Figure 5.2, Figure 5.4, Figure 5.6 and Figure 5.8 to the overhead introduced in the LU-MZ benchmark as shown in Figure 5.10. The overall overhead remains bounded, with more than 10 seconds per set of operations aligning with the overhead introduced in the BT-MZ application.

5.4.2 NAS LU-MZ malleability with factor 0.50

When comparing the results from Figure 5.9 with those from Figure 5.11, we observe an emerging trend suggesting that creating a checkpoint of an application using multiple MPI ranks introduces some overhead. With that in mind, the data show that using only 1 or 2 MPI ranks creates the best performance gain compared to containerized environments.

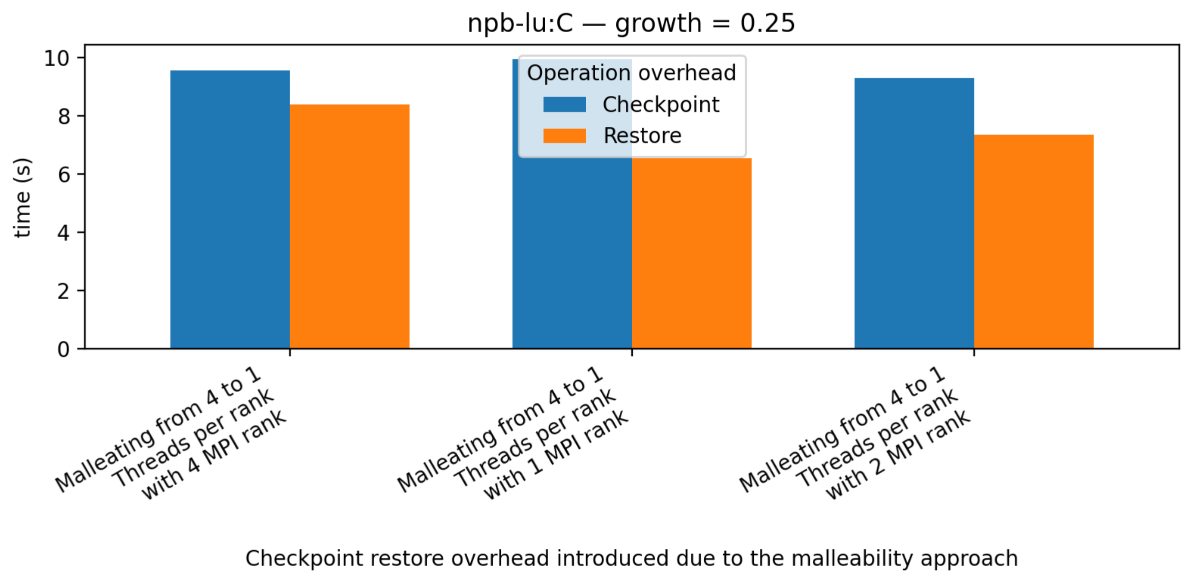

Figure 5.10: NAS LU-MZ experiments checkpoint and restore overhead

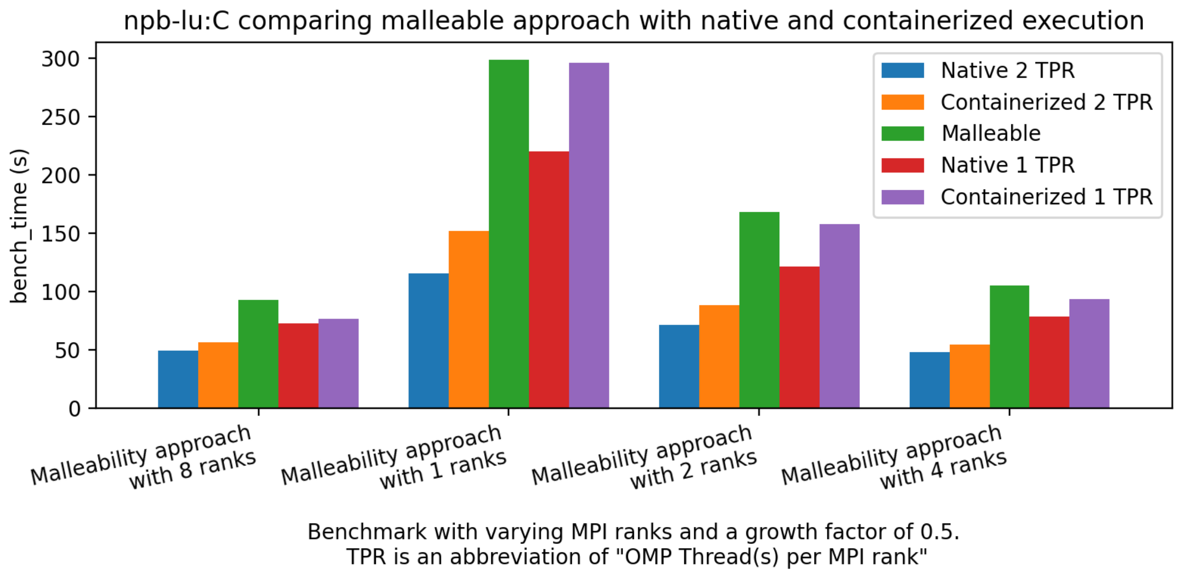

Figure 5.11: illustration of the performance results for the NAS LU-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 0.5, meaning that the number of OMP threads per MPI rank will be reduced by 50% halfway throughout execution.

Further analysis also suggests that more MPI ranks result in less initial overhead when running the application inside a containerized environment, implying that there might be an upper limit for which the malleability approach can be used to achieve something useful without unintentionally wasting resources. This upper limit may depend on several factors, including overall job execution time and whether the application has a significant memory dependency or primarily requires compute resources.

The overhead of the malleability approach, as shown in Figure 5.12, further reiterates the relation between increased overhead and the number of MPI ranks present in the original configuration before any malleability has occurred.

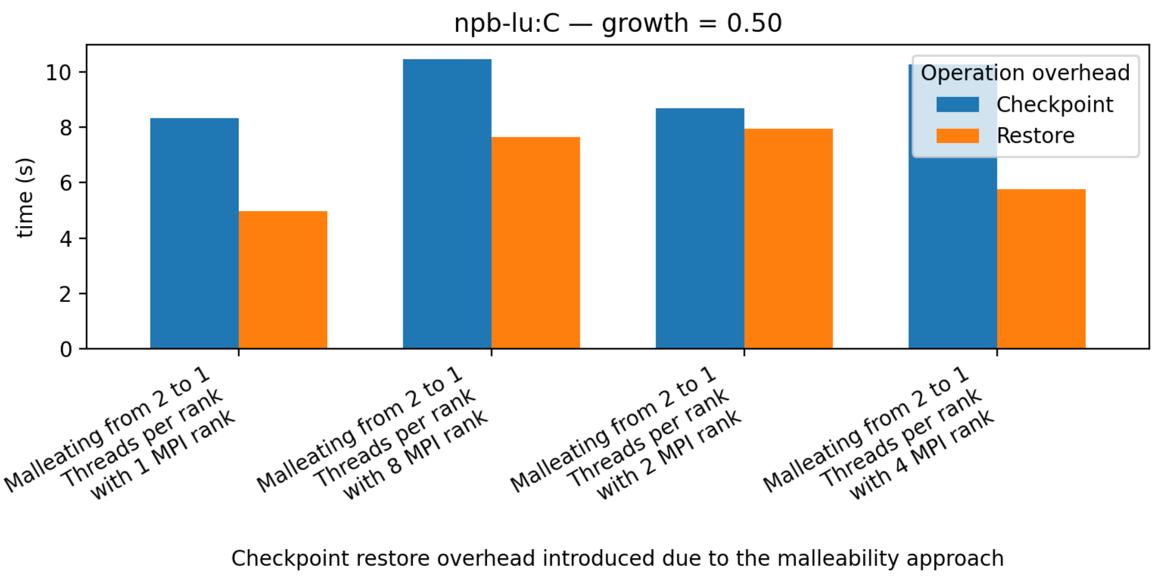

Figure 5.12: NAS LU-MZ experiments checkpoint and restore overhead

5.4.3 NAS LU-MZ malleability with factor 2.0

The data shown in Figure 5.13 indicates that the overhead introduced by containers depends on the total number of processes and becomes less significant as overall compute resources increase. It also shows that growing malleability towards the application can provide meaningful performance benefits, offering almost the same performance as the post-malleability configuration in containerized environments.

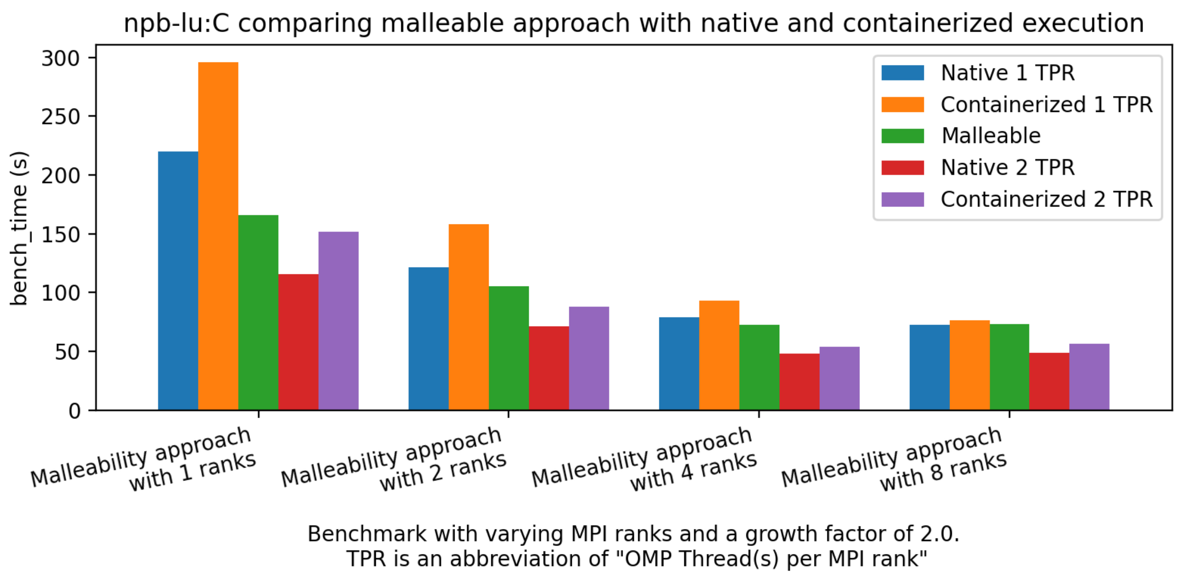

Figure 5.13: illustration of the performance results for the NAS LU-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 2.0, meaning that the number of OMP threads per MPI rank will be doubled halfway throughout execution.

It is also apparent that there is a meaningful performance benefit with the given malleability approach, specifically doubling the number of available threads halfway through execution, as the total execution time approaches the post-malleability benchmark time with an increasing number of MPI ranks.

The overall trend, in which overhead is proportional to the total number of processes, remains apparent

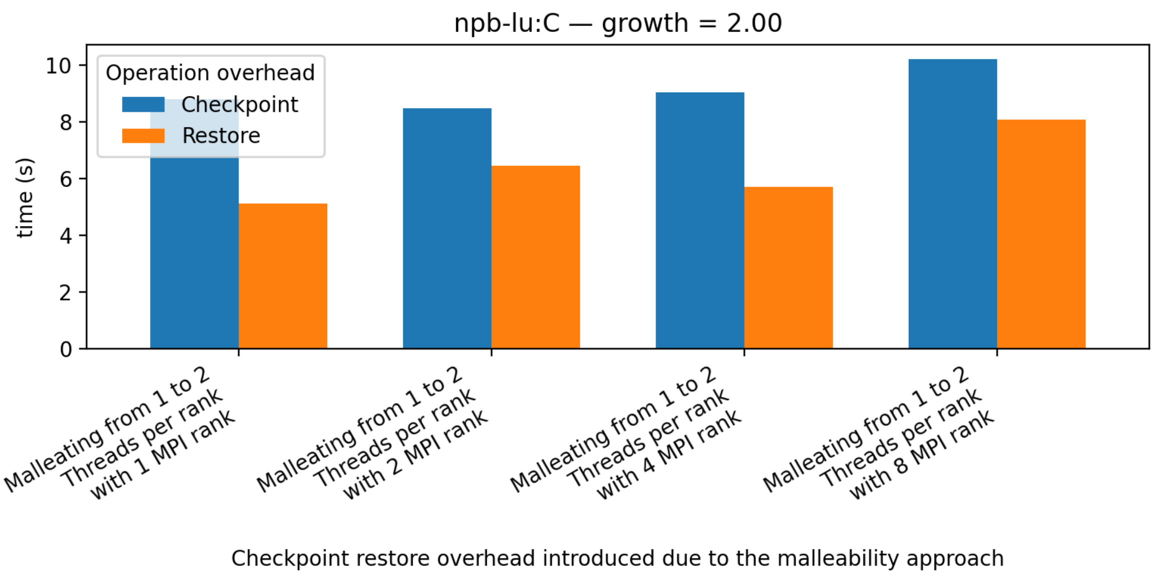

Figure 5.14: NAS LU-MZ experiments checkpoint and restore overhead

when comparing 5.12 and the other previously mentioned overhead graphs to Figure 5.37. It is noteworthy, however, that checkpointing and restoring 8 MPI ranks with 1 thread per rank appears to be faster than checkpointing and restoring 4 MPI ranks with 1 thread per rank. Although this phenomenon persists across many runs, this thesis does not investigate the underlying reasoning behind it.

5.4.4 NAS LU-MZ malleability with factor 4.0

With a growth factor of 4.0, as visualized in Figure 5.15, the malleability approach shows vast performance benefits compared to the benchmark being run with the initial configuration, especially under consideration of Figure 5.16 which shows that if the checkpoint and restore operations could be free that the overall execution time of the malleated run approaches the execution time of the containerized application. The data show a performance benefit of up to 50% when compared to the initial configuration, achieved by increasing the number of threads per rank four times.

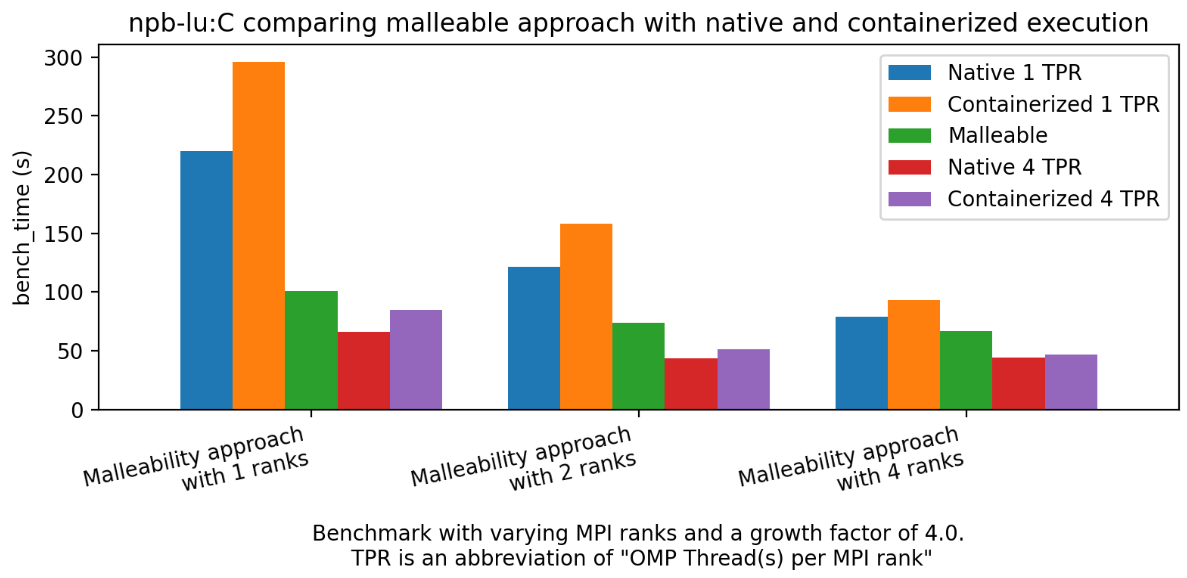

Figure 5.15: illustration of the performance results for the NAS LU-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 4.0, meaning that the number of OMP threads per MPI rank will be quadrupled halfway throughout execution.

This performance benefit diminishes when the overall number of MPI ranks increases. Diminishing returns are expected with the malleability approach, as the relative increase in threads per rank is constant, whereas the absolute increase is more pronounced for applications run with a smaller total number of processes.

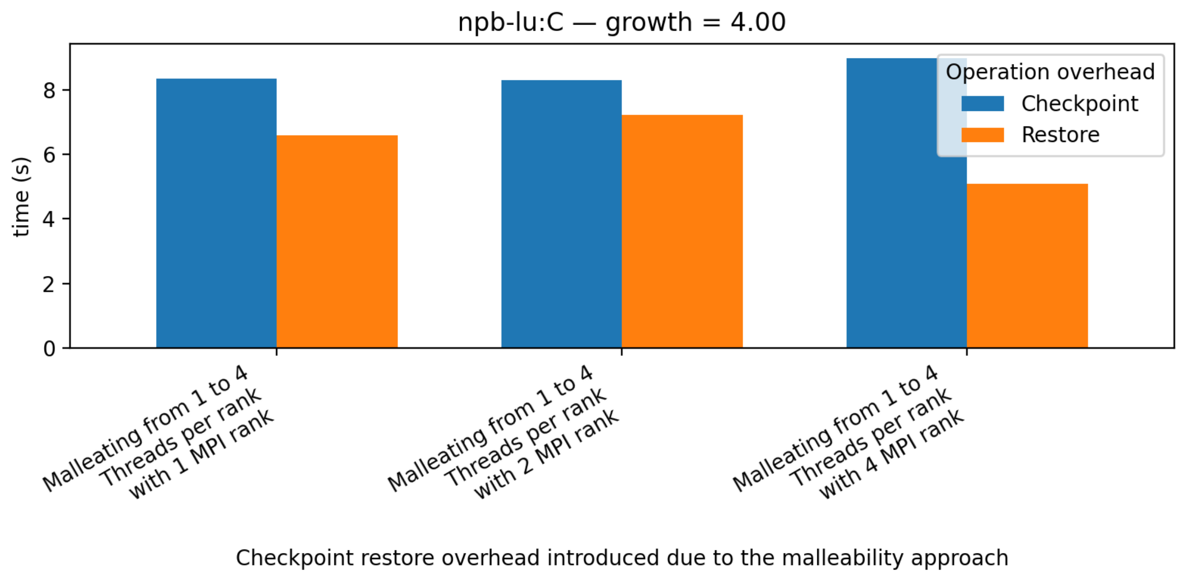

Figure 5.16: NAS LU-MZ experiments checkpoint and restore overhead

5.5 NAS SP-MZ Performance

5.5.1 NAS SP-MZ malleability with factor 0.25

The data shown in Figure 5.17 shows a similar trend compared to both Figure 5.1 and Figure 5.9 indicating that the performance deficit obtained due to introducing shrinking malleability almost negates the benefit of adaptability to resource changes. The deficit in total execution time is less pronounced than in the BT and LU benchmarks, which may be explained by the SP benchmark being less compute-bound than the BT and LU benchmarks.

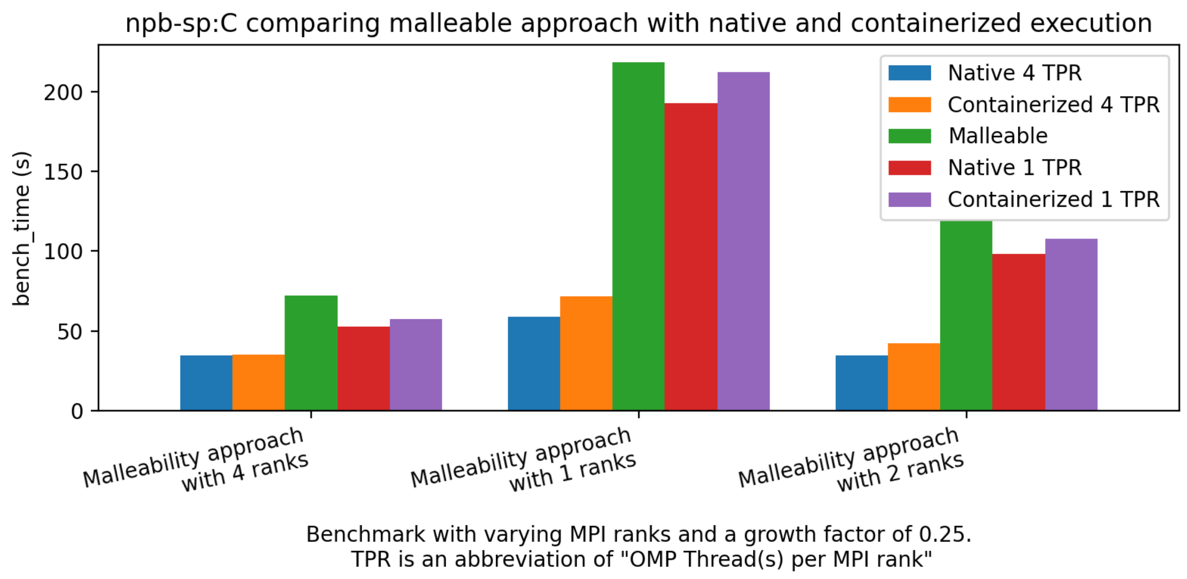

Figure 5.17: illustration of the performance results for the NAS SP-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 0.25, meaning that the number of OMP threads per MPI rank will be reduced by 75% halfway throughout execution

There is again an outlier in which the containerized environment has a shorter total execution time compared to the native environment when run with 4 MPI ranks and 4 OMP threads per rank. Given that this benchmark has less dependency on raw computing, it is suggested that this could be explained by containerized environments having stronger resource isolation compared to native applications, resulting in some performance benefits.

Figure 5.18 shows that the overall checkpoint and restore overhead is approximately the same as that of the BT and LU benchmarks, with less than 20 seconds of overhead for the execution of both operations.

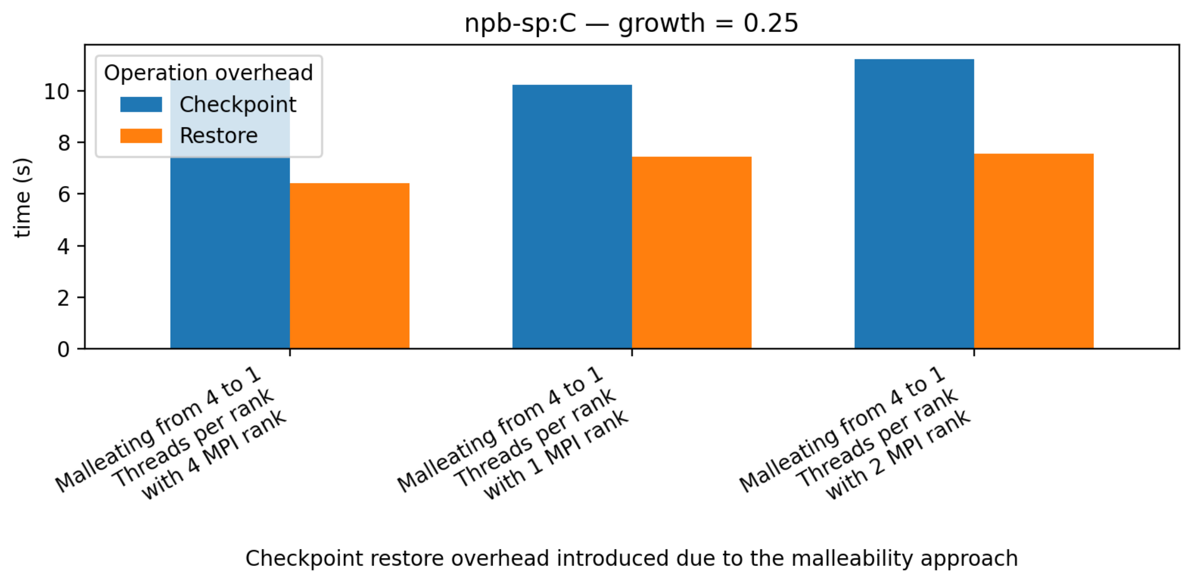

Figure 5.18: NAS SP-MZ experiments checkpoint and restore overhead

5.5.2 NAS SP-MZ malleability with factor 0.5

Following the same trend as shown in Figure 5.3 and Figure 5.11, does the SP benchmark have similar runtime behavior, shown in Figure 5.19. When adjusting the overall execution time using the malleability approach, as shown in Figure 5.20, it appears that the application is approaching a similar total execution time as if it were run with the post-configuration to begin with.

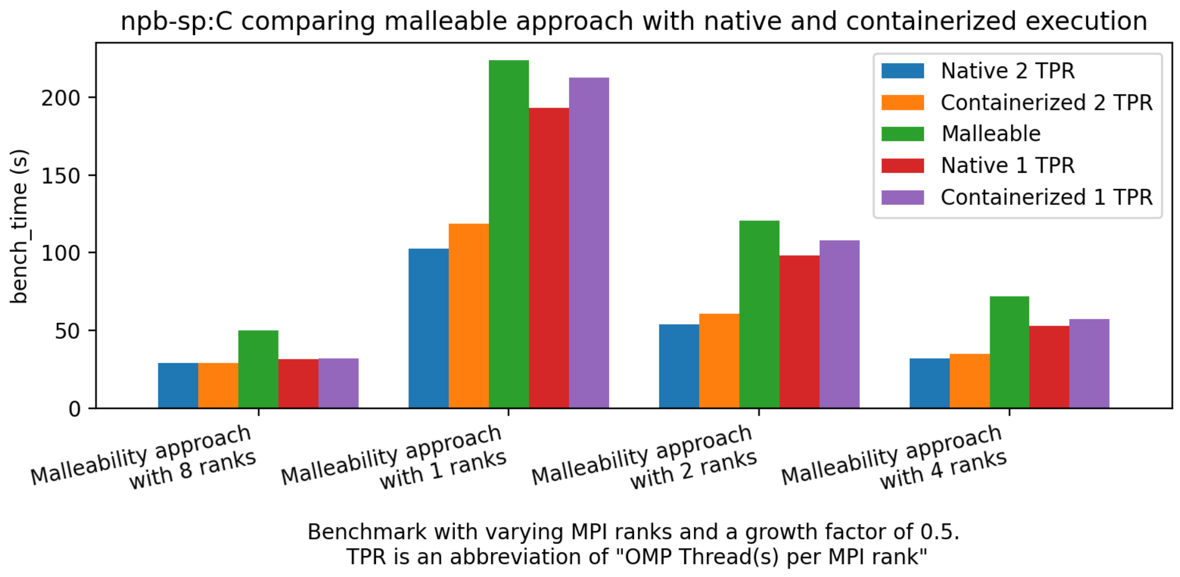

Figure 5.19: illustration of the performance results for the NAS SP-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 0.5, meaning that the number of OMP threads per MPI rank will be reduced by 50% halfway throughout execution.

When comparing the overall overhead introduced by the checkpoint and restore operations, as shown in Figure 5.20, does the data shown in Figure 5.19 suggest that execution time could almost approximate containerized execution with minimal overhead when halving the total number of threads halfway through the run.

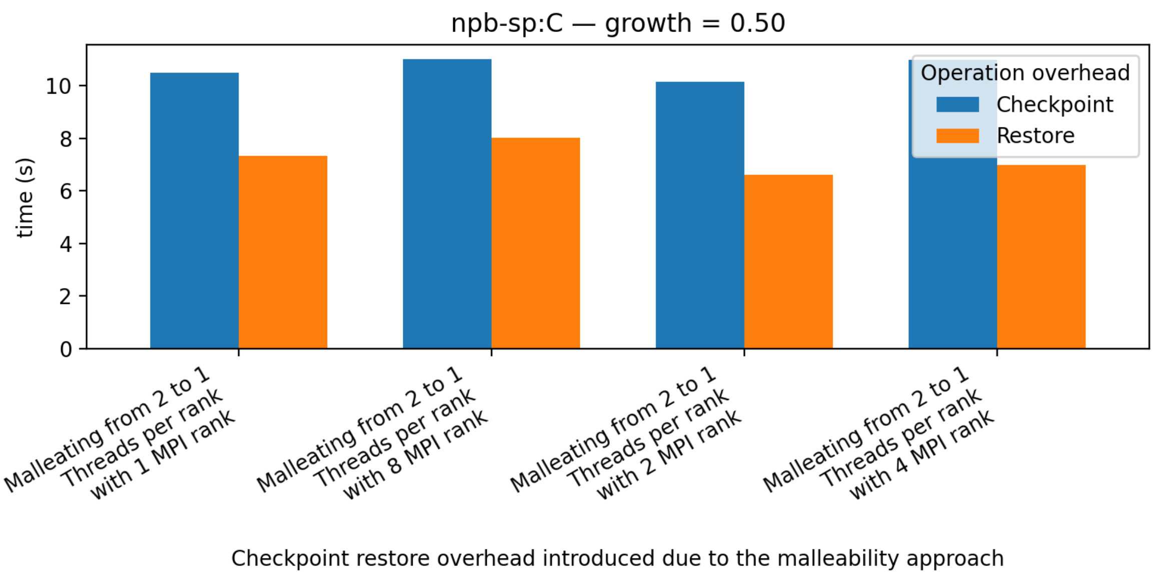

Figure 5.20: NAS SP-MZ experiments checkpoint and restore overhead

5.5.3 NAS SP-MZ malleability with factor 2.0

The data shown in Figure 5.21 follows a similar trend compared to Figure 5.5 and Figure 5.29 indicating that applications with many parallel regions respond to environment adjustments and scale accordingly. The approach generally still demonstrates the potential to increase overall resources, allowing the total execution time to exceed native execution time.

The total runtime of the execution has to be considered since the overhead stays rather constant, as shown in Figure 5.22. However, the execution time decreases, suggesting that there is a point of execution in which the malleability approach would harm the total execution time regardless of growth or shrinkage.

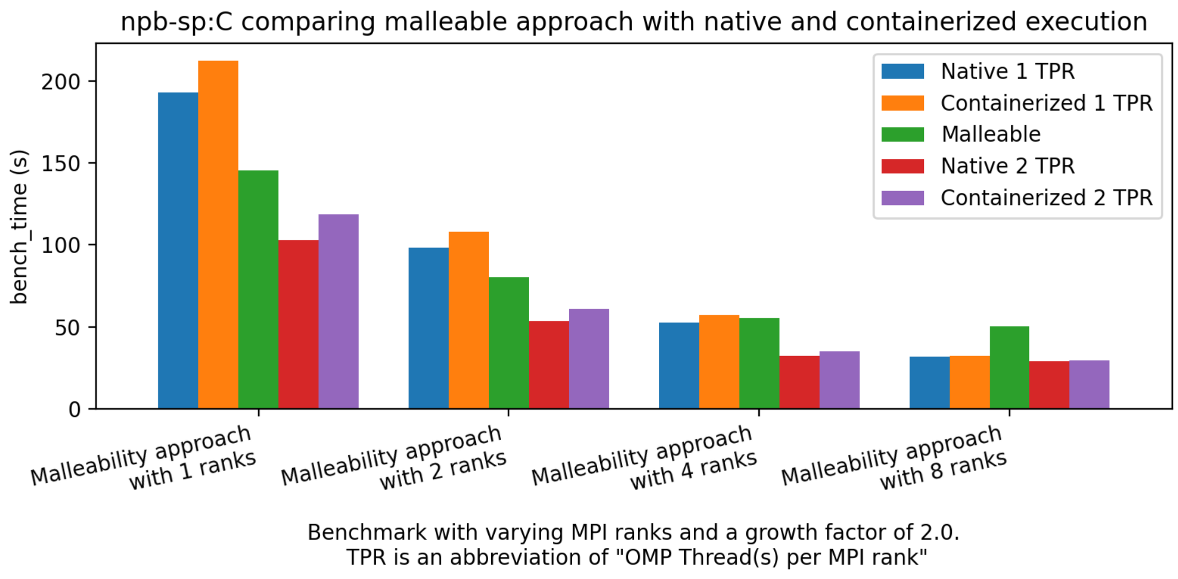

Figure 5.21: illustration of the performance results for the NAS SP-MZ benchmark when employing the malleability approach with a constant number of MPI ranks at a growth factor of 2.0, meaning that the number of OMP threads per MPI rank will be doubled halfway throughout execution.

The total overhead introduced by the checkpoint and restore operations, as shown in Figure 5.22, suggests that even under the assumption that checkpoint and restore are operations with no associated cost, would still not be as fast as running within the containerized environment to begin with.

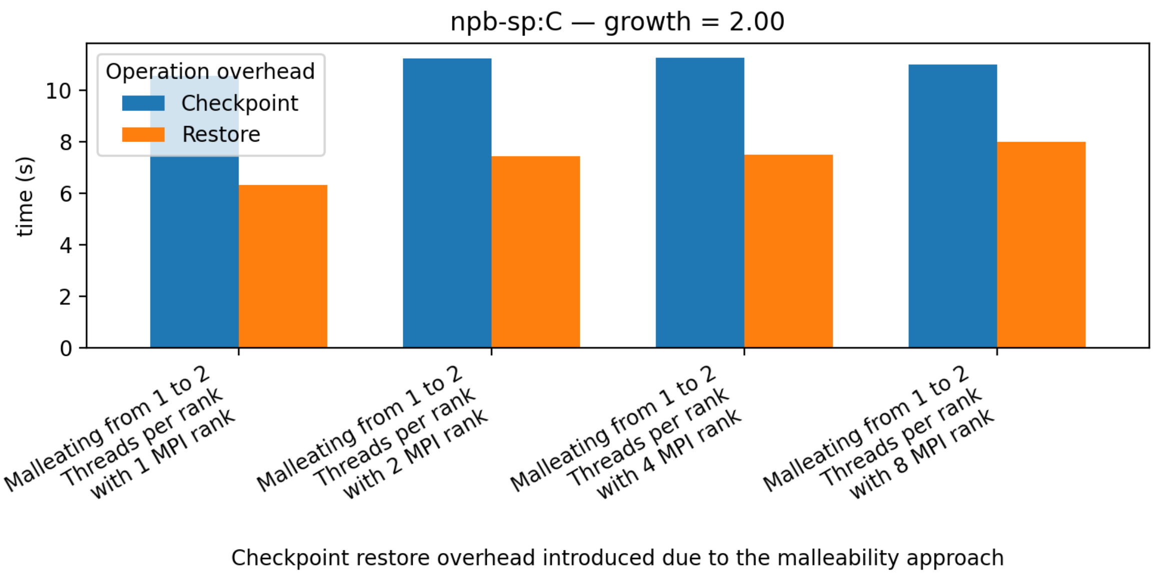

Figure 5.22: NAS SP-MZ experiments checkpoint and restore overhead

5.5.4 NAS SP-MZ malleability with factor 4.0

The data shown in Figure 5.23 suggests a similar trend as Figure 5.22 such that the proposed malleability approach reaches a point at which malleability does not offer any meaningful benefit due to the introduced overhead, as shown in Figure 5.24, as the introduced overhead stays relatively constant, but grows slightly with the number of MPI ranks. The total execution time would have to be long enough for this overhead to become negligible.

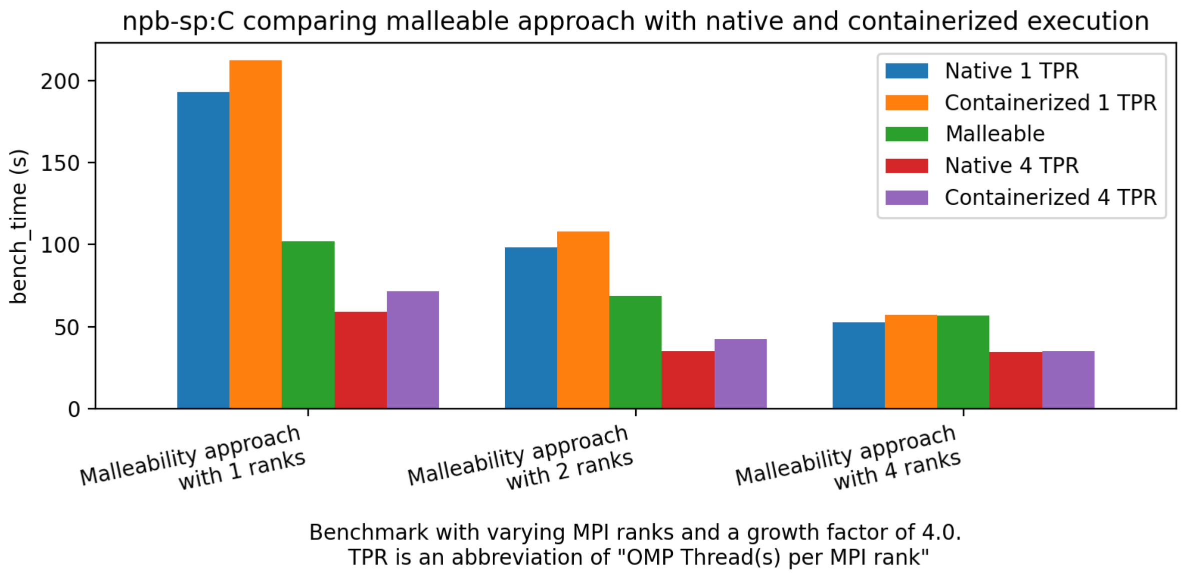

Figure 5.23: NAS SP-MZ Class C performance when using the malleability approach using physical core remapping

The overall overhead of the checkpoint and restore overhead introduced in the SP benchmark follows a linear trend that grows with the number of MPI ranks, as shown in Figure 5.18, Figure 5.20, Figure 5.22 and Figure 5.24.

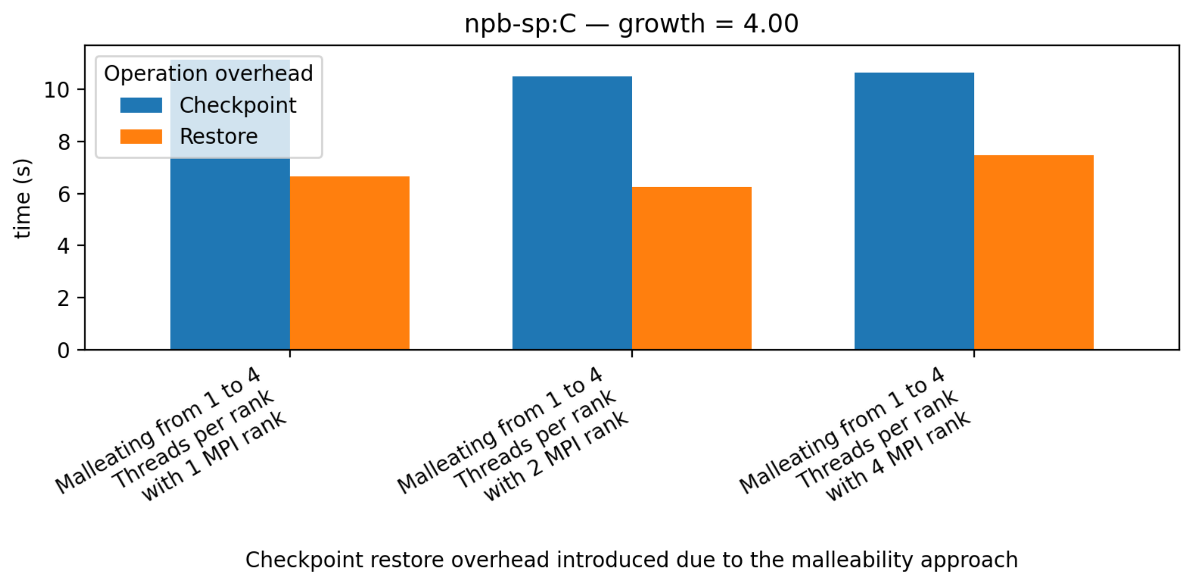

Figure 5.24: NAS SP-MZ experiments checkpoint and restore overhead

5.6 LLNL/LULESH Performance

5.6.1 LULESH malleability with factor 0.25

Figure 5.25 illustrates the performance results of the LULESH benchmark when employing the malleability approach with a growth factor of 0.25, meaning that the number of OMP threads per MPI rank will be reduced by 75% halfway throughout execution.

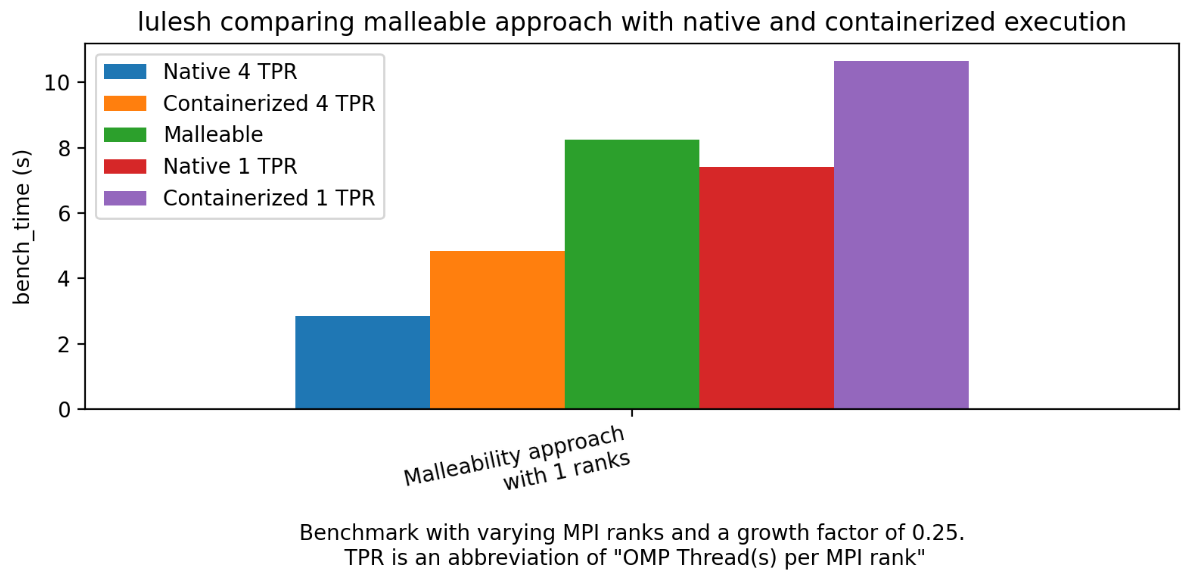

Figure 5.25: LLNL/LULESH experiments when scaling from 4 OMP thread per MPI rank to 1 OMP thread per MPI rank

The data shown in Figure 5.25 indicates that performance can be relatively expected when compared to containerized versions of the benchmark, with configurations in line with the previously demonstrated malleability of OMP threads per MPI rank.

It is noteworthy that the LULESH benchmark can only be run with either 1 or 8 MPI ranks, as the benchmark requires the total number of MPI ranks to be a cubic number. Moreover, since the system this paper relies on has at most 12 cores and 24 threads, as shown in Figure 4.2, only the configurations with 1 and 8 MPI ranks are evaluated. Further analysis of systems with more available cores should be conducted to verify the general trend observed in this specific benchmark.

The data shown in Figure 5.25 shows a different trend compared to Figure 5.1, Figure 5.17 or Figure 5.9 in which the overhead of having to checkpoint and restore majorly affects overall execution time. For LULESH, in particular, the checkpoint and restore overhead does not dominate applications with a lower number of processes.

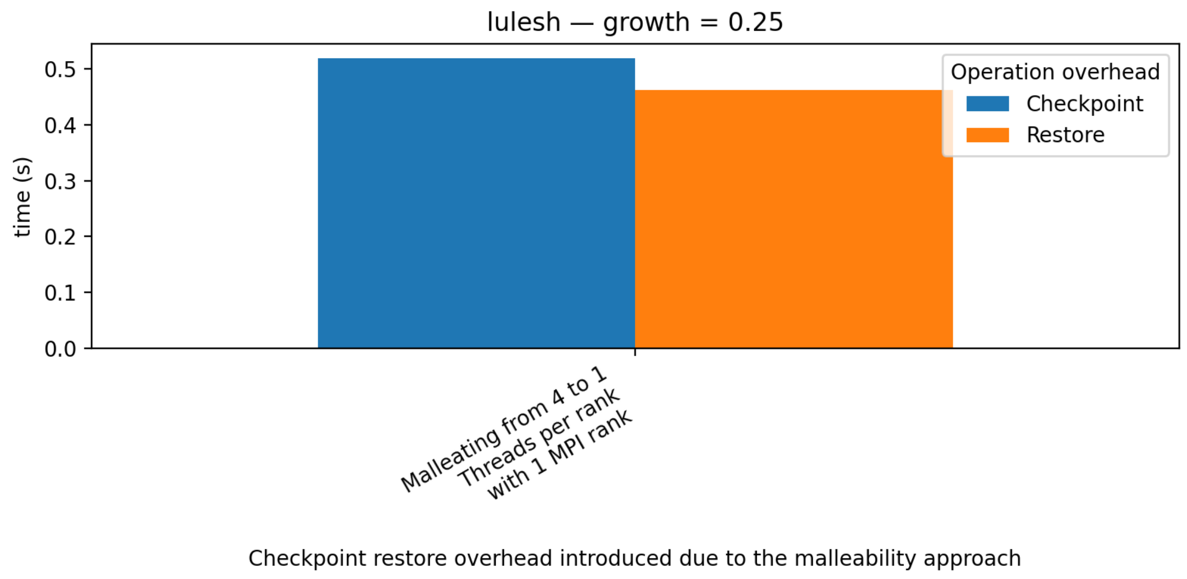

Figure 5.26: LLNL/LULESH experiments checkpoint restore overhead

Figure 5.26 illustrates the overhead incurred by checkpointing and restoring the application to adjust the available resources dynamically.
When comparing the overhead introduced in the NAS benchmark suite, by having to checkpoint and restore, shown in Figure 5.2, Figure 5.10 and Figure 5.18 to Figure 5.26 we observe that the penalty for the LULESH benchmark is effectively negligible in contrast as it only takes slightly over one second instead of more than ten seconds.

5.6.2 LULESH malleability with factor 0.5

Figure 5.27 illustrates the performance results of the LULESH benchmark when employing the malleability approach with a growth factor of 0.50, meaning that the number of OMP threads per MPI rank will be reduced by 50% halfway throughout execution.

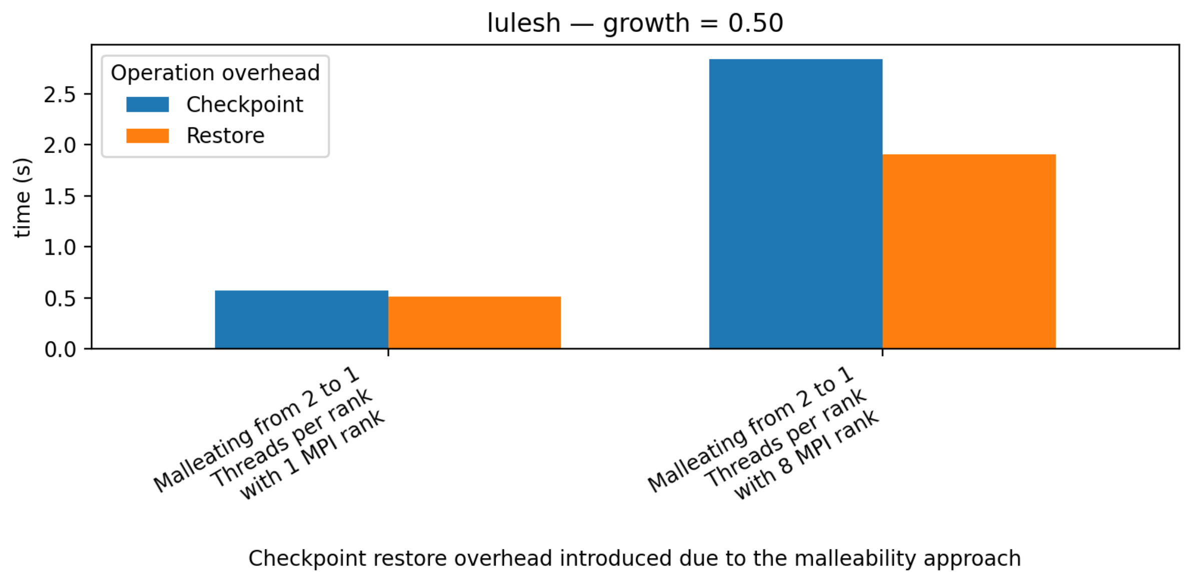

Figure 5.28: Performance overhead introduced by checkpoint and restore operations within the LULESH benchmark when halving the number of threads available

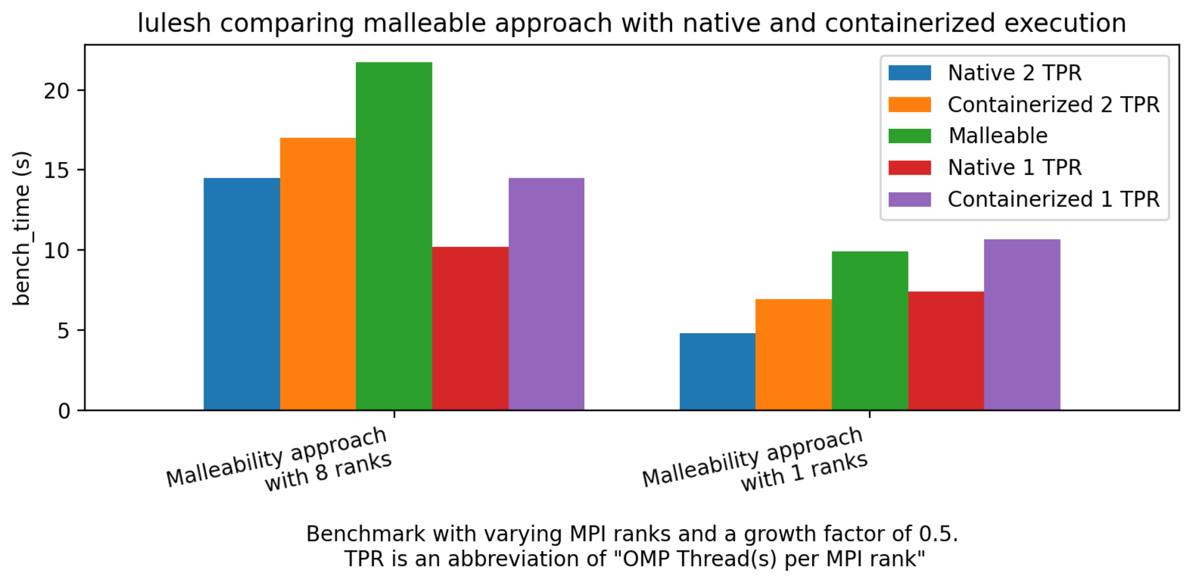

Figure 5.27: LLNL/LULESH experiments when scaling from 2 OMP thread per MPI rank to 1 OMP thread per MPI rank

The data shown in Figure 5.27 shows an interesting trend in which the performance loss can not be explained by heavy overhead due to checkpoint and restore functionality as shown in Figure 5.28 as the total overhead introduced, by having to checkpoint and restore, does not grow beyond 5 seconds. Figure 5.28, however, shows that the checkpoint and restore overhead grows with the number of MPI ranks.

5.6.3 LULESH malleability with factor 2.0

The data shown in Figure 5.29 shows the potential performance loss introduced by the malleability approach. Similar to Figure 5.28, the overhead of the checkpoint and restore operation, as shown in Figure 5.30, does not explain the apparent performance loss. The overhead introduced by the checkpoint and restore operations, however, does explain the apparent

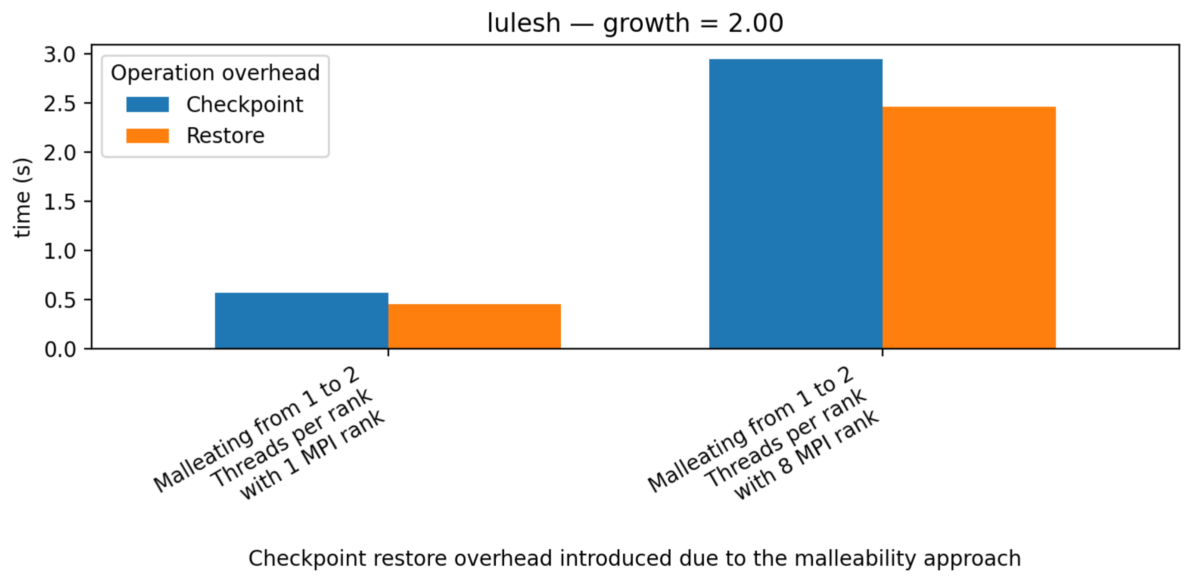

Figure 5.30: Performance overhead introduced by checkpoint and restore operations within the LULESH benchmark when doubling the number of threads

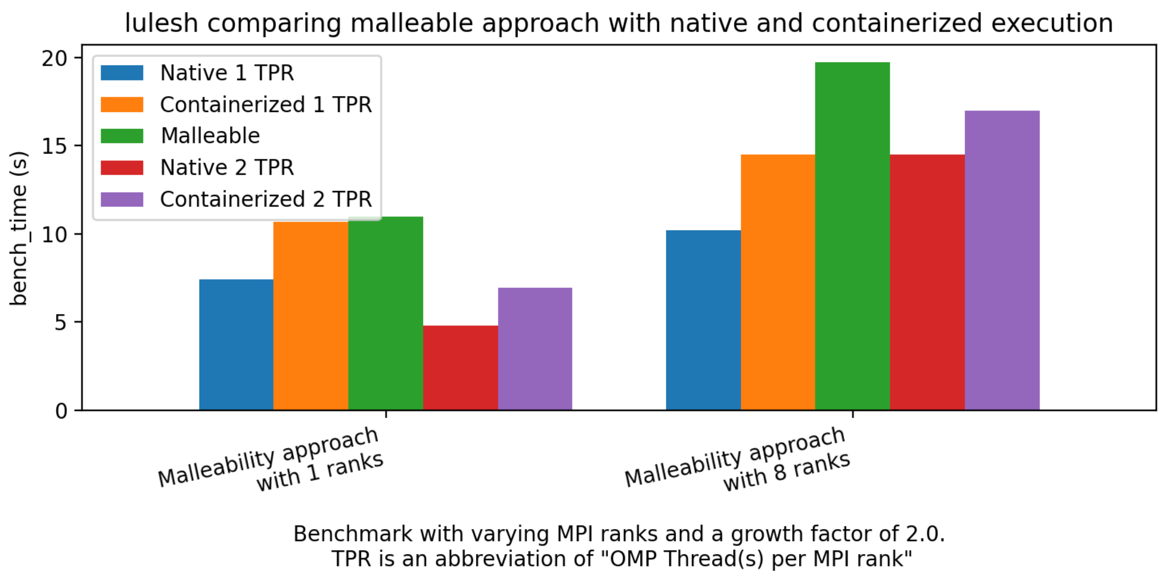

Figure 5.29: LLNL/LULESH experiments when scaling from 1 OMP thread per MPI rank to 2 OMP thread per MPI rank

5.6.4 LULESH malleability with factor 4.0

The data shown in Figure 5.31, combined with the information on the introduced overhead shown in Figure 5.32, indicates that the overall premise of growth within applications can lead to performance benefits, but may also yield similar results when compared to the original containerized configuration.

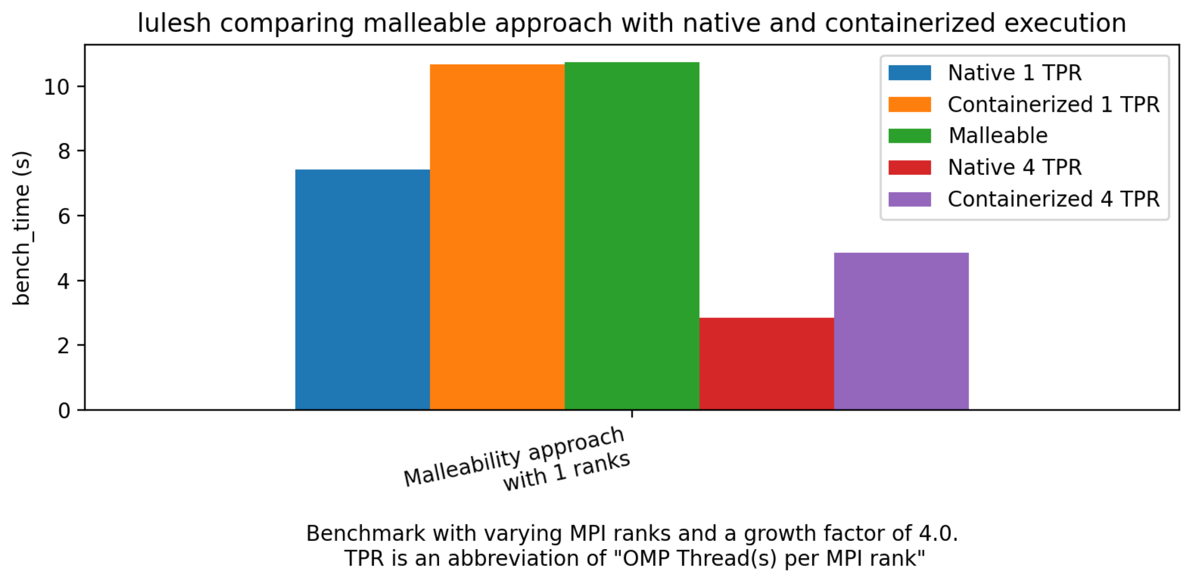

Figure 5.31: LLNL/LULESH experiments when scaling from 1 OMP thread per MPI rank to 4 OMP thread per MPI rank

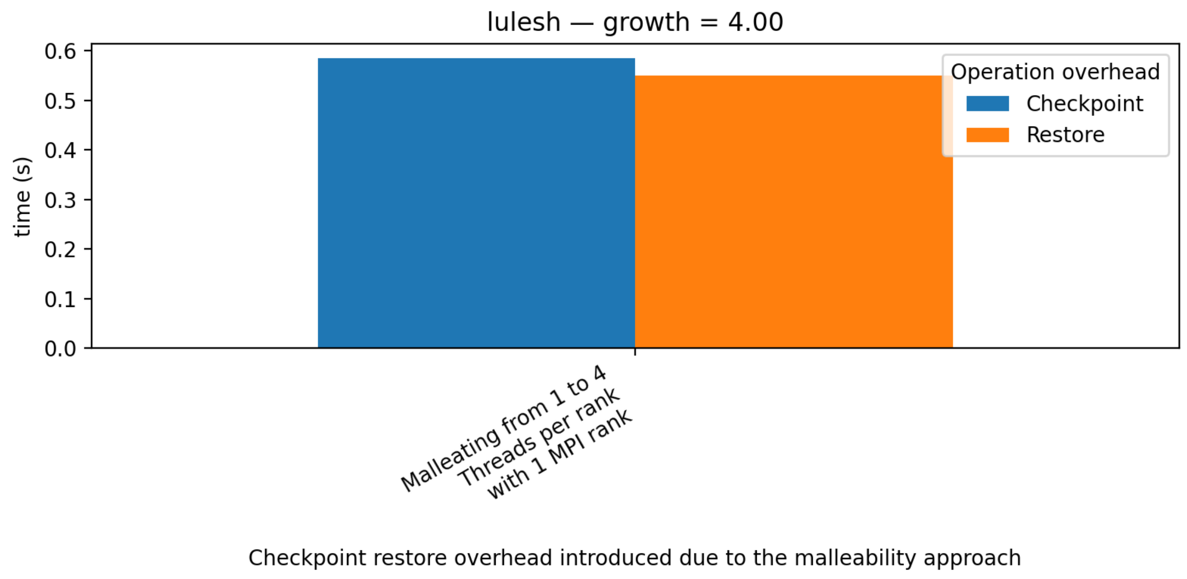

Figure 5.32: Performance overhead introduced by checkpoint and restore operations within the LULESH benchmark when doubling the number of threads

Further analysis of all obtained benchmark data, in conjunction with their respective overhead data, suggests that as the average execution time increases by 1%, the LULESH benchmark does not appear to respond to changes in the available resources within the environment. Several reasons could result in this outcome, the most prominent being that the LULESH benchmark has a singular parallel region, which limits its ability to respond to OMP thread pool adjustments. Another potential reason is that the benchmark itself primarily involves executing operations consecutively, meaning that no operation is truly parallel. The exact reason for this occurrence has not been investigated in this paper.

5.7 LLNL/AMG Performance

5.7.1 AMG malleability with factor 0.25

The Figure 5.33 shows an interesting result as the requirement to checkpoint to disk and restore from said disk introduces major overhead. This can be easily explained by examining 5.34, which shows that the overall overhead of the checkpoint and restore operation exceeds 100 seconds in any case. The reason for the overhead being so large is due to the benchmark's reliance on RAM, which requires about 30GB for the benchmark cases. Another reason for the introduced overhead is evident when comparing this to Figure 4.2, which shows that the system has a total of 32GB available, indicating that there is no space available to store the checkpoint file in RAM. Assuming that the overhead introduced by having to store the process state on disk does not exist, the data suggests an additional overhead of about 50% exists when purely subtracting the checkpoint and restore overhead compared to running the application immediately with the post-malleability configuration.

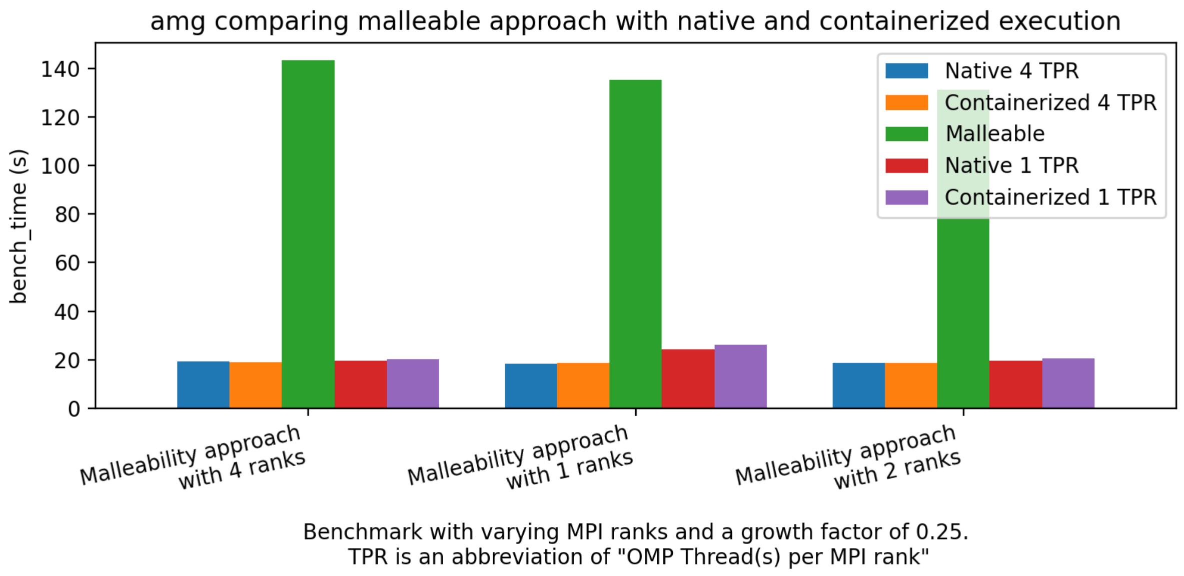

Figure 5.33: LLNL/AMG performance when using the malleability approach using physical core remapping

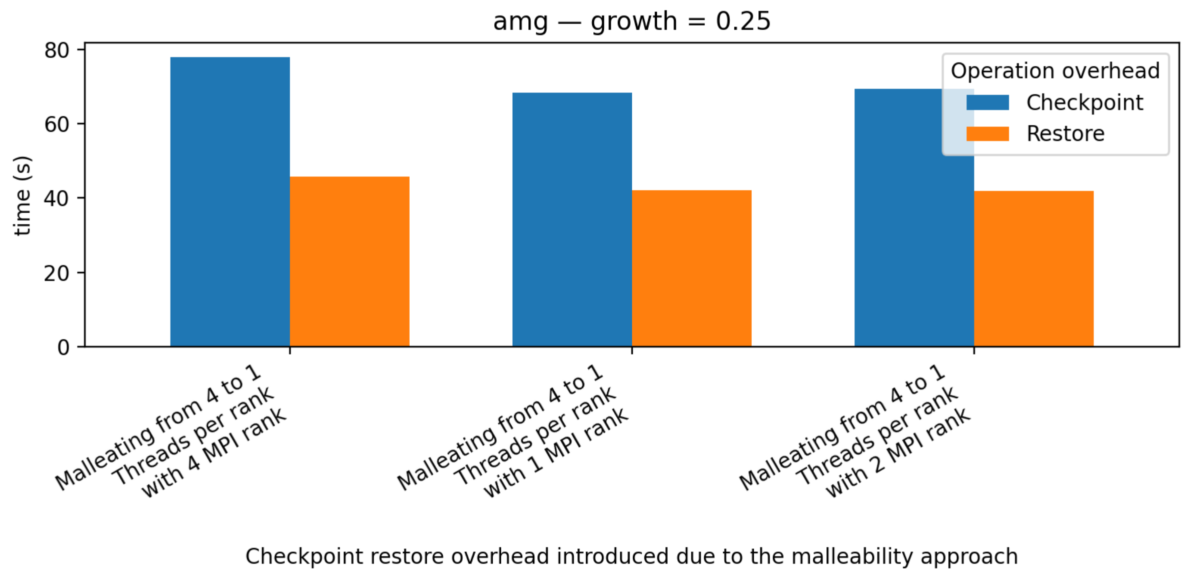

Figure 5.34: Performance overhead introduced by checkpoint and restore operations within the AMG benchmark when doubling the number of threads

5.7.2 AMG malleability with factor 0.5

Figure 5.35 illustrates another flaw in the proposed malleability approach, particularly for applications that run close to the available memory limit. Specific configurations of the benchmark fail to resume entirely due to memory mapping issues (*Segmentation fault: address not mapped to object at address*). This problem is not directly addressed in this thesis, but it can be overcome if more RAM is available. This problem can also be observed in Figure 5.36, where the entry is missing for the benchmark that failed to resume.

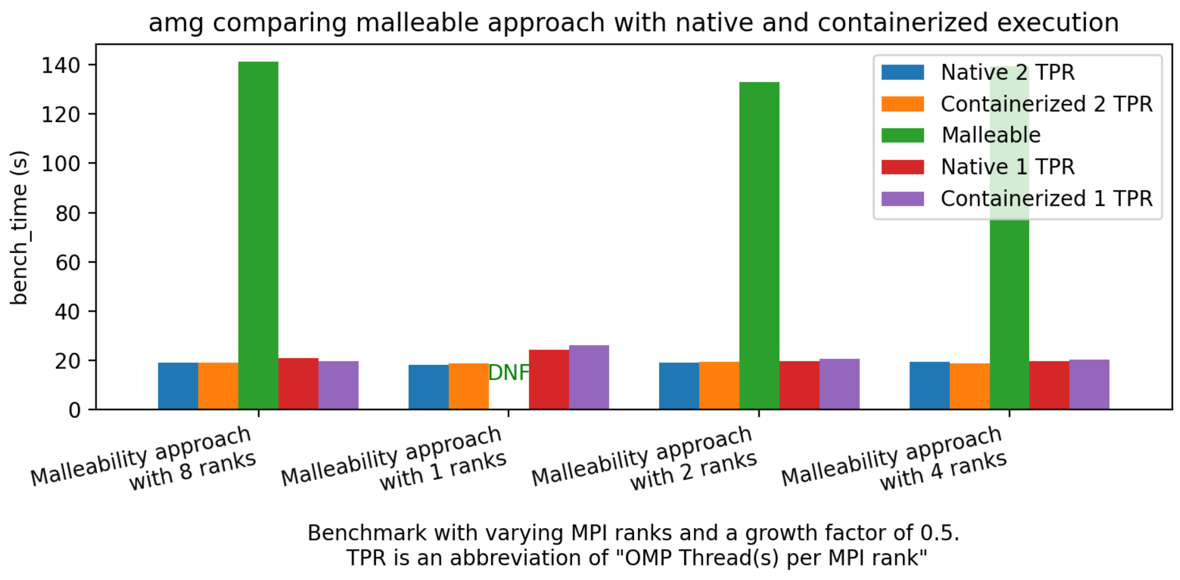

Figure 5.35: LLNL/AMG performance when using the malleability approach using physical core remapping

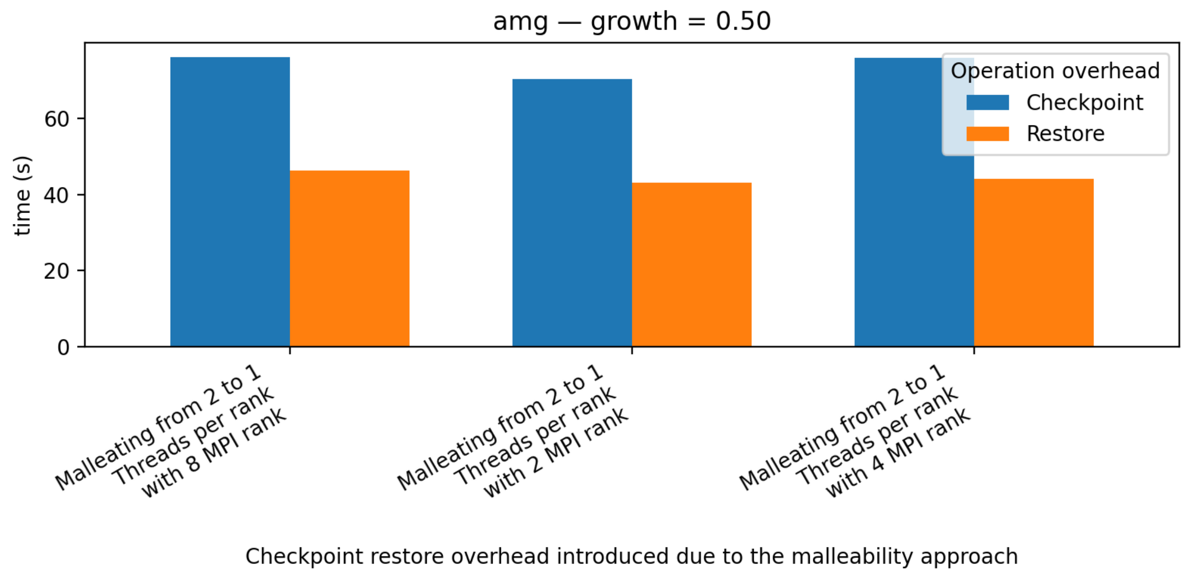

Figure 5.36: Performance overhead introduced by checkpoint and restore operations within the AMG benchmark when doubling the number of threads

5.7.3 AMG malleability with factor 2.0

A similar limitation is visible in Figure 5.37, which shows that the benchmark is running out of RAM, particularly with configurations that host a small number of MPI ranks. The reason why execution runs of the benchmark with a higher number of MPI ranks is not determined within the scope of this work. There is however, a similar trend in total execution time compared to Figure 5.35 and Figure 5.33 in which the total introduced overhead exceeds the normal runtime of the application, as shown in Figure 5.38, in any configuration multiple times, further reinforcing careful consideration of applications that are close to the available memory limit of the system.

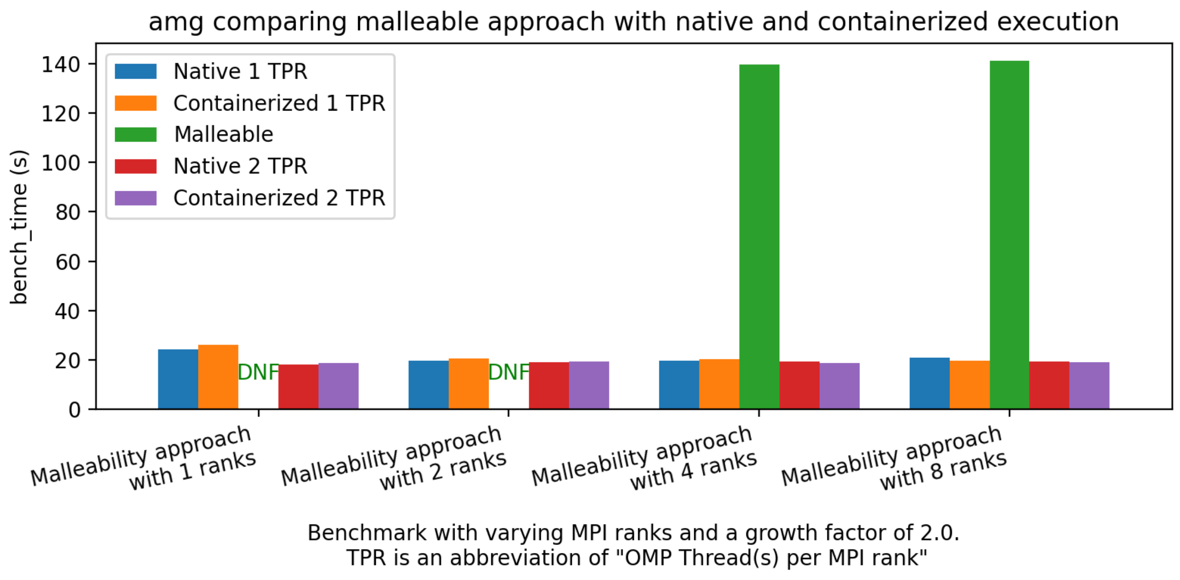

Figure 5.37: LLNL/AMG performance when using the malleability approach using physical core remapping

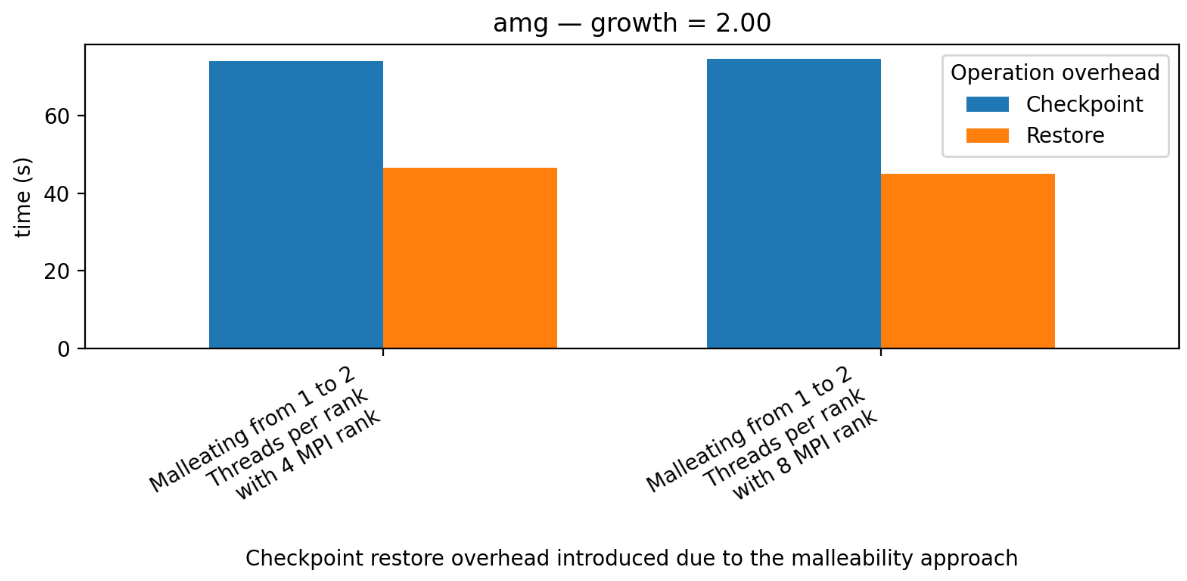

Figure 5.38: Performance overhead introduced by checkpoint and restore operations within the AMG benchmark when doubling the number of threads

5.7.4 AMG malleability with factor 4.0

Similarly, as Figure 5.37 does, the trend continues where the benchmark with a lower number of MPI ranks fails to resume. Overall, data comparison of the existing data shows that pure containerization does not introduce a major overhead compared to native execution, suggesting that future work can address the limitations mentioned in this section.

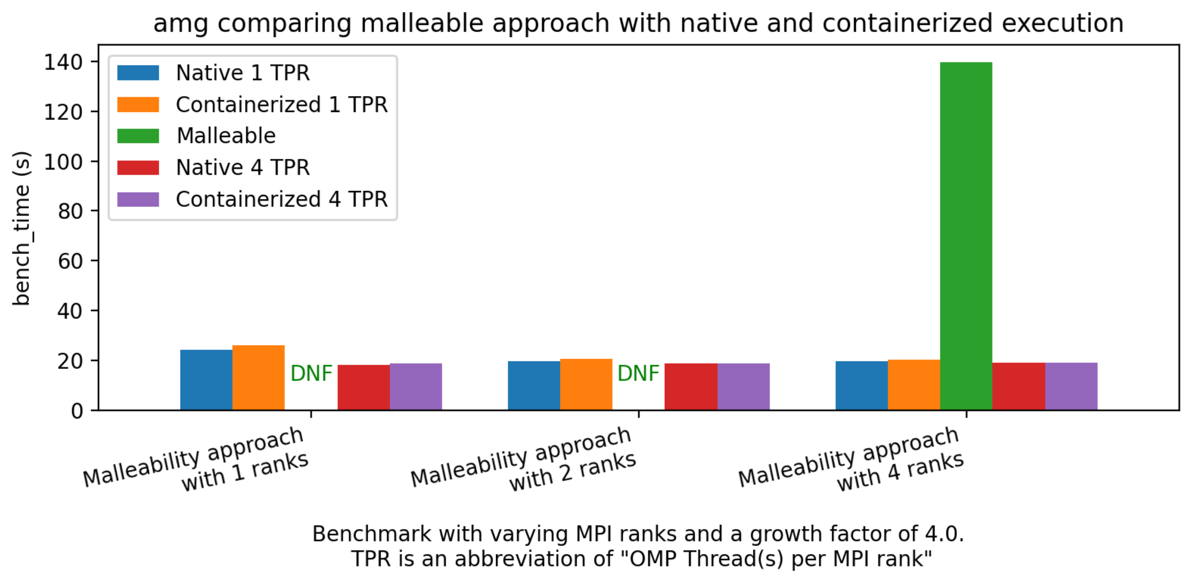

Figure 5.39: LLNL/AMG performance when using the malleability approach using physical core remapping

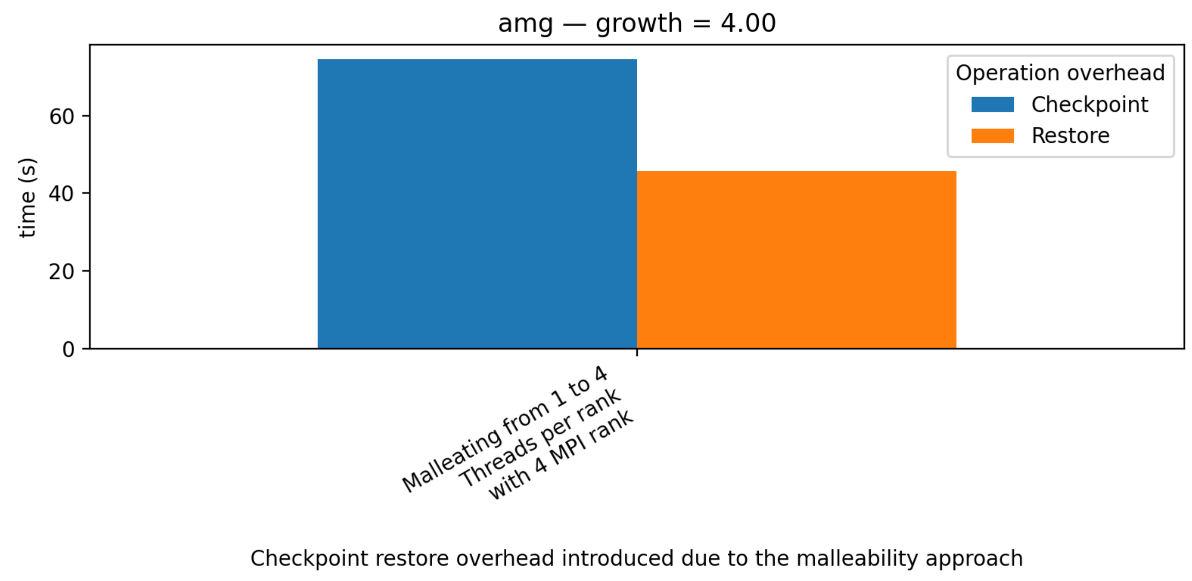

Figure 5.40: Performance overhead introduced by checkpoint and restore operations within the AMG benchmark when doubling the number of threads

Chapter 6

Discussion

6.1 Interpretation and Significance of the Results

The experiments reported in Chapter 5 confirm that our malleability framework, in a containerized environment, is both practical and beneficial for existing HPC applications, provided that two conditions are met. The code should contain multiple OpenMP parallel regions that can adapt to changes in the number of available cores, and its memory footprint should be small enough to fit within the available memory. When satisfied, checkpoint and restore mechanisms are applicable. This paper concludes that dynamically increasing OpenMP thread pool sizes at runtime provides performance gains, validating the thesis hypothesis that existing tools running in user space (in our case, Docker and CRIU) can deliver meaningful malleability toward existing applications.

Providing this capability demonstrates that existing applications can be run within a new environment to achieve malleable behavior simply by utilizing the work provided in this thesis, eliminating the need for code adjustments or kernel level tools. It also shows that HPC environments with meaningful job fragmentation or over-estimation of job resources could benefit from malleable containers to absorb freed resources and increase throughput without job restarts or manual reconfiguration. Ultimately, the success demonstrated in this paper suggests that malleability attempts for OMP scaling offer a possible path toward greater flexibility in containerized HPC environments.

Second, the observed performance improvements, particularly when scaling up from constrained configurations, highlight the potential of malleability to improve the overall system throughput. In environments where job fragmentation and idle cores are common, malleable applications could dynamically utilize the unused resources and improve throughput without the need for job resubmission or manual reconfiguration.

And finally, the successful demonstration of checkpoint and restore operations along with resource adaptation of parallel applications within containers contributes to the emerging discussion on containerized HPC applications and environments, where flexibility, portability, and adaptability are increasingly important.

6.2 Comparison with Related Work

Previous research, such as FlexMPI Martín et al. [2013], focuses entirely on MPI malleability by dynamically resizing ranks, but it requires modifications to the source code of their developed interface. In contrast to the work of FlexMPI, the work presented in this paper provides a source-independent solution for malleability in workloads that meet the previously mentioned conditions (multiple parallel regions and a smaller memory footprint).

Our results also extend the literature on container performance in HPC. Previous work has demonstrated nearly native performance for MPI applications when using Docker or Singularity, provided that they are correctly tuned. We further enhance this by demonstrating that the same container environment can host malleable workloads and checkpoints and restores parallel applications in seconds with an impact on overall job duration.

6.3 Remaining Challenges and Limitations

Apart from the already mentioned lack of MPI malleability, other challenges and limitations exist. Dependencies introduced by CRIU require specific kernel versions. Upgrading the underlying operating system may resolve these issues, but it is not always practical in already established HPC environments that are running older versions. This paper has also shown that, depending on the scale of the application, downtime is incurred whenever a malleability change is performed. Depending on the length of the job, a downtime of a few seconds is acceptable; however, this paper also shows that it may increase to several times the length of the original application.

Chapter 7

Conclusion

7.1 Main contribution

This work demonstrates that malleability in HPC workloads can be effectively achieved using containerized environments without requiring privileged access or modifications to the application source code when only the binary is available. This work builds on a contained solution with dynamic resource adjustment techniques, along with checkpoint and restore functionality, as a practical approach to adapting running HPC applications towards malleability.

The primary contribution is the evaluation and implementation of a malleable execution strategy that operates entirely in user space, relying almost entirely on already existing technologies.

This thesis shows that dynamic runtime resource scaling can offer benefits when introduced in the correct environment. Within areas of memory dependence, there does not appear to be much benefit; however, this conclusion cannot be reasonably drawn due to the short runtime of the overall benchmarks. It may have a constant effect on the overall runtime, leading to potential performance benefits.

7.2 Future work

While the outcomes of this study demonstrate the viability and utility of malleability on a thread level, several important research directions remain open. Notably, full MPI malleability was not achieved within the scope of this work. Attempts using tools such as FlexMPI were found to be impractical without significant integration and adjustments on a source level. Future research could focus on enabling dynamic resizing of MPI ranks within mainstream libraries or on developing hybrid execution models that combine static MPI allocations with dynamic threading more fluidly.

The performance implications of running an application in a distributed environment were also neglected in this paper, requiring more careful consideration and potential adjustments on the software side to prevent the overhead from growing out of proportion.

An area that has not been explored in this work is malleability support for workloads that use GPUs or other hardware-accelerating devices.

Checkpoint and restore operations, while feasible, remain dependent on kernel support and CRIU compatibility. Therefore, exploring alternative approaches to enable malleability without relying on full checkpoint/restore cycles represents an important direction for future research.

Moreover, while this paper focused on manual adjustments during execution, true malleability integrated into HPC environments would require integration with job schedulers (SLURM or other batch schedulers). Future work could expand on this framework by providing support for dynamic resource adjustments based on system load, job-specific variables, or policy constraints.

Finally, although this thesis emphasized performance, malleability techniques may also offer benefits in fault-tolerant computing. The ability to checkpoint and restore applications dynamically could serve as an entry towards execution models with a focus on recovery, particularly in elastic and HPC platforms where failure rates and volatility are higher.

In conclusion, this thesis demonstrates that malleability within a containerized environment is both achievable and can be used for further research toward more adaptive execution models in HPC, laying the foundation for future work to enhance flexibility, resilience, and efficiency in computational science workflows.

Bibliography

Mpich is a high performance and widely portable implementation of the message passing interface (mpi) standard. https://www.mpich.org/.

Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic vertical elasticity of docker containers with elasticdocker. In *2017 IEEE 10th international conference on cloud computing (CLOUD)*, pages 472–479. IEEE, 2017.

Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE international symposium on parallel & distributed processing*, pages 1–12. IEEE, 2009.

Jérémy Buisson, Françoise André, and Jean-Louis Pazat. A framework for dynamic adaptation of parallel components. In *International Conference ParCo*, volume 33, page 65, 2005.

Xuan Chen and Shun Long. Adaptive multi-versioning for openmp parallelization via machine learning. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 907–912. IEEE, 2009.

Kaoutar El Maghraoui, Travis J Desell, Boleslaw K Szymanski, and Carlos A Varela. Dynamic malleability in iterative mpi applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 591–598. IEEE, 2007.

Dan Fulton, Laurie Stephey, R Canon, Brandon Cook, and Adam Lavely. Containers everywhere: Towards a fully containerized hpc platform. *Proc. Cray User Group (CUG)*, 2023.

Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

Jan Hungershöfer, Achim Streit, and Jens-Michael Wierum. Efficient resource management for malleable applications. *Paderborn Center for Parallel Computing, Tech. Rep. TR-003-01*, 2001.

Sergio Iserte, Iker Martín-Álvarez, Krzysztof Rojek, José I Aliaga, Maribel Castillo, Weronika Folwarska, and Antonio J Peña. Resource optimization with mpi process malleability for dynamic workloads in hpc clusters. *arXiv preprint arXiv:2506.14743*, 2025.

Gonzalo Martín, Maria-Cristina Marinescu, David E. Singh, and Jesús Carretero. Flex-mpi: an mpi extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, page 138–149, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 9783642400469. doi: 10.1007/978-3-642-40047-6_16. URL https://doi.org/10.1007/978-3-642-40047-6_16.

Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 11(2): 146–178, 1993.

Ahmad Qawasmeh, Abid M Malik, and Barbara M Chapman. Adaptive openmp task scheduling using runtime apis and machine learning. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 889–895. IEEE, 2015.

Manuel Rodríguez-Pascual, Jiajun Cao, José A Moríñigo, Gene Cooperman, and Rafael Mayo-García. Job migration in hpc clusters by means of checkpoint/restart. *The Journal of Supercomputing*, 75: 6517–6541, 2019.

Alex Scherer, Thomas Gross, and Willy Zwaenepoel. Adaptive parallelism for openmp task parallel programs. In *Languages, Compilers, and Run-Time Systems for Scalable Computers: 5th International Workshop, LCR 2000 Rochester, NY, USA, May 25–27, 2000 Selected Papers 5*, pages 113–127. Springer, 2000.

Gladys Utrera, Julita Corbalan, and Jesus Labarta. Implementing malleability on mpi jobs. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 215–224. IEEE, 2004.

Declaration of Scientific Integrity