Technische Universität München
TUM School of Computation, Information and Technology

Leveraging Dynamic Resource Management for Power Management and Fault Tolerance in High Performance Computing

Jophin John

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Darius Burschka

Prüfende der Dissertation:

1. Prof. Dr. Hans Michael Gerndt
2. Prof. Dr. Michael Georg Bader

Die Dissertation wurde am 16.04.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 02.09.2024 angenommen.

Zusammenfassung

Im High Performance Computing (HPC) werden Rechner der höchsten Leistungsklasse verwendet, die Billiarden oder sogar Trillionen von Berechnungen pro Sekunde ausführen können. Allerdings gibt es mehrere Herausforderungen bei der Entwicklung und dem Einsatz dieser Supercomputer. Die Packungsdichte von Transistoren nähert sich den physikalischen Grenzen, was das Ende des Mooreschen Gesetzes bedeutet und eine Steigerung der Rechenleistung nur durch die Kombination von immer mehr Rechnern ermöglicht, der Energieverbrauch pro Supercomputer erreicht heutzutage schon bis zu 30 MW und die Häufigkeit von Fehlern in diesen Systemen nimmt mit der Zahl der Komponenten drastisch zu. Daher werden innovative Lösungen benötigt, um diese Herausforderungen im High Performance Computing zu bewältigen. Die dynamische Ressourcenverwaltung ist eine solche Lösung, die einen flexiblen und einheitlichen Ansatz bietet. Diese Arbeit verbessert die Systemauslastung und somit die effiziente Nutzung von Supercomputern, reduziert den Stromverbrauch und erhöht die Fehlertoleranz. Hierzu wurde ein adaptiven Batch-Scheduler, ein Management des Power Corridors des HPC Rechners und ein adaptives Checkpointing-System entwickelt. Diese Systeme unterstützen sowohl klassische statische HPC-Anwendungen als auch adaptive Anwendungen, die in der Erweiterung iMPI des Message Passing Interfaces (MPI) programmiert sind. Der adaptive Batch-Scheduler verwendet Planungsstrategien, die die Rechenleistung und Effizienz der Anwendungen berücksichtigen und die laufenden Anwendungen rekonfigurieren und neue Anwendungen starten, um die Gesamtauslastung des Systems zu optimieren. Der energiegewahre Scheduler nutzt dynamische Techniken, um den Betrieb des Systems in einem festgelegten Stromkorridor zu gewährleisten. Das adaptive Checkpointing-System nutzt Kommunikation über entfernte Speicherzugriffe und das dynamische Management von Ressourcen, um schnellere und effizientere Checkpointing-Dienste für statische und adaptive MPI-Anwendungen bereitzustellen. Es automatisiert auch die Datenumverteilung für adaptive MPI-Anwendungen nach einer Anpassung der Ressourcen.

Abstract

In high-performance computing (HPC), supercomputers are the epitome of power and performance, processing billions and trillions of calculations per second. However, there are several obstacles that supercomputers must overcome as the demand for computational power continues to increase. For example, the physical limits of transistors are approaching (signalling the end of Moore's law), the energy consumption per supercomputer is exploding (the power consumption of an exascale supercomputer is sufficient to power a small city), and the inevitability of failures in systems with billions of components is increasing. As a result, innovative solutions are needed to overcome such challenges in supercomputing, ranging from scalability to power management and fault tolerance. Dynamic resource management is one such solution that offers a flexible and unified approach. This work improves system utilisation, optimises power consumption, and enhances fault tolerance by leveraging dynamic resource management techniques. Towards that, an adaptive batch scheduler, a power-aware scheduler and a fully adaptive checkpointing system are developed as part of this work to aid HPC applications that were created using standard and malleable (dynamic) Message Passing Interface (MPI) applications. The adaptive batch scheduler uses performance-aware scheduling strategies and improves the overall system utilisation by reconfiguring the running applications and launching new applications. The power-aware scheduler employs dynamic resource reconfiguration techniques and job scheduling to ensure the system's operation in specified power corridors. The adaptive checkpointing system, which is a major contribution of this work, leverages remote direct memory accesses and dynamic resources to provide faster and more efficient checkpointing services for standard and malleable MPI applications. In addition, this work also demonstrates that a checkpointing system can be used for data redistribution in malleable applications.

Acknowledgments

First and foremost, I extend my deepest gratitude to God for endowing me with the strength and capability to undertake and complete this endeavour. My heartfelt appreciation goes out to my supervisor, Prof. Michael Gerndt, who has been a source of inspiration and wisdom throughout my academic journey, from my master's degree to the present. I am immensely grateful for the opportunity to conduct my research under his guidance. His relentless curiosity and pursuit of knowledge have inspired me to evolve as a researcher. I owe a profound debt of gratitude to my parents (Amma and Chachan) for their relentless hard work and sacrifices, which have paved the path to where I stand today. Their love and blessings have been the cornerstone of my achievements. I am also thankful to my brother (Joby), who encouraged me to delve into research and continually motivated me to improve myself. My girlfriend, Jeeta, deserves special thanks for her unwavering presence, encouragement, and support throughout my research and personal challenges; her contribution has been invaluable. My journey was made more meaningful thanks to the company and support of my wonderful colleagues (especially Mohak, Eishi and Isaac) at the Chair of Computer Architecture and Parallel Systems. I am grateful to the InvasIC project for providing me with such a fascinating area of research. A special thanks goes to all my educators who have enriched me with their knowledge throughout my educational journey. Lastly, I am thankful for the prayers and positive energy from countless well-wishers and kind-hearted individuals in my life.

Contents

Zusammenfassung iii

Abstract v

Acknowledgments vii

1 Introduction 1
- 1.1 Motivation . . . 3
- 1.2 Contributions . . . 5
  - 1.2.1 Resource and Job Management . . . 5
  - 1.2.2 Power Management . . . 7
  - 1.2.3 Fault Tolerance . . . 8
- 1.3 Outline . . . 10

2 Invasive Computing 12
- 2.1 Research Groups . . . 12
- 2.2 Dynamic Resource Management Infrastructure . . . 13
  - 2.2.1 Invasive Resource Manager . . . 14
    - 2.2.1.1 Overview of Slurm . . . 14
    - 2.2.1.2 Extensions to Slurm . . . 16
    - 2.2.1.3 Expansion and Reduction for Malleable Batch Jobs . . . 17
  - 2.2.2 Invasive Message Passing Interface . . . 18
    - 2.2.2.1 iMPI Concepts . . . 19
    - 2.2.2.2 Writing a Simple Malleable MPI Program . . . 20
    - 2.2.2.3 Elastic Phase Oriented Programming Model . . . 23
- 2.3 Building Upon and Leveraging Invasive Infrastructure . . . 24

3 Related Work 26
- 3.1 Adaptive Batch Scheduling . . . 26
- 3.2 Power Corridor Management . . . 28
- 3.3 Checkpointing . . . 30

4 Adaptive Batch Scheduling 34
- 4.1 The Adaptive Batch Scheduler . . . 34
- 4.2 Adaptive Scheduling Strategies . . . 37
  - 4.2.1 Job Management Policies . . . 38
  - 4.2.2 Malleability Management Policies . . . 38
    - 4.2.2.1 Favor Previously Started Malleable Applications First (FPSMA) . . . 39
    - 4.2.2.2 Equi-Grow Shrink (EGS) . . . 40

4.2.3 Performance-aware Scheduling of Malleable Jobs . . . 42

5 Power Corridor Management **43**
5.1 Power-aware Batch Scheduler Concepts . . . 44
5.1.1 Linear Programming Model for Resource Reconfiguration . . . 44
5.1.2 Guarantees . . . 46
5.1.3 Power Measurement . . . 47
5.1.4 Forecasting . . . 47
5.2 Power-Aware Batch Scheduler Implementation . . . 48
5.2.1 Dynamic Power Corridor Management . . . 50
5.2.2 Limitations . . . 51

6 iCheck – Invasive Checkpointing System **52**
6.1 Architecture . . . 52
6.1.1 iCheck Core . . . 53
6.1.1.1 Agent . . . 53
6.1.1.2 Manager . . . 54
6.1.1.3 Controller . . . 54
6.1.2 iCheck Workflow . . . 54
6.1.3 iCheck library . . . 55
6.1.3.1 API for Checkpoint Transfer . . . 56
6.1.3.2 API for Malleability Support . . . 58
6.1.3.3 API for Faster Data Transfer . . . 58
6.2 Data Transfer in iCheck . . . 58
6.2.1 MR Approach . . . 59
6.2.2 SHMR Approach . . . 60
6.2.3 Buffer Management in iCheck . . . 60
6.2.3.1 Pipelining . . . 61
6.2.3.2 Versioning . . . 63
6.2.4 Asynchronous Checkpointing . . . 63
6.2.5 Multilevel Checkpointing in iCheck . . . 64
6.3 Monitoring . . . 64
6.3.1 Memory . . . 64
6.3.2 Checkpoint Operations . . . 65
6.3.3 Bandwidth . . . 65
6.3.4 Agent Count . . . 66
6.3.5 Power Usage . . . 66
6.3.6 iCheck Configuration File . . . 66
6.4 Dynamism in iCheck . . . 67
6.4.1 System Level . . . 67
6.4.2 Application Level . . . 68
6.5 Failures in iCheck . . . 70
6.5.1 Application Perspectives to Failure . . . 70
6.5.2 System Perspectives to Failure . . . 71
6.5.3 Node Failure . . . 72

7 Resource Management in iCheck **73**
7.1 Agent Management in iCheck . . . 73

7.1.1 Agent Count Selection . . . 74
7.1.2 Agent Placement . . . 75
7.1.2.1 Node Selection . . . 76
7.1.2.2 Distribution Scheme Selection . . . 77
7.1.2.3 Count Selection . . . 77
7.2 Malleable Application and Agents . . . 78
7.2.1 MA-RA Strategy . . . 78
7.2.1.1 Resource Expansion . . . 79
7.2.1.2 Resource Reduction . . . 79
7.2.2 MA-MA Strategy . . . 80
7.2.2.1 Resource Expansion . . . 80
7.2.2.2 Resource Reduction . . . 81
7.2.3 Agent Count Selection Algorithm . . . 81
7.2.4 Pseudocode for Malleable Applications . . . 83
7.3 iCheck and Invasive Resource Manager . . . 85
7.3.1 Messages and Policy in the Controller . . . 85
7.3.1.1 Communication Extensions . . . 85
7.3.1.2 Policy for Resource Extensions . . . 86
7.3.2 iCheck Aware Scheduler . . . 91
7.3.2.1 Communication Extensions . . . 91
7.3.2.2 Scheduler Plugin . . . 91
7.3.3 Controller-iRM-Application Interaction . . . 94
7.4 Data Distribution in iCheck . . . 96
7.4.1 API for Data Redistribution . . . 96
7.4.1.1 Replicated Data . . . 96
7.4.1.2 Distributed Data . . . 98
7.4.2 Pseudocode for Data Distribution . . . 99
7.4.3 State Diagram . . . 101
7.4.4 Resource Malleability Using Checkpointing . . . 101

8 Evaluation Setup **102**
8.1 System Setup . . . 102
8.1.1 SuperMUC Phase 2 . . . 102
8.1.2 SuperMUC NG . . . 103
8.1.3 Running RJMS inside RJMS . . . 103
8.2 Application Setup . . . 104
8.2.1 Adaptive Batch Scheduling . . . 104
8.2.2 Power Corridor Management . . . 104
8.2.3 iCheck – Performance and Resource Adaptivity Analysis . . . 105
8.2.3.1 Synthetic Application . . . 106
8.2.3.2 Synthetic Malleable Application . . . 107

9 Results **108**
9.1 Adaptive Batch Scheduling . . . 108
9.2 Power Corridor Management . . . 111
9.2.1 Forecasting . . . 112
9.2.2 Upper and Lower Power Corridor Enforcement . . . 112
9.2.3 Dynamic Power Corridor Enforcement . . . 113

9.2.4 Power-aware Scheduling Strategy . . . 113
9.3 iCheck – Performance Analysis . . . 115
9.3.1 iCheck vs MPI-IO . . . 115
9.3.2 Blocking vs Non-Blocking Checkpointing . . . 116
9.3.3 Push vs Pull Strategies . . . 116
9.3.4 iCheck Overhead Analysis . . . 117
9.3.5 Comparing iCheck, SCR and MPI-IO . . . 118
9.4 Resource Management in iCheck . . . 120
9.4.1 Adaptive Resource Management in iCheck . . . 120
9.4.1.1 Impact of Dynamic Agents . . . 120
9.4.1.2 Effect of Agent Placement Strategies . . . 121
9.4.1.3 Adding Nodes to iCheck . . . 122
9.4.1.4 Fault Tolerance in iCheck . . . 124
9.4.1.5 Checkpoint Compression in iCheck . . . 125
9.4.2 Malleable Application and Agents . . . 125
9.4.2.1 Malleable Application with Increasing Checkpoint Size . . . 127
9.4.2.2 Malleable Application with Fixed Checkpoint Size . . . 129
9.4.2.3 Impact of Pipelining . . . 131
9.4.3 Data Distribution in iCheck . . . 135
9.4.3.1 Performance Analysis . . . 136
9.4.3.2 iCheck vs MPI-IO . . . 136

10 Discussion **138**
10.1 Adaptive Batch Scheduling . . . 138
10.2 Power Corridor Management . . . 139
10.3 iCheck - Invasive Checkpointing System . . . 140

11 Conclusion **141**
11.1 Into the Future . . . 142

Index **145**

List of Figures **145**

List of Tables **147**

List of Algorithms **148**

List of Listings **149**

Bibliography **150**

Webliography **165**

1

Introduction

Supercomputing, or high-performance computing (HPC), is an advanced technology that orchestrates powerful machines to handle computationally intensive tasks and process massive amounts of data efficiently. Use cases of these powerful machines span multiple domains ranging from particle physics to molecular biology, quantum chemistry to chip design, data analysis to deep learning, and vaccine development to digital twins. Because of their high-speed parallel processing capability and enormous computational capacity that surpasses cloud computing, these robust machines are primarily used to perform complex simulations such as weather predictions, nuclear reactions simulations, biochemical interactions modelling, and large dataset processing in the domain of genomics and particle physics [1–14].

A supercomputer typically consists of tens of thousands or even millions of cores (the fastest supercomputer currently has 8.73 million cores [15]) working in parallel for problem-solving. These cores can be distributed across multiple machines using high-performance networks in a computing cluster or within a large machine. Complex tasks are broken down into smaller, independent parts that can be processed simultaneously across multiple cores, allowing tackling problems beyond the computational reach of independent components.

Supercomputing performs various roles across different industries, with each application generally necessitating considerable computational capabilities or the management of extensive data volumes. The two most common use cases of supercomputers and their domains are briefly listed below [1–6, 9–11]:

- **Simulations**: Supercomputers are indispensable in scientific research to simulate intricate phenomena that would be too costly and impractical to study via direct experimentation. They are instrumental in predicting weather and climate patterns, comprehending particle physics, analysing the formation of galaxies, and modelling biological systems such as the human brain or the protein folding process. Supercomputers are used across various industries, including automotive, aerospace, and electronics, to simulate and test the performance of designs in a virtual environment before the beginning of expensive and time-consuming physical manufacturing. Supercomputers advance drug discovery by simulating drug interactions with biological organisms. During the COVID-19 pandemic, supercomputers were used to simulate the virus's structure and develop vaccines [2].
- **Data Analysis and Artificial Intelligence (AI)**: Supercomputers are crucial for analysing and processing the vast amounts of data generated globally. For example, supercomputers can be efficiently used to process data for genome sequencing. Training complex AI models requires immense computational resources. Supercomputers can dramatically reduce the time required to train these models,

thereby accelerating the development and deployment of AI technologies (ChatGPT was trained on a specialised supercomputer [16]).

The impact of supercomputers is not limited to the above fields, and is spread across national security, energy exploration, cryptoanalysis and a plethora of other sectors. As seen above, HPC has numerous applications in scientific research, industry, and government and is crucial in climate modelling, biomedical research, data analytics, and national security. National governments or research institutions spend hundreds of millions of dollars to build the world's most powerful supercomputers [17–20]. By providing exceptional computational capability, these state-of-the-art machines aid in tackling complex problems and making scientific advancements that would not otherwise be feasible.

In high-performance computing, supercomputers are the epitome of power and performance, processing billions and trillions of calculations per second. However, there are several obstacles that supercomputing must overcome as the demand for computational power continues to increase [21]. Some of the key challenges in HPC are the following:

- **Power Consumption and Heat Dissipation**: Supercomputers are power-hungry machines. For instance, the Frontier supercomputer at Oak Ridge National Laboratory in the United States consumes about 21 megawatts of power – enough to power a small city [22]. As supercomputers continue to evolve and their power requirements grow, managing this power consumption becomes a critical issue. The immense power consumption of supercomputers along with millions of components generates lot of heat, facilitating the need for efficient cooling techniques. Failure to cool down the system can lead to hardware damage and results in system downtime. In addition, the cost of cooling the system costs around 25 - 40% of the total energy cost of a supercomputing facility [23, 24].
- **Fault Tolerance**: Millions to billions of individual components in a supercomputer indicates the failure is inevitable in some of the components. In addition, the Mean Time Between Failures (MTBF) for exascale machines is predicted to be in minutes [25]. Hence, developing supercomputers that can handle component failures is a significant challenge and fault tolerance must be delegated to applications to effectively handle the multitude of failures.
- **End of Moore's Law**: The development of supercomputing faces a new challenge as the physical limits of transistor size are approaching. Moore's Law has been relied upon for years to double the system performance by densely packaging the transistors every two years, but this growth rate is stalling [26–29]. As a result, the trend of performance improvements seen over the decades by employing intricate hardware design might cease to exist.
- **Software Optimisation**: As the end of Moore's Law is on the horizon, it is necessary to employ software optimization techniques to exhaustively utilize these powerful machines' resource capabilities. Inefficient software (both system and application) can cause the system not to scale well, leading to reduced performance. As systems continue to grow in size and complexity, efficient software development becomes paramount to ensure the full potential of a supercomputer and deliver optimal performance [30].

The HPC community extensively researches the above challenges, and novel solutions are proposed to tackle many of these challenges. Various techniques are employed to tackle different pieces of the challenges. For example, DVFS and power capping are used for power management and reducing heat dissipation [31, 32], fault tolerance techniques spanning application to system levels are used to bring resilience to the system [33], application optimisation and parallelisation techniques [34] are used to utilise the system effectively and to scale the application. This multitude of solutions are spread across different layers of the system.

System	Architecture	Cores	Perf. (Pflop/s)	Power. (kW)
Frontier	AMD EPYC 64C, AMD Instinct	8,699,904	1,194.00	22,703
Aurora	Xeon CPU Max 9470, Intel GPU	4,742,808	585.34	24,687
Eagle	Xeon Platinum 8480C, Nvidia H100	1,123,200	561.20	–
Fugaku	A64FX 48C	7,630,848	442.01	29,899
LUMI	AMD EPYC 64C, AMD Instinct	2,752,704	379.70	7,107

Table 1.1: Name, architecture, number of cores, performance and power usage of the top five HPC systems in the Top500 list on November 2023 [15].

One of the software-level techniques that can apply to all the above challenges is dynamic resource management [35], which has been gaining traction recently in the HPC community. In essence, dynamic resource management refers to a system's ability to dynamically adjust the usage of the computational resources, such as processing power and memory, based on the current system and application requirements. This became the motivation for this research work, where the challenges associated with system performance, power management and fault tolerance were tackled using dynamic resource management.

1.1 Motivation

Moore's Law has been a driving force behind the growth of supercomputing for decades [29]. However, as the physical limits of transistor miniaturisation are approached, relying on Moore's Law to provide regular performance increases is not feasible. As a result, the practice of stacking up resources to improve performance will be challenging. Instead, new ways must be found to improve the system's performance with existing resources. By optimising the use of existing resources, more performance can be obtained out of a supercomputer without requiring more hardware components. Hence, the traditional notion of static resources, where a subset of the machine's resources are given exclusively to an application until the application ends in HPC (static resource management), is getting challenged. More and more research activities are going on to develop dynamic applications and dynamic resource management techniques [35–39]. The applications can be programmed to adapt to resource allocation changes and enhance performance. The resource manager can be modified to reconfigure the resources of the running applications. It has opened up avenues of opportunity for improving the application as well as the system's performance by enhancing the overall system utilisation, application waiting time, and applications' makespan.

The application execution pipeline of a supercomputer is as follows. The application information, along with the requirements (number of resources, total execution time) for the application, is submitted to the batch system. This process is typically known as job submission. Upon job submission, the batch system adds the application (job) to a queue (job queue), assigns a priority (job priority), and runs the application based on the availability of resources and site-level policies for job management in the supercomputing centre. Different policy factors come into play between a user and an HPC system, unlike in a cloud scenario where users can run the applications if they can pay for the resources [40, 41]. These policy factors are inculcated into the supercomputing centre's resource and job management systems and are unique to each centre. The state-of-the-art techniques for job management in these centres are static and can lead to inadvertent delays in launching new jobs. For example, static resource management hinders the possibility of rearranging the running jobs to make space for new jobs in the queue, thereby increasing the job response time. This research work catalyses the growing trends in resource dynamism (malleability) by proposing adaptive resource management strategies that improve system performance by dynamically redistributing workloads across the supercomputer's resources and launching new jobs in the queue.

Dynamic resource management can cater to usefulness beyond the system and application performance. Towards that extent, in this research work, the dynamism of resources was also leveraged to provide power corridor management and fault tolerance. As governments and institutes compete to bring more performance into their latest supercomputers [18–20], the issue of power management and fault tolerance becomes more relevant than ever in the HPC community. Table 1.1 demonstrate that supercomputers' power consumption and component complexity change dramatically with each new supercomputer. It can be inferred from the table that the average power usage and number of components of the top five supercomputers are humongous and are 20MW and 4 million, respectively. As a result, power consumption and resilience to failures became the central focus of this work.

Typically supercomputing centres have an energy budget that is negotiated with the energy suppliers [42]. The supercomputing centres are obliged to operate within the so-called power corridor, and penalties are incurred for the power corridor violations. To maintain the power budget, these centres employ various techniques spanning shutting down the system, power capping and dynamic voltage and frequency scaling [32]. These strategies often kill the applications if the upper and lower budget cannot be maintained. Dynamic resource management can be leveraged to tackle this by ensuring the system operates at the power budget by adjusting resource usage in real-time. For example, parts of the supercomputer can be powered down by redistributing the applications running on those resources. This dynamic approach to power management can ensure that the applications can be run without interruption and that the system's power usage can be managed simultaneously. Dynamic resource management can intelligently allocate computational jobs and balance power consumption by continuously assessing the system workload and identifying underutilised resources. Additionally, power usage can be proactively managed, promoting sustainability and reducing operational costs. For example, a supercomputing centre can make a policy so that the system can work at its full potential when energy is obtained from renewable sources while reducing energy usage with non-renewable energy sources. This thesis demonstrates that the resources of a running application can be reconfigured proactively to adhere to the system-level power budget and maintain overall system-level power usage.

With the continuous addition of computational power to achieve exascale performance, maintaining resilience against failures becomes a critical aspect in developing HPC applications [43]. The significant leap in performance from petascale to exascale is expected to increase the rate of failures [44]. Currently, the mean time between failures (MTBF) in petascale systems averages around hours, but it is anticipated to decrease to minutes in the exascale era [25]. Consequently, application developers need to prepare for more frequent failures and implement adequate fault tolerance and recovery mechanisms. Dynamic resource management has yet to be fully utilised in fault tolerance techniques. Dynamic resource management systems can continuously monitor the system state and adjust its use of resources to compensate for the failure, for example, by rescheduling tasks to other components or bringing spare components online. This ensures that the application can continue even in case of failure. One of the most famous fault tolerance techniques employed in simulation codes is checkpointing [33], where an application frequently stores its state (checkpoint) such that whenever a failure happens, the application can restore itself from the most recent checkpoint. However, the application's rigid characteristics (non-dynamic) hinder the state-of-the-art checkpointing designs from attaining peak performance. Additionally, checkpointing systems are rigid due to the predominantly static view of resources in the HPC community. This work proposes a new checkpointing system that dynamically changes its resources and properties based on the system state, ensuring better performance for rigid and dynamic applications. Furthermore, leveraging dynamism enables the optimisation of checkpointing strategies by intelligently choosing checkpoint intervals and improving the checkpoint transfer rate dynamically by allocating more resources to minimise the loss of work in case of failure. It can also lead to improvement in application performance. This requires dynamic resource management capability on the system side and a dynamic checkpointing system. In this work, the checkpointing performance of

applications is improved significantly by using techniques of adaptive resource management.

Dynamic resource management offers many optimisation and performance enhancement opportunities that are otherwise impossible with a rigid view of resources [36]. However, a dynamic system will only be benefitted if there are dynamic applications that can adapt to the resource changes. One of the most popular libraries used to write distributed memory applications is Message Passing Interface (MPI) [45], which is inherently static. In short, it is not possible to write pure malleable applications, i.e., applications that can change their resources during the execution. There are proposals in the MPI community for bringing malleability, and different research groups have their own custom MPI that supports malleability [36, 37, 46]. Nevertheless, writing malleable applications is not trivial, and one key challenge in doing so is the data redistribution during the resource change. In this work, it is demonstrated that the checkpointing system can also act as a data redistribution framework for a malleable application.

In contrast to the dynamic ecosystem of the cloud technology where portability mattered more than performance, HPC remained static to adhere to the performance objectives of the compute intense applications [47]. The scientific applications are fine-tuned for particular machines to obtain maximum performance [48]; hence, the static view of resources was sufficient. The demand for accelerator support and the prevalence of GPUs and FPGAs is driving the HPC's model of homogeneous architecture into oblivion, thereby transforming the field into a heterogeneous environment (or more cloud-like) [49]. As a result, to support the varying workloads and rapidly evolving field of artificial intelligence, HPC will eventually transition into dynamic resource management to assist different workloads and use cases; for example, AI focuses on faster execution and lower precision calculations, while scientific simulations focus on high accuracy and performance [47]. Hence it becomes imperative that priority be given to dynamic resource management and investigate how it can improve high-performance computing. This research demonstrates that dynamic resource management can address some of HPC's prevalent challenges by improving the job management system, maintaining the system-level power budget, and enhancing the checkpointing performance. The following section provides a synopsis of the contributions written in this thesis.

1.2 Contributions

The contributions from this work span three distinct areas, and the primary focus was leveraging the dynamism in resource management to demonstrate its impact in a multidimensional space. The contributions were made in the following three areas, with significant contribution in the area of fault tolerance:

- Resource and Job Management
- Power Management
- Fault tolerance

Combining these areas cover the entirety of the field of HPC, and the niche for dynamic resource management in each area was found. The contributions in each area are explained in each of the subsections below.

1.2.1 Resource and Job Management

Resource and Job Management is one of the core tasks of any high-performance computing centre. These systems are known as Resource and Job Management Systems (RJMS), Resource Management Systems (RMS) or Batch Systems. Slurm [50], Flux [51], Load Leveler [52], and PBS/Torque [53] are some of

the popular RJMS systems used in top supercomputing centres worldwide. These systems typically have a resource management component as well as a job scheduler component. The resource management aspect of the component deals with the job requests, execution of the job, managing the communication between compute nodes and creation and termination of application processes. The job scheduler finds resources for a given job. It iterates through the job queue and resource list to find the job-to-resource mapping. For this, many different sets of policies (for example, the priority level of users) and site-level factors (for example, maximum allocatable resources for an individual user) are considered. Once the mapping is found, the resource manager continues with the job launch and execution.

The problem with current state-of-the-art RJMS systems in HPC is the rigidity of the resource allocation; that is, the resource managers expect that the application will need the same number of resources from start to end. This assumption does not hold for dynamic applications like adaptive mesh refinement or embarrassingly parallel applications where scalability is proportional to the number of resources. In the former case, different phases of the application ideally require different numbers of resources. For example, for a tsunami simulation, in an ideal scenario, the number of resources needed in the beginning is less compared to the end of the simulation. This can be attributed to the fact that the beginning of the simulation contains fewer grid points than the middle, where the wave propagates to wider areas. The state-of-the-art resource managers do not consider these kinds of application-specific characteristics. Research has focused on these aspects [36], and runtime managers were proposed to reconfigure the applications during the execution based on the policy.

However, it is essential to note that such runtime managers can only perceive the jobs currently being executed. As a result, it may overlook several optimization opportunities that could potentially improve the system throughput while making resource reconfiguration decisions. To overcome this limitation and enable optimal resource allocation, a batch system that integrates both the runtime system and the job scheduler is crucial. For example, consider a scenario where a runtime system is reallocating the resources of applications based on their performance. It decides to take away resources from the low-performing application to better-performing ones even though a better-performing one does not need more resources. In such cases, resource reconfiguration can be performed by taking the resource requirement of jobs waiting in the queue into account to improve the overall system throughput. This gap between the dynamic runtime system and the job scheduler is bridged in this work. Since creating a whole RJMS system is impractical, the dynamic runtime system created as part of the InvasiC project [54] is used in this work. It is an extension of Slurm, one of the world's most popular RJMS systems used in top supercomputers worldwide.

The contributions from this work spread across both job scheduler and resource management. As part of this work, the job scheduler of Slurm is extended to support the malleable applications and created a new malleable job scheduling plugin. The resource management component of Slurm is modified to accompany the new job scheduling plugin to improve the system throughput. The dynamism aspect of the work comes into play as follows. Typically, when a job is submitted, the job scheduler looks for the resource requirements and, if matched, maps the job to the resources. This work extends this job requirement to support job moldability along with job-specific constraints (For example, square number of nodes). Moldability refers to the ability of the batch system to change the number of resources for the job before its start. The constraint support is provided such that the job scheduler can select a valid start resource requirement within a minimum and maximum resource requirement that satisfies the job constraint. For example, in the proposed work, the job request can contain a minimum resource requirement of 16 and a maximum of 36 with a constraint square number of resources. It means the job scheduler can map the job with a resource count of 16, 25 or 36. The proposed work also supports dynamic resource management by analysing and reconfiguring the resources of the running applications on-the-fly and scheduling new jobs. For that, the application performance is analysed, the resource requirement of jobs in the queue is taken into account,

and then applications with poor performance are penalised by taking away the resources and given to new jobs.

Multiple job scheduling strategies were implemented to demonstrate the effectiveness of the proposed system. The contributions in this area [55] can be summarised as follows:

- A performance-aware job scheduling plugin where the running application's performance is considered for job scheduling.
- The system is evaluated on a job allocation on SuperMUC-NG [56] and demonstrated an improvement in job response time up to 29%, job waiting time by up to 26%, and makespan by up to 19% compared to a backfill scheduler [57].
- Demonstrates running an RJMS system inside another RJMS system in SuperMUC-NG. Techniques were developed to launch a custom Slurm software within a job executed by another Slurm scheduler.

1.2.2 Power Management

As seen in the motivation section, power management is one of the HPC challenges in the exascale era. The current state-of-the-art techniques to address these challenges were geared towards the solutions from the system perspective. For example, power capping can be used to limit the power usage of a node. In contrast, Dynamic Voltage and Frequency Scaling (DVFS) can be used to limit the processor frequency and voltage to reduce the overall power used in the node. In addition, idle nodes are shut down to maintain the power usage of a supercomputing centre. In some scenarios, the running applications are killed, and the nodes are shut down to bring the system-level power usage into the desired corridor. These solutions are practical and used across different supercomputing centres worldwide; however, solutions that also consider the application perspective can improve the existing solutions. Consider a scenario where shutting down a system's nodes and killing applications to maintain the system-level power budget. Here, the abrupt killing of the application results in the cessation of the application's progress, and as a result, it needs to recompute everything from the start during the restart (or restart from the checkpoint). In such scenarios, dynamic resource management can be beneficial.

This proposed work shows the usage of dynamic resource management for power management by manipulating the power usage characteristics of the applications. The proposed works focus on power corridor management, which aims to maintain the system power usage within a lower and upper bound. A power-aware scheduler plugin, along with a power-aware runtime management system, is proposed in this work. The core idea is to take away resources from power-hungry applications and give them to low-power running applications whenever there is a need to reduce power usage and vice versa to increase power usage. The application's power usage characteristics are modelled using a linear programming model.

RJMS used for this work is the adaptation of the RJMS proposed in the above section. The resource management and job scheduler components of the Slurm are extensively modified to enable this work. A rudimentary power model from Narvaez [58] is extended to support multiple applications. Two approaches, ❶ proactive and ❷ reactive, were considered in this work. In the proactive approach, multiple models were used on-the-fly to predict the systems' power usage and applications are reconfigured to maintain the system-level power boundary if a power corridor violation is anticipated. In the reactive approach, the resource manager will only react when there is a power corridor violation. In addition, new jobs are scheduled to maintain the system-level power budget. Similar to the contribution in section 1.2.1, the moldability aspect is also considered for the scheduling. The new resource requirement for a job in the queue is considered in relation to the application's power consumption. Hence, it is ensured that the new application will

not violate the power corridor. The advantage of the proposed work is that the system's power usage can be maintained without shutting down the nodes or killing the power-hungry applications. Hence the application progress can be maintained.

In the proposed work, the power corridor can also be altered dynamically. The resource manager obtains the power corridor values from the external source and uses them for making scheduling decisions. Such a technique can act as a trigger for high utilisation of the machine. For example, suppose the upper and lower power corridors are high. In that case, the power-aware resource manager will provide additional resources to power-hungry applications and ensure the system remains in the specified corridor. High-performance computing centres can use these techniques to use clean energy (renewable energy sources) ideally. The centres can set high power corridor values while using clean energy and low power corridor values while using coal energy. However, the fundamental requirement for such a work is the availability of sufficient malleable applications.

The contributions from this work [55, 59] can be summarised as follows:

- A power-aware job scheduling plugin – The running application's power usage is considered for job scheduling along with applications in the queue. New jobs are scheduled by maintaining the system-level power constraints.
- Supports dynamic power corridors – The power corridors can be changed on the fly, and the power-aware runtime will reconfigure the applications to bring the system back into the power corridor.
- The experiments were run on SuperMUC-NG and demonstrated that power-aware scheduling strategies can prevent power corridor violations if certain guarantees are met.

1.2.3 Fault Tolerance

The concept of resilience to failures in computing is a research domain active since the advent of the field of computing itself. Tolerance to failure is one of the critical challenges in the HPC ecosystem due to the complexity of components in creating a supercomputer. An increase in complexity increases the chances of failure. The failure of components impacts the application's performance, and most high-performance computing applications are not resilient to failures. When it comes to addressing failures in HPC applications, it's important to approach the issue from a software engineering perspective. In particular, since many HPC applications use the Single Program Multiple Data programming model with the Message Passing Interface (MPI), it's especially important to provide fault tolerance for MPI applications [45]. MPI is one of the most popular programming models for writing distributed memory applications, so it's critical to ensure that these applications can continue running even in the face of failures.

The MPI [45] lacks a built-in fault tolerance mechanism, and failure in one process affects the entire application.. This means that if even one process fails, the entire job could be at risk of failure. While attempts are being made to develop a fault-tolerant version of MPI [60–63], it has yet to be standardized. In the meantime, many HPC applications, including multiphysics and multiscale codes, still rely on checkpointing [64] to manage fail-stop errors. This method entails regularly saving the current state of the application so that, in the event of a failure, the application can be restarted from the most recently saved state.

There are two main approaches to implementing a system for saving and restoring data in case of failures: application-level checkpointing and system-level checkpointing [65]. In application-level checkpointing, the user decides what specific data must be stored and recovered, while in system-level checkpointing, the entire checkpoint process is hidden from the user. In the event of a failure, the application is restarted by restoring the saved data and continuing where it left off. The proposed work focuses on application-level

checkpointing because it is more efficient, has less size per checkpoint, low overhead, and quicker restart capability than system-level checkpointing.

Parallel file system contention is a significant issue in such approaches. When numerous applications attempt to write checkpoints simultaneously, it causes resource contention and negatively impacts performance [66]. To address this problem, we propose iCheck, a dedicated compute node software system that utilises remote memory accesses to save checkpoints from application into the compute nodes of the software instead of writing the checkpoint data directly into the Parallel File System (PFS) [66]. iCheck offers asynchronous multilevel checkpointing support to both MPI and non-MPI applications using dynamically reconfigurable remote direct memory access (RDMA). Resource and data management decisions are made while considering factors such as available memory, bandwidth, and checkpoint frequency.

As seen in the above sections, there are limitations to static resource management techniques prominent in HPC that can hinder many optimisation opportunities provided by dynamic resource reconfiguration. This static behaviour is prevalent from system services to applications [51, 67]. However, the potential for dynamic applications using dynamic resources extends far beyond better system utilization [38, 59, 68, 69]. Checkpointing is one area where this is particularly evident [70–72], as dynamic checkpointing can lead to faster transfer times and more efficient utilisation of resources such as memory and file system bandwidth. With the trend towards malleability in MPI and resource management infrastructure [36, 37, 73–76], it becomes even more critical for checkpointing systems to be adaptable to changing system requirements. None of the state-of-the-art checkpointing systems can adapt to the change in the number of resources of the application. By redistributing data during resource change, checkpointing systems can help address one of the significant challenges in writing malleable applications.

The proposed works address these challenges by positing checkpointing systems as a resource and data management system that manages the checkpointing provisions of multiple applications and interacts with resource managers to deliver seamless checkpointing capabilities. iCheck can dynamically reconfigure its resources (changes the number of checkpoint retrieval processes) and offers application-level checkpointing and data redistribution services to malleable (dynamic) MPI applications. This technique of utilising application-level checkpointing as a dynamic resource and data management service offers novel contributions to the checkpointing field w.r.t. to the state-of-the-art checkpointing systems.

Using dynamic resource management, iCheck brings two critical benefits to application-level checkpointing in HPC. Firstly, iCheck has the ability to horizontally scale its resources (For example, checkpointing nodes). As a result (in Subsection 9.4.1.3), the necessity to assign a dedicated group of resources for checkpointing becomes obsolete. A supported resource manager can dynamically alter the number and types of nodes allocated to checkpointing services. To provide horizontal scaling into iCheck, the above-described RJMS system is extended with iCheck-aware plugins to support iCheck. As a result, iCheck can obtain more resources from the RJMS system and can utilise them to improve the checkpointing services provided to the application. Secondly, iCheck can also dynamically modify the checkpointing processes assigned to an application based on metrics like available memory and checkpoint frequency. It is demonstrated in Subsection 9.4.1.1 that reconfiguring checkpointing resources (agents) dynamically improves the checkpointing performance, and distributing it across separate compute nodes can have a significant effect on the overall checkpointing bandwidth utilisation (Subsection 9.4.1.2). Furthermore, the resource dynamism in iCheck also addresses the rising trends in malleable MPI [55, 59, 75]. For example, a resource change happening in a malleable MPI application can initiate a corresponding change in checkpointing resources to maintain current checkpoint performance.

The contributions from this work [77, 78] can be summarised as follows:

- Adaptive application-level Checkpoint/Restart library *iCheck* leveraging RDMA and supports standard MPI, malleable MPI and non-MPI applications.

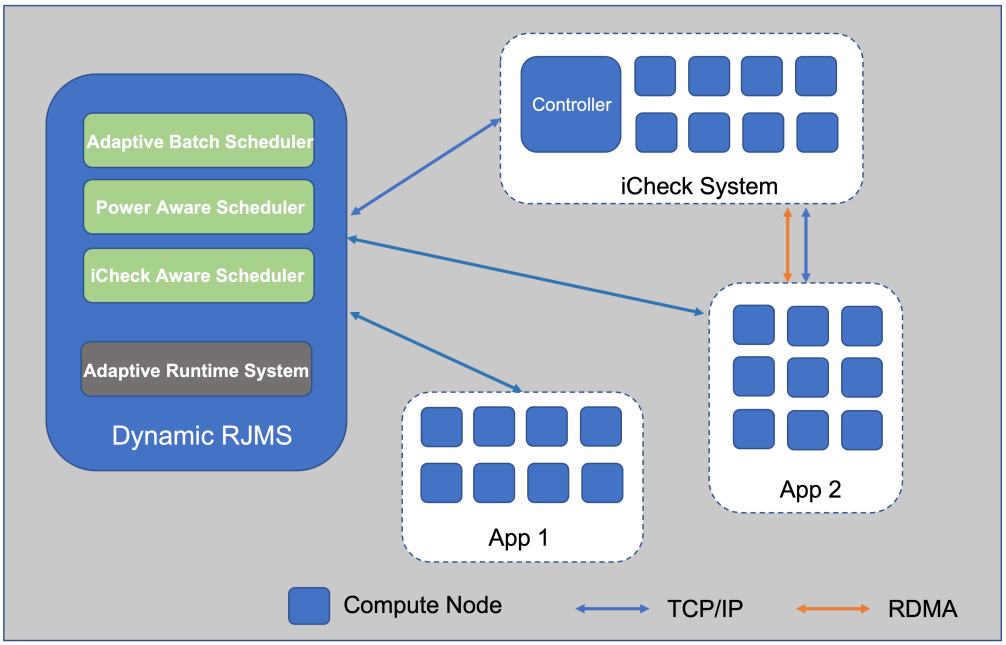


Figure 1.1: Overview of the contributions from this work

- A fault tolerance system that scales its resources horizontally and vertically based on the checkpointing requirements of the system and application.
- Checkpointing system that supports malleable applications and offers a prototype data distribution library to assist the development of malleable MPI applications.

1.3 Outline

The overall contributions from this work are depicted in Figure 1.1. The figure shows the complete system design performed as part of this work. The RJMS system in the figure has a resource-aware scheduler, a power-aware scheduler and an iCheck-aware scheduler. Based on the Resource Management (RM) policy, a corresponding scheduler is picked for job management. The applications are connected to the RJMS system, and the resource redistribution is performed based on the current RM policy. The RJMS also communicates to the iCheck system when the policy is iCheck-aware. iCheck works as independent software, and the interaction with RJMS occurs only during iCheck-aware policy. The applications running on compute nodes contacts iCheck for checkpointing and based on current checkpointing policy, iCheck performs the optimal checkpointing of the application. Furthermore, applications can also use iCheck for data distribution.

In a nutshell, the work of the thesis regarding the chapters can be described as follows:

This work ([55, 59, 77, 78]) takes an available dynamic resource management system (7.4.4) and then

1. Added adaptive batch scheduling with a performance-aware algorithm that improves makespan over the state-of-the-art backfill scheduler (Chapter 4) [55].
2. Created a new batch scheduler to demonstrate that dynamic resource management can be used for power corridor management (Chapter 5) [55, 59].
3. Created a self-adaptive checkpoint system that dynamically performs resource management and supports dynamic applications (Chapter 6) [77].

4. Extended adaptive batch scheduler to create an invasive checkpoint system that dynamically grows or shrinks based on the resource manager (Chapter 7) [78].
5. Demonstrated that a checkpoint system can be used to perform data redistribution in a malleable application (Chapter 7).

The outline of the chapters in this thesis is as follows. Chapter 2 describes the concept of Invasive Computing, which lead to the development of the existing dynamic resource management. Related works associated with the contributions in this work is described in detail in Chapter 3. The adaptive batch scheduler part of this work is described in Chapter 4. Chapter 5 covers the power-aware resource management in detail. Chapters 6 and 7 describe the checkpointing aspect of this work. Chapter 6 introduces iCheck and its design principles. Chapter 7 explores the dynamism in iCheck and covers the data distribution feature of the iCheck. Chapter 8 introduces the system setup used in the experiments and the applications used for the evaluation. Chapter 9 summarises the results in each of the three areas covered in this thesis. Chapter 10 discusses the results and open questions, and Chapter 11 provides the key takeaways from this work and an outlook into the future.

2

Invasive Computing

This work was conducted as part of the Transregional Collaborative Research Center Invasive Computing (TCRC 89) [54], funded by the Deutsche Forschungsgemeinschaft [79], to explore resource-aware programming of future parallel computing systems.

Invasive Computing [80] aims to create a hardware and software stack that supports dynamic applications that can utilize the compute resources efficiently in terms of energy and performance. The invasive paradigm employs the concept of invade, infect, and retreat, as shown in Figure 2.1. The application can first invade (choose a core) a compute resource, then infects (executes a part of the code in the selected core) it, and then retreats (give back the core).

The project spans various technical disciplines, from embedded systems to high-performance computing, compilers to simulations, and modeling to machine learning. The Invasive Computing project had three research and funding phases, and this work is part of the last funding phase of the project. The project has different research groups investigating different domains surrounding the resource-aware programming paradigm.

2.1 Research Groups

The research groups contained researchers from three participating universities in Germany (Friedrich-Alexander-Universität Erlangen-Nürnberg [82], Karlsruher Institut für Technologie [83], Technische Universität München [84]). The research groups [81] were divided into five groups based on the research area, with the group name being letters from A, B, C, D, and Z. Within each group, there are subgroups with the number followed by the group identifier letter (e.g., A1, D3) as its name. The figure 2.2 provides an overview of the research domains of each group and its subgroups. The horizontal axis depicts the hardware abstraction covered in the project, while the vertical axis visualizes the abstract layers concerning the subproject.

Project area A focuses on invasive computing fundamentals, language, and algorithm design. Subproject A1 investigates the fundamentals of invasion, which includes the core concepts, key features, and extensive analysis of invasive programs and their management strategies in invasive systems. A4 explores and characterizes existing algorithms and develops novel algorithmic patterns for invasive computing to obtain the

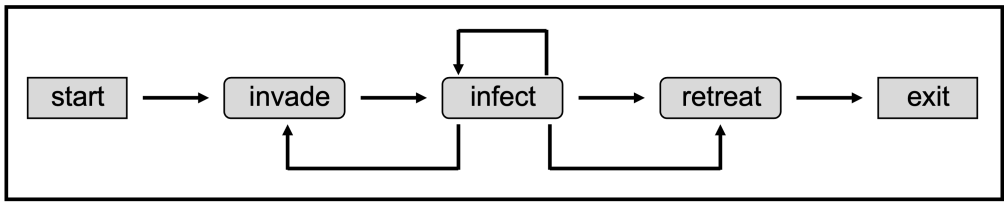


Figure 2.1: Invasive computing fundamental concepts [81]

application's desired objectives (For example, performance and energy usage). A5 explores the techniques for scheduling invasive applications under uncertain input to ensure performance guarantees regarding predictability.

Project area B focuses on the area of architectural research. Subproject B1 investigates the mechanisms for application-specific adaptivity using a runtime reconfigurable fabric in the micro-architecture. B2 focuses on employing invasive computing on tightly coupled processor arrays (TCPAs) to provide energy efficiency and performance improvement to computational-intensive applications that use nested loops. Subproject B3 focuses on developing novel methods for hardware and software building blocks to devise a power-efficient multiprocessor system on a chip (MPSoCs) using the invasive computing paradigm. B4 explores techniques for runtime verification and distributed monitoring of invasive applications. B5 designs and develops hardware and protocols for invasive Networks-on-a-Chip (NoC), bringing dynamism to the interconnect architecture.

Project area C explores the system software aspect of the invasive computing project. C1 develops a highly scalable operating system (iRTSS - Invasive Run-Time Support System) to enable and support resource-aware applications. C3 creates the compilers for invasive applications and program transformation and optimization techniques. Subproject C5 focuses on bringing security into the invasive computing domain to ensure confidentiality, integrity, and availability in invasive systems.

Project area D investigates the use of invasive computing concepts on the application side. D1 subproject explores the advantages and disadvantages of using invasive computing in humanoid robotics. D3 investigates the benefits and limitations of invasive computing concepts in High-Performance Computing.

Subproject Z2 span across all of the above project areas and assists in validating and demonstrating the results from the invasive computing project.

2.2 Dynamic Resource Management Infrastructure

This research work was done in the context of subproject D3, where Invasive Computing meets HPC. As part of the effort to bring dynamism (synonymous with the concept of invade, infect, and retreat) into HPC, D3 developed software infrastructure to support dynamic resource management and application development.

A holistic and layered approach is essential to bring resource dynamism into the HPC software stack. Changes must be made in the existing software stack at the system and application levels. At the system level, the resource and job management system need to be adapted, and at the application level, the programming interfaces should be modified, and the programming style needs to be updated. In Subproject D3, as part of the holistic approach, the Invasive Resource Manager (`iRM`), the Invasive MPI (`iMPI`), and the Elastic Phase Oriented Programming model (`EPOP`) were designed and developed. These three different parts together constitute the invasive infrastructure.

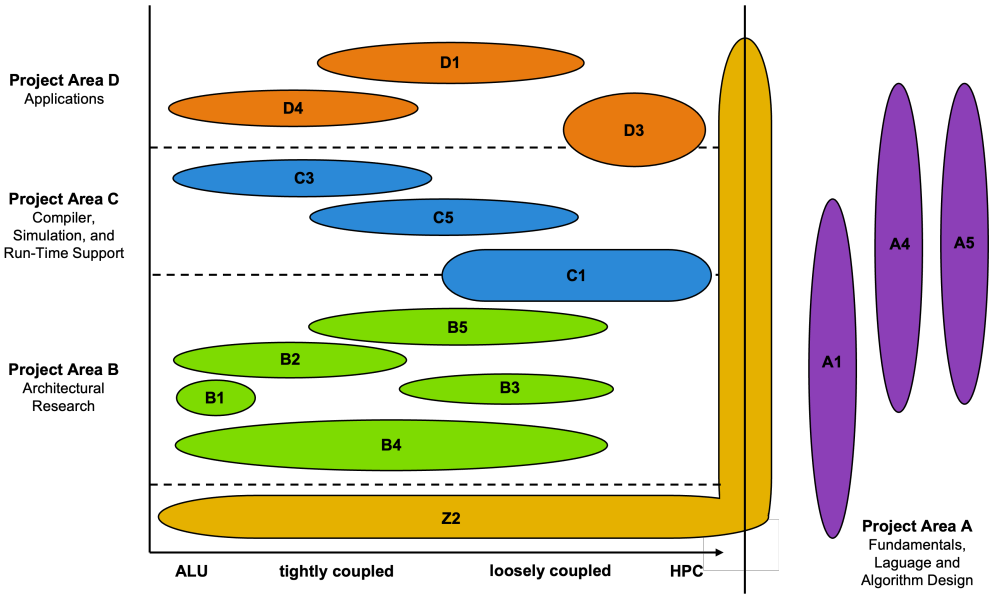


Figure 2.2: Research groups in Invasive Computing [81]

2.2.1 Invasive Resource Manager

Resource and Job Management System (RJMS) is crucial to modern HPC systems. It is responsible for decision-making and allocation of resources for application execution, pinning of processes on the hardware, accounting, and analysis of the system resources. It is a comprehensive middleware component that consists of subsystems for resource management, job management, and scheduling of resources. There are many such systems, and Slurm is one of the world's most widely used RJMS systems and an integral part of some of the world's fastest supercomputers.

Invasive Resource Manager (`iRM`) [46] is an extension of Slurm to support running malleable jobs, i.e., jobs that can change their compute resources during the execution. Traditional RJMS systems in HPC are designed to accommodate only rigid jobs, i.e., jobs that use a fixed number of compute resources.

2.2.1.1 Overview of Slurm

Slurm [85], formerly Simple Linux Utility for Resource Management, is an open-source, scalable, fault-tolerant, and popular RJMS software from SchedMD. About 60% of the TOP500 supercomputers use it as the workload manager [86]. In a nutshell, Slurm provides the core functionalities of ❶ allocating resources to users for a certain amount of time, ❷ managing the queue of pending jobs and performing resource arbitration, ❸ providing frameworks to start, execute, and monitoring nonparallel as well as parallel jobs (For example, Message Passing Interface support).

Slurm is designed to be a highly scalable RJMS system with a general-purpose plugin mechanism to support a wide variety of infrastructures [87]. Though multiple plugins cater to different use cases, such as accounting to scheduling, node selection to network topology in different infrastructures, the system design and components remain the same across different architectures. Some of the critical components (binaries) in Slurm with regard to job management and execution are `slurmctld`, `slurmd`, `slurmstepd`, `srun`, and `sbatch`. These components run on different compute resources (nodes) in a typical HPC system. Figure 2.3 shows that each component is placed among compute nodes based on its functionality.

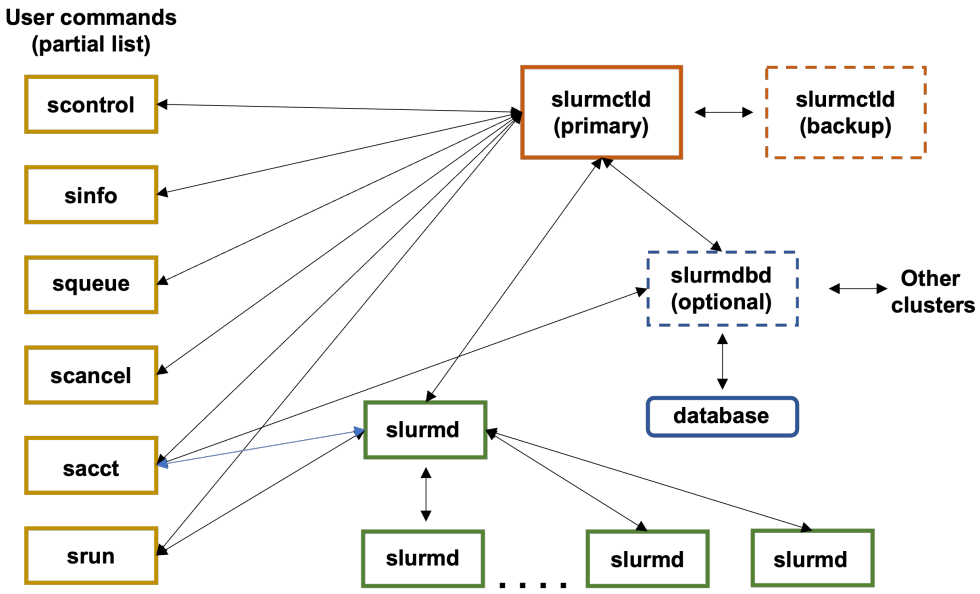


Figure 2.3: Slurm Architecture

`slurmctld` (Slurm controller) is the centralized resource manager monitoring system resources and job executions. `slurmd` is a worker daemon running on every compute node of the system. `slurmd` waits for the work, executes it, and then returns its status to the Slurm controller. Upon getting the work, the `slurmd` daemon launches the `slurmstepd` daemon, which is in charge of launching the job inside a compute node and manages the communication among node-local processes. `srun` is an interactive blocking tool to initiate the jobs and uses the command line to pass the job metadata to the Slurm controller. `sbatch` is used to submit batch jobs (using a job script that contains the job metadata) and is non-blocking. These binaries communicate with each other using Remote Procedure Calls (RPCs).

A simple control flow of job execution in Slurm can be defined as follows:

1. Users submit a job script using the `sbatch` command. The job script (as seen in Listing 2.1) contains the essential information for that job, such as the node count, wall clock time, partition requirements, user information, application information (application binary and input), and other relevant information required by the HPC admin for accounting and bookkeeping. Job script can also contain multiple `srun` commands to launch multiple applications. Each instance of such an `srun` call is called a job step, and the order of execution of a job step follows the order defined in the batch script.
2. Slurm places the submitted job in a job queue (if resources are not immediately available or some constraints are not met). A job is executed immediately when the resources are available, and the job constraints can be satisfied. The starting time of a job cannot be guaranteed and depends on multiple factors, including user constraints (job duration, partition), priority, and scheduling strategy. The system administrator of the cluster sets the scheduling policy and job priorities based on their site-level policy [40].
3. When the resources become available, and sufficient priority is attained, the `slurmctld` allocates the compute nodes for the job. It informs the `slurmd` daemon inside the first allocated compute node to launch the job using the RPC request message (This message contains the job metadata). The `slurmd` daemon goes through the job script, creates job steps (using `srun`), and initiates the `slurmstepd` to launch the job step based on the job script parameters. After the execution, corresponding RPC calls

```
#SBATCH --job-name sample_test_job
#SBATCH --time=00:30:00
#SBATCH --output=sample_test_output.txt
#SBATCH --account=testuser
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=96
srun sample_app #first job step
srun sample_app #second job step
```

Listing 2.1: Example batch script with additional parameters for `sbatch`.

are sent to notify the end of job steps. Once all the job steps are finished, `slurmctld` notifies the `slurmd` node daemons to terminate jobs and `slurmd` notifies the `slurmctld` immediately after the node cleanup. After the cleanup, the allocated nodes are available, and `slurmctld` can use them for scheduling other jobs.

A design flaw in such an approach was that it takes away many optimization opportunities provided by the dynamic characteristics of various applications. For example, an application can have multiple phases where a compute-intensive phase might need n nodes while other phases (IO, finalization) only need m ($m < n$) nodes. In such cases, $n - m$ nodes can be efficiently utilized for applications that benefit from additional nodes. Slurm cannot perform such dynamic resource management and needs to be extended rigorously to support it.

2.2.1.2 Extensions to Slurm

The components `slurmctld`, `slurmd`, `slurmstepd`, `srun`, and `sbatch` were extended to bring the resource dynamism into Slurm. As seen from the above section, `slurmctld` is in charge of job selection and execution, node selection and allocation, while the other components submit and execute the job. Hence to support dynamism in resource management, the `slurmctld` needs to be extended to two levels. In the first level, the job scheduler is modified to manipulate the resource allocation (expand/shrink the nodes) of running jobs, while in the second level batch system is modified to select the jobs to schedule (pick jobs from the queue based on the node availability). As a result, the `slurmctld` was replaced by two components an Invasive Runtime Scheduler (`iRS`) and an Adaptive Batch Scheduler (`ABS`). `iRS` was designed and developed by Ureña [46] while `ABS` was designed and developed as part of this work and is explained in detail in Chapter 4.

`iRS` is in charge of performing the runtime resource reconfiguration of the running applications. It is also extended with the capability to measure the performance of running applications. Based on the application performance, `iRS` picks the ideal applications to expand and shrink the nodes. The granularity of the resource reconfiguration is node-level, i.e., as part of an expansion or reduction operation, a complete compute node is given or taken back.

`ABS` is involved in job submission and selecting the next job to schedule. This is not trivial since a submitted job can be rigid or malleable. `ABS` provides configuration parameters to users to mark a job as rigid or malleable. `ABS` has two priority-ordered queues to accommodate rigid and malleable jobs, which is not the case with the traditional batch scheduler. `ABS` employs different heuristics to select the next job to be launched from the queue. `ABS` has a global view of the system and can ask the `iRS` to adjust the compute resources of running malleable jobs to make sufficient resources for the job in the queue to be launched. Different strategies that favor rigid jobs, malleable jobs, or equally favoring both were provided in `ABS` (see section 4.2). `ABS` interaction with `iRS` is event-triggered and occurs ❶ after every scheduler tick seconds (configured during Slurm setup), ❷ during a new job submission, or ❸ on job completion.

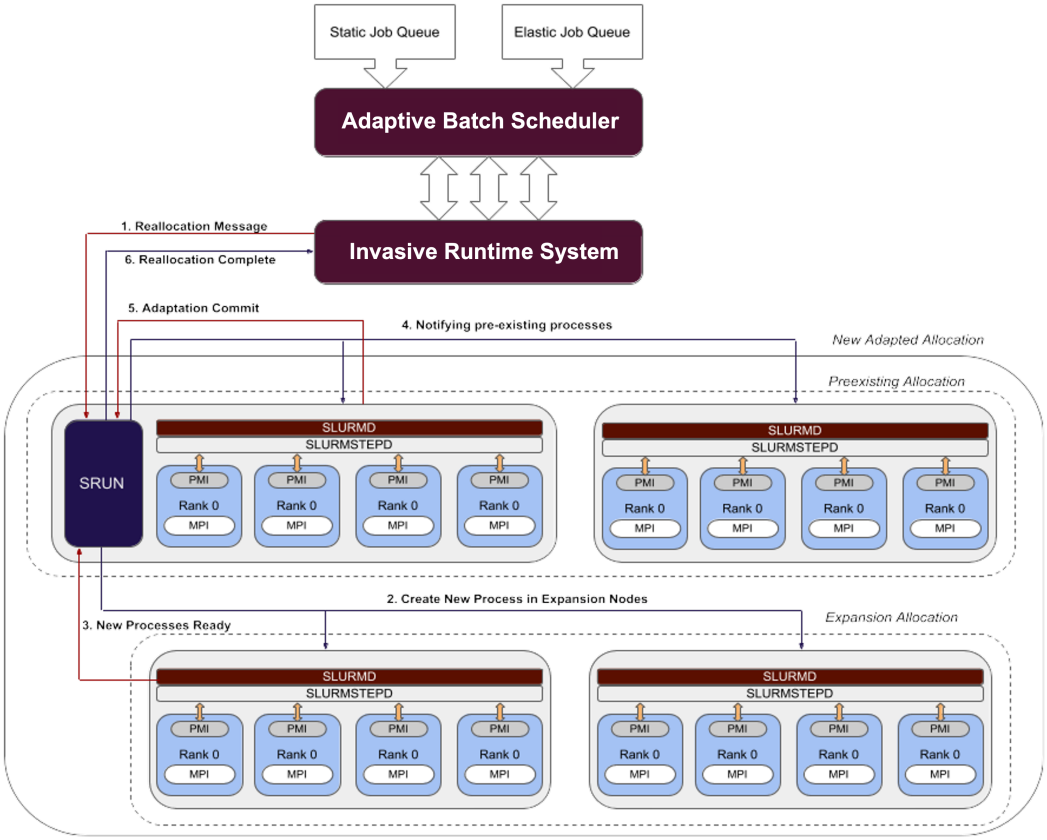


Figure 2.4: Invasive Resource Manager [88]

To realise `iRS` and `ABS`, new functionalities were added to different Slurm components. Towards that, `sbatch` is modified to recognize and support malleable job scheduling and submission (see Chapter 4). `srun`, `slurmd`, and `slurmstepd` were extended to support the expansion and reduction of compute resources in a running application.

2.2.1.3 Expansion and Reduction for Malleable Batch Jobs

Upon receiving resource reconfiguration information from `ABS`, `iRS` is responsible for performing the resource change while ensuring the consistency of the system state. As seen in the above section, the application launch is performed as a result of continuous interaction between `srun` and `slurmd`. A resource reallocation handler was added to `srun`, which is in charge of creating new processes during a resource expansion, terminating the processes during a resource reduction, and changing the context of a running job. `iRS` interacts with the resource allocation handler inside `srun` to trigger the resource adaptation. During the application launch, `iRS` is notified with the reallocation handler address of that application by `srun`. The communication is done via the modified RPC messages. `iRS` must constantly contact the reallocation handler to ensure resource adaptation since `iRS` can trigger the resource change for multiple applications simultaneously. All of the corresponding reallocation handlers will notify `iRS` about the adaptation progress. This is essential since `ABS` support the simultaneous expansion and reduction of multiple applications. To

efficiently utilize the resources, iRS first triggers resource reduction among selected jobs. Then the released resources are considered for the expansion of the selected jobs.

To support such a resource adaptation technique, the communication between iRS, reallocation handler, and srun is vital. It is done in a six-step process, as seen in the Figure 2.4. Following steps are performed for a **resource expansion** in iRM.

1. A reallocation message is sent from iRS to the reallocation handler in srun. This contains information about the type of operation, whether it is an expansion or reduction of resources.
2. For an expansion operation, srun informs the slurmd daemons in the node, which is to be used for resource expansion to start the new set of processes.
3. slurmd daemons prepare for the resource change by updating the metadata that applications will use to query the adaptation type, resource change, and other application-specific information like MPI ranks, communicator, and size. slurmd notifies the srun that the new processes are launched and ready for adaptation.
4. Upon receiving the message from slurmd, srun informs the slurmd of preexisting nodes about the resource change. slurmd updates the metadata of the application about the change in resources. As a result, the application in the preexisting nodes is aware of the newly launched processes of that application in the new nodes. The application can then perform tasks like data redistribution and load balancing.
5. slurmd informs srun about the completion of adaptation once it receives the adaptation finish information from the application.
6. srun informs iRS about the end of the resource change operation, and iRS updates the job's metadata to reflect the current job state.

Meanwhile, in the case of **resource reduction** using iRS, steps two and three can be avoided. In the fourth step, slurmd daemons, while updating the metadata of an application, also specify whether the current node will be removed from the application.

Using the dynamic resource management technique provided by iRS and ABS can significantly improve the job-level and system-level performance metrics like average waiting time, response time, makespan, and system utilization compared to the *backfill* scheduler [57] from the Slurm. The more malleable applications are present, the better the impact of iRS and ABS on performance. However, developing distributed malleable applications using the currently available programming models in HPC is not trivial. As a result, the D3 subproject [81] created an Invasive version of Message Passing Interface (MPI) called iMPI.

2.2.2 Invasive Message Passing Interface

MPI is the most commonly used distributed memory programming model in HPC systems [89]. There are different implementations of MPI from the academic and private sectors, like MPICH [684], OpenMPI [685], Intel MPI [90], IBM Spectrum MPI [91], and MVAPICH [686]. iMPI is an extension of the MPICH implementation of the MPI standard with custom features and functionalities to support dynamic process management.

2.2.2.1 iMPI Concepts

The two fundamental design objectives behind iMPI were to create a user-friendly paradigm for resource-aware programming and to reduce the latency of dynamic process creation in the standard MPI. Four new operations were added to the MPICH implementation towards realizing that objective. They are:

1. `int MPI_Init_adapt(int *argc, char **args, int *local_status);`

   `MPI_Init_adapt` notifies the invasive resource manager that the application is malleable. This function has similar signature to the standard `MPI_Init()` with the presence of an additional `local_status` parameter later used to identify the origin of the process, i.e., whether it was launched by the reallocation handler in iRS as part of resource reconfiguration or was by `srun` during an application launch. The value `MPI_ADAPT_STATUS_NEW` in the `local_status` parameter signals that the process was created during the application launch while `MPI_ADAPT_STATUS_JOINING` indicates that it was launched during a resource adaptation. Identifying the types of processes is essential because the newly joined process (with value `MPI_ADAPT_STATUS_JOINING`) should call the `MPI_Comm_adapt_begin` routine immediately to participate in the adaptation operation. In addition, preexisting processes should also trigger the `MPI_Comm_adapt_begin` function to take part in the resource adaptation.

2. `int MPI_Probe_adapt(int *pending_adaptation, int *local_status, MPI_info *info);`

   `MPI_Probe_adapt` function checks whether any resources are available for expansion or to be returned to the resource manager. After the probe function call, the argument `pending_adaptation` can either have the value `MPI_ADAPT_TRUE` indicating a pending adaptation or `MPI_ADAPT_FALSE` conveying no resource change. The parameter `local_status` specifies whether a process is staying, leaving, or joining by returning the values `MPI_STATUS_STAYING`, `MPI_ADAPT_STATUS_LEAVING` or `MPI_STATUS_JOINING`. For newly joining processes, these information can be obtained from the `MPI_Init_adapt` call.

3. ```
   int MPI_Comm_adapt_begin(MPI_Comm *intercomm, MPI_Comm *intracomm,
               int *stayingcount, int *leavingcount, int *joiningcount);
   ```

   This method call marks the beginning of an adaptation window. Newly joining processes must immediately call the `MPI_Comm_adapt_begin` function after the initialisation. It is a blocking collective function and waits for all the application processes to call this operation. Preexisting processes should call the `MPI_Comm_adapt_begin` function to begin an adaptation window if the `local_status` from the `MPI_Probe_adapt` call is leaving or staying. Preexisting processes are informed about the adaptation once all of the newly joined processes have executed the `MPI_Comm_adapt_begin` function. This operation creates two new communicators `intercomm` and `intracomm`. A communicator is an MPI object that represent a set of processes inside an MPI application. The communicator `MPI_COMM_WORLD` includes all MPI processes. The `intercomm` communicator returned by the `MPI_Comm_adapt_begin` call comprises all processes created as part of a resource expansion or reduction in an application. The `intracomm` contains all processes in an application and is different based on the type of adaptation operation. It includes joining and preexisting processes during an expansion operation and only staying processes during a reduction operation.

4. `int MPI_Comm_adapt_commit( );`

   This routine marks the end of an adaptation operation inside iMPI. After all application processes complete this function call, the `intracomm` communicator will be the new `MPI_COMM_WORLD`. All processes
   ```

having local status value of `MPI_ADAPT_STATUS_JOINING` or `MPI_STATUS_STAYING` will remain after the resource adaptation, and processes with `MPI_ADAPT_STATUS_LEAVING` status will be terminated at the end of the adaptation window.

The adaptation window (created by the functions `MPI_Comm_adapt_begin` and `MPI_Comm_adapt_commit`) can also be used for data redistribution and load balancing inside an application. The communicators created by the `MPI_Comm_adapt_begin` routine give insights into the new process configuration and can be leveraged for load balancing and data redistribution.

2.2.2.2 Writing a Simple Malleable MPI Program

MPI programs typically follow a Single Program Multiple Data (SPMD) model where multiple instances of the same program are launched in parallel, working on the same or different data to reach a solution. From a technical perspective, the resource manager (or an MPI launcher) launches the same application binary (MPI program) multiple times simultaneously across different cores in same or different nodes (as part of a job allocation). Once a launcher launches multiple instances of these programs, they can use MPI APIs to communicate with each other. `MPI_Init` is the first MPI call by a standard MPI program (MPI sessions do not need MPI Init call [92]), and it initialises the library (for example, prepares modules for communication, assigns a unique ID to each process and is referred to as `rank`). MPI programs are typically C, C++, Fortran, or Python programs using the message passing API to communicate with each other.

Consider parallelizing a sample sequential application that uses a Jacobi Kernel to calculate the 2D heat transfer process on a metal plate. For simplicity, the sequential application can be broken down into three parts. ❶ There is an initialisation part where the 2D grid (abstraction of a metal plate) is defined, and data for computation is initialised (populate the 2D grids) based on some initial settings. ❷ Then there is a compute part where the Jacobi iterative solver resides and solves the equation iteratively until it converges. ❸ In the last part (or finalization part), the results from the computation are displayed.

Each part of the sequential application must be extensively modified to parallelize such an application in MPI. ❶ In the initialisation part, the 2D grid should be defined across multiple processes and a virtual topology of processes must be created to distribute the 2D grid. Following that, the grid initialisation should be performed locally among each process. ❷ In the compute part, each process performs the stencil computation iteratively on its local data. Since it is a 2D grid and data is distributed locally among all the processes, after each iteration, the local data along the border of each process in the grid should be exchanged (ghost cell exchange) with neighbor processes to ensure correctness. The data exchange and iterative computation are performed until the application converges. ❸ In the finalization part, the primary process will aggregate the results from the participating processes and displays them.

Converting the above rigid MPI application into a malleable application can be more demanding than converting the sequential version of the same application into a parallel one. The concept of newly joined processes (created during an expansion operation) and the data required by them, outgoing processes (leaves during a reduction operation) and the data required from them, the transfer of data among new and preexisting processes, and the creation of a new topology with the new process configuration make the application transformation challenging. The first task in a malleable `iMPI` application is determining the entry point for new processes, followed by the data redistribution. For that, the application should probe periodically for a resource change. If a resource change is observed and new processes are launched, the corresponding `iMPI` routines should be called to start the adaptation window. Data redistribution can be performed during this window.

This is not trivial since multiple control statements are necessary to differentiate new and old processes during a resource change. A program can have two different control flows, one intended for joining processes

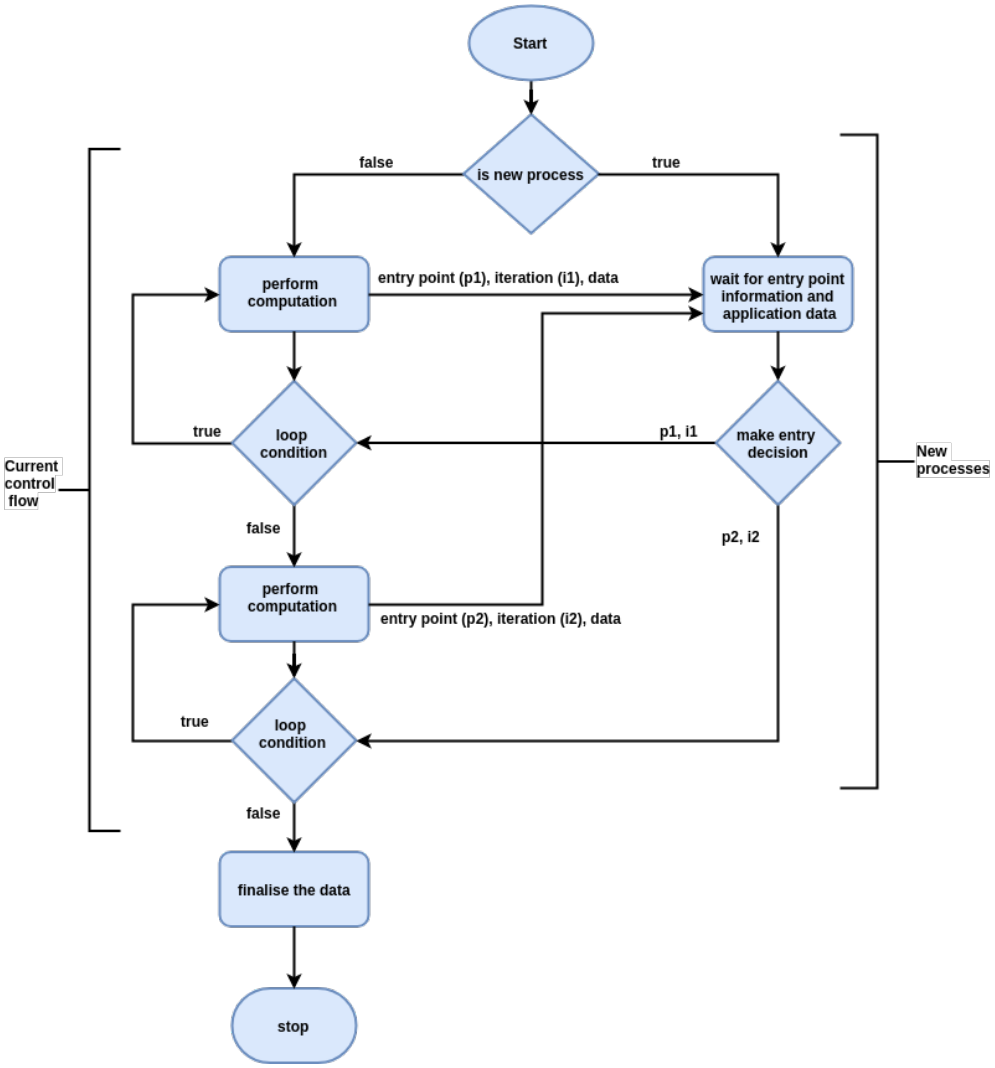


Figure 2.5: Control flow of a sample iMPI application [88]

(during a resource expansion) and the other for preexisting processes, and these control flows will be merged typically at the end of an adaptation. Additionally, newly joined processes should avoid the execution of certain parts of the program to avoid logical errors. The execution of corresponding `iMPI` routine completes a resource adaptation, and newly joined processes, using the entry point, reaches the common control flow and resumes the normal execution of the program. Figure 2.5 visualises the merging of new processes to a pre-existing control flow. Similarly, during a reduction operation, it is essential to find out the exit point from which the processes can be removed gracefully, and the data from the leaving processes should be redistributed.

As a result, the initialisation part of malleable MPI needs to identify the process types and perform the grid creation based on it. Newly joining processes should not create a 2D grid in the initialisation part since the new MPI topology will be calculated later. In the compute part of the application, during an adaptation, the new process topology should be created with the new resource configuration, and the newly added processes must wait for the entry point information (iteration number) from preexisting processes to start their execution from the proper iteration. In addition, the local data required for the stencil computation must also be sent to newly joined processes. The finalization part is similar to a rigid MPI implementation.

```
1  #include<mpi.h>
2  int main() {
3    /* Initialisation part of the application */
4    MPI_Init_adapt(..., type) /* iMPI is intialised */
5    if (type == joining) {
6      /* Start of adaptation window in joining processes */
7      MPI_Comm_adapt_begin(...);
8      /* Get metadata from preexisting processes, perform data distribution */
9      MPI_Comm_adapt_commit();
10   }
11   /* Iterate over compute intensive phase of the application */
12   while (true){
13     /* iMPI processes must call probe periodically */
14     MPI_Probe_adapt(resource_change,...);
15     if (resource_change) {
16       /* Start of adaptation window in preexisting processes */
17       MPI_Comm_adapt_begin(...);
18       /* Pass metadata to joining processes, perform data distribution. */
19       MPI_Comm_adapt_commit();
20     }
21     /* Compute part of the application */
22   }
23   /* Finalisation part of the application */
24   MPI_Finalize();
25 }
26
```

Listing 2.2: Pseudocode of a sample iMPI application.

The pseudocode of such a simple malleable `iMPI` application is shown in Listing 2.2. The four `iMPI` specific routines act as follows. `MPI_Init_adapt()` (in line 4) will notify the malleable nature of the application with the resource manager and returns the type of the calling `iMPI` process. A process can be initial (started during the application launch) or joining (launched during the expansion operation). Initial processes continue the application execution (skipping lines 5-10) and regularly call `MPI_Probe_adapt()` (line 14) to check for any resource change. However, joining processes immediately triggers the collective `MPI_Comm_adapt_begin()` (line 7) function and waits for the initial group of processes (preexisting processes) to join. Meanwhile, the `MPI_Probe_adapt()` (in line 14) call informs initial processes about the resource change triggered by the malleable resource manager. As a result, they call the `MPI_Comm_adapt_begin()` (line 17). Once all the application processes (initial and joining) completes `MPI_Comm_adapt_begin()` (in lines 7 and 17), they can begin redistribution of data, share process-specific data (for example, data only needed by a lead rank) and additional control information among them. Then, they execute `MPI_Comm_adapt_commit()` (lines 9 and 19) method to finalise the resource adaptation and together continue the execution of the application.

In short, even for a simple malleable application, the complex control flow of the joining and preexisting processes, data redistribution needed during the resource change, and synchronisation among processes will become necessary. These challenges encountered during the invasive application development led to the creation of the Elastic Phase Oriented Programming model.

2.2.2.3 Elastic Phase Oriented Programming Model

The Elastic Phase Oriented Programming Model (EPOP) considers an application program as a collection of multiple phases with different characteristics. Intuitively, a typical application has a phase where initialisation and data distribution occur, a phase where computations happen, and a phase where writing results are performed. Typically, in a parallel program, the performance of the application is inhibited by the sequential part of the program (Amdahl's law [93]) that cannot be parallelised (or parallelisation does not have any impact). In the EPOP model, such parallelism-inhibiting parts are referred as rigid phases, and highly parallelisable parts are considered as elastic phases.

EPOP model has a simple syntax that enables the application to be split into different *phases* and the control flow can be defined. The *Init*, *Rigid*, *Elastic*, and *Branch* phases are the four available phases in EPOP. These phases can be used to label the computational parts of an application as rigid or malleable. An application can be written as a collection of phases using the EPOP API (Listing 2.3), and a vector of these phases (phase vector) is created during the compilation. During the application execution, the driver component of the EPOP cycles through the phase vector and invokes the phases according to the phase type and declaration order (Lines 18-19).

The *init* phase marks the beginning of every EPOP program, and the code for the application initialisation must be defined in this phase. The *init* phase will only be called once by the EPOP driver. *Elastic* phases represents the computationally intensive part of the application that benefits from resource change. There can be multiple *elastic* phases in an application, and each of them should contain ❶ a computational code block, ❷ an iterator logic that determines the number of code block execution, and ❸ a resource change (adapt) block. The resource change block is called only during a resource adaptation. The *rigid* phase defines the block of code that doesn't yield any benefit with a resource change. An ideal candidate is the code associated with application finalisation. The main purpose of the *branch* phase in an application is to alter its control flow. Typically, an EPOP driver iterates through different phases in the order of phase creation, and a *branch* phase can be leveraged to jump from one phase to another (see Figure 2.6).

```
1  void init_block(...){
2    /* Initialisation part of the application */
3  }
4  void elastic_block(...){
5    /* Compute part of the application */
6  }
7  bool iterator (...){
8    /* Iterate over compute part of the application */
9  }
10 void adapt(...){
11   /* Start of adaptation window */
12   /* Data redistribution*/
13 }
14 void finalise_block(...){
15   /* Finalisation part of the application */
16 }
17 extern "C" phase_vector epop_program(){
18   setInit(init_block);
19   setElastic(elastic_block, iterator, adapt);
20   setRigid(finalise_block,...);
21 }
22
```

Listing 2.3: Pseudocode of a sample EPOP application [59].

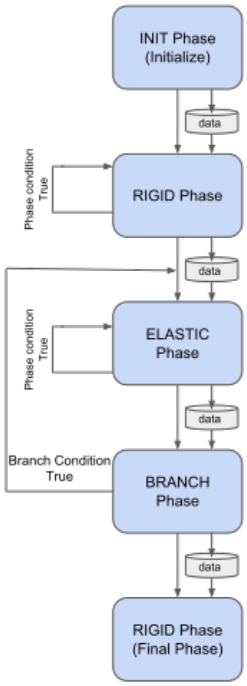


Figure 2.6: Control flow of a sample EPOP application [88]

The `EPOP` driver starts by loading the supplied `EPOP` program, then it begins iterating through various phases, executing each based on the specified phase conditions, and facilitates the transfer of application data between these phases. The driver will call `iMPI` routines (`MPI_Init_adapt`, `MPI_Probe_adapt`, `MPI_Comm_adapt_begin` or `MPI_Comm_adapt_commit`) internally for resource adaptation during the execution of the program and also calls the adaptation block at appropriate points.

Following steps can be used to convert the sample heat transfer application defined in Subsection 2.2.2.2 into the `EPOP` model. First, an *init* phase can be used to mark the initialisation part of the application. Followed by an *elastic* phase to mark the compute intensive part. Lastly, a *rigid* phase to mark the finalization part. Listing 2.3 shows the `EPOP` equivalent of the `iMPI` application (in Listing 2.2). As can be seen in Listing 2.3, the initialisation, compute, and finalization parts are written as `init_block` (Line 1), `elastic_block` (Line 4), and `finalize_block` (Line 14), respectively. The `EPOP` helper functions `setInit()`, `setElastic()`, and `setRigid()` (Lines 18-20 in Listing 2.3) record the function blocks as *phases*. A phase vector created during the compilation contains metadata about all declared phases in an application. This information will be used by `EPOP` driver for application execution.

2.3 Building Upon and Leveraging Invasive Infrastructure

The dynamic resource management infrastructure introduced in Section 2.2 with its tools for resource dynamism (`iRM`) and application malleability (`iMPI, EPOP`) were utilised as a foundation for this thesis work. The contributions from this work were defined coarsely in Section 1.3. A more fine-grained description of the contribution is as follows:

1. Extended `iRS` and `srun` to add support for `sbatch`. Created a new adaptive batch scheduler (`ABS` with `sbatch` support) with a performance-aware algorithm that improves system utilisation compared to the backfill scheduler from Slurm (Chapter 4).
2. Extended `ABS` for power corridor management (Chapter 5) by creating a new power-aware scheduler. Leverages `EPOP` driver for power measurements.
3. Created a self-adaptive checkpoint system (iCheck) that dynamically performs resource management and supports dynamic applications written using `iMPI` (Chapter 6).
4. Extended `ABS` to enable iCheck to dynamically grow or shrink its resources (Chapter 7).
5. Demonstrated that iCheck can be used to perform data redistribution during resource adaptation in `iMPI` application (Chapter 7).

3

Related Work

This chapter gives an overview of related work in the order of contributions defined in this thesis. The first section provides information about related research in the area of RJMS compared to the contribution of this work. The second section discusses research works associated with power corridor management. The last section presents insights into related works performed in the field of checkpointing.

3.1 Adaptive Batch Scheduling

To effectively schedule and manage adaptive applications, three crucial components must be taken into account: ❶ an adaptive parallel runtime system, ❷ an adaptive job scheduler, and ❸ an adaptive runtime scheduler. Extensive research has been carried out in all of the above three areas, with various researches developing systems that have adaptive runtime capabilities, scheduling strategies for resource-aware applications, and even enhancing existing batch systems for dynamic management of resources. Below paragraphs will delve into the relevant work in each of these areas.

One approach for supporting malleability in MPI applications is through the use of Adaptive MPI (AMPI) [94–96] and Charm++ [75, 97, 98]. To provide malleability, these systems utilize virtual MPI processes and automatic load balancing [99] features. The applications running on the system receive periodic updates regarding the availability of resources from the scheduler. Based on this information, the applications are reconfigured using object migration from Charm++ to a different count of virtual MPI processes. This virtual MPI process migration is leveraged for automatic load balancing in Charm++ [99]. Additionally, for fault tolerance, the AMPI runtime provides automated checkpoint and restart mechanisms [100]. However, these systems do not support newer features introduced in MPI-3.1 [687]. Another system, Invasive MPI (iMPI), follows the current MPI execution model and does not utilize virtual MPI processes [101, 102]. It achieves resource elasticity through its adaptive runtime system. In this work, iMPI is utilized for bringing malleability.

Flex-MPI [103] and ReSHAPE [104] are other examples of adaptive runtime frameworks. These frameworks focus on dynamic resource reconfiguration for iterative Single Program Multiple Data (SPMD) MPI applications by considering their performance characteristics. Flex-MPI is built on top of MPICH [684] and uses application-specific performance data to make decisions about application reconfiguration. It uses

a computational forecast model developed with the aid of Performance Monitoring Counters (PMCs) and network data. Additionally, support for various communication patterns (irregular and regular patterns) in parallel applications is provided in Flex-MPI. In ReSHAPE, identical iterations are assumed in computation and communication aspects of all applications. Unlike Flex-MPI, it uses expensive and comprehensive approach to decide about resource reconfiguration.

The efficient resource utilisation and parallel application scheduling is an indispensable research domain in HPC and computer science for many years [105, 106]. Its significance in HPC arises from the fact that the throughput of a system directly depends on the type of jobs it schedules and executes. Feitelson and Rudolph [107] classify jobs into four categories based on the flexibility of the jobs. Rigid jobs need the same amount of resources throughout their execution and are the most common type. Moldable jobs allow for resource requirements to be altered by a batch system before execution but remain fixed during runtime. Evolving jobs request resource allocation changes during execution, while malleable jobs allow the batch system to trigger resource changes. Applications like Adaptive Mesh Refinement [108] showcases load imbalances as a result of varying characteristics of it's computational phases and can be considered in both evolving and malleable job categories. This work primarily focuses on scheduling malleable and rigid jobs. Additionally, the moldability of the jobs is also considered in this work.

When it comes to improving system utilization and reducing response and makespan times, executing malleable jobs on modern HPC systems can be highly effective. Gupta et al. [109] and Hungershofer [110] demonstrated this in their research. To make scheduling malleable applications more efficient, several strategies have been proposed. Carroll et al. [111] recommend an incentive-compatible online scheduling technique that aims to reduce response time by assigning resources to jobs in a way that meets hard deadlines. During job submission, users provide additional information such as arrival time and deadline and are given incentives if their jobs are completed by the specified hard deadline. Sun et al. [112] propose a fair and efficient scheduling approach for malleable applications. They use the equipartitioning algorithm to achieve fairness, which divides the available processors equally among running applications. In addition, they also employ a feedback-driven adaptive scheduler that considers the executing jobs' history for efficiency. These approaches results in better response times and system utilisation in comparison to a rigid scheduler. However, it is essential to mention that the strategies in references [111] and [112] focus on the theoretical facets of scheduling, and their evaluation has been confined to simulations.

In contrast to these theoretical approaches, there are practical implementations that integrate an adaptive batch system with a dynamic job scheduler. For example, Utrera et al. [113] introduced a job scheduling strategy that employs virtual malleability (VM). VM preserves the initial process count while enabling the job to adapt to variations in CPU availability during runtime. MPI and the interposition mechanism are used to implement the VM concept. They offer a malleable job scheduling policy based on First Come First Serve (FCFS). An event-driven algorithm is triggered upon a job arrival and job exit, and the resources are assigned to jobs in the order of earliest-started-job-first. TCP/IP socket communication is used to communicate between the job scheduler and the VM library. The authors proved that FCFS's malleable job scheduling attains a remarkable 31% improvement in average response time in comparison to the popular EASY backfilling strategy when a cluster consists solely of malleable jobs.

The Torque/Maui batch system was extended by Prabhakaran et al. [114] to improve resource management and allocation for evolving jobs. Their approach promotes fairness between evolving and rigid jobs, resulting in reduced turnaround and waiting times, increased system utilization, and improved throughput. Additionally, they expanded the Torque/Maui batch system to support resource expansion and reduction performed by malleable jobs [115]. A communication protocol with the Charm++ runtime has been proposed to support the malleability of applications. They have also proposed a Dependency-based Expand Shrink (DBES) scheduling algorithm that combines the scheduling of rigid, evolving and malleable jobs. It uses

backfilling strategy and dependency analysis for scheduling efficiently across changing workload dynamics. It also employs the equipartitioning strategy to ensure fairness in providing idle resources to different jobs after the scheduling stages. The approach was evaluated and demonstrated for the modified Effective System Performance Benchmark (ESP) [116] using the metrics average response time and system utilisation. The DBES scheduling strategy performs better than other strategies for varying numbers of malleable jobs in the workload. However, DBES does not account for the performance of ongoing applications when making resource reallocation decisions. Apart from the proposed work, it represents the sole comprehensive batch system detailed in the literature that integrates a dynamic batch system with an adaptive parallel runtime.

In the context of scheduling malleable applications within GRID systems, Buisson et al. [117] proposed an adapted equipartition strategy alongside a Favour Previously Started Malleable Applications (FPSMA) scheduling policy, which was assessed using the KOALA [118] multicluster grid scheduler. In the proposed work, these scheduling policies for malleable jobs were extended with additional constraints based on the number of nodes and the application characteristics. Additionally, these policies were extended to take performance-aware dynamic resource reconfiguration decisions for running malleable applications. The proposed system in this work was evaluated on a Supercomputer.

3.2 Power Corridor Management

The enormous energy requirements of powerful supercomputers have forced the HPC community to consider the power usage along with the performance aspect for consideration while building new HPC systems [119]. In addition, the rising energy costs [120] lead to research on power usage as a metric for optimising the applications and systems. Typically the energy contract negotiated by the supercomputing centres with the energy providers requires these centres to maintain the power usage within the stipulated bounds specified in the corresponding contracts. Failure to adhere to the stipulated bounds might incur financial penalties to the centres. Furthermore, it is necessary for these centres to maintain the power usage of the machines to prevent issues like overheating and causing damage to the systems. As a result, these supercomputing centres employ various techniques to keep the power usage in check [121]. These employed strategies primarily rely on the techniques such as dynamic power management (DPM), dynamic voltage and frequency scaling (DVFS), and power capping [122].

DPM is used to turn off the idle nodes or lower the processor frequency [123]. DVFS, as the name suggests, can be used to control power usage by regulating the voltage and frequency of the device [124]. Power models of the systems are utilised to ensure the accuracy of the DVFS [125]. Power capping is introduced to overcome the limitations in the DVFS approach [122, 126]. A power threshold is assigned for a particular device in the technique of powercapping [127] using the tools provided by the device manufacturers [128]. For example, Intel provides Running Average Power Limit (RAPL) [129], NVIDIA provides NVIDIA System Management Interface (nvidia-smi) [130] and IBM provides Energyscale [131]. These techniques are combined to manage the power consumption in a device and can be applied statically or dynamically.

From the engineering perspective, different centres combine different approaches for resolving their power consumption problem. One frequently used technique is dynamically stopping the jobs whenever a power budget is reached [121]. In another approach, the energy-aware scheduler considers the energy efficiency of past runs of the job for potential resource allocation. There is also a technique where intelligent energy-aware backfilling algorithm and adaptive powering down of nodes [132] are used to control the total power usage [133]. In some techniques, the idle nodes are powered down to satisfy the power usage requirements of the centres[134]. In addition, these centres utilise the power capping tools from the hardware vendors to force the system to operate at certain selected frequencies. Plenty of works combine the power capping

and DVFS to reduce the power consumption of the system [135–137]. Furthermore, Bodas [134] proposed a power-aware scheduling approach that applies a uniform frequency mechanism where a power budget is distributed to the job, and the scheduler ensures the operation of system in the power corridor.

A critical difference between the work proposed in this thesis and the above techniques is that this work leverages dynamic resource management for power corridor management. The applications' power usage characteristics are modelled and utilised for resource redistribution to maintain the system level power corridor requirements. Also, most of these systems only use a reactive approach where they perform various techniques to bring the system back into the corridor only if it is out of the power corridor. In contrast, we also use a proactive approach where resource adaptations are performed based on the power usage predictions. In addition, the power-aware scheduler proposed in this work also employs application moldability to schedule the jobs. For example, the scheduler launches the job with fewer resources than requested if it sees that the application will violate the system's power corridor if it is launched with the requested resources. Furthermore, our system is capable of adjusting to fluctuating power budget demands, ensuring it can adaptively respond to dynamic power corridor changes.

From the algorithmic side, there are plethora of energy-aware scheduling algorithms that focus on power usage on CPUs and GPUs [138–140]. Specialised techniques using randomized algorithms [141], fuzzy logic [142], integer programming [143], dynamic programming [138], evolutionary algorithms [144–146], constraint programming [147] and machine learning [147] were developed to model and control the power usage. The primary objective of these algorithms is to balance the application performance while maintaining power efficiency. However, this work's core focus is enforcing the system-level power usage, and as a result application's performance is not considered. In addition, the proposed work can also be used to balance the power usage of applications during peak hours. For example, by dynamically changing the power corridor in the proposed system, the applications are reconfigured by the runtime system to bring the overall power usage into the new limit. To bring down the power usage, the power-aware scheduler will only schedule low-power jobs and queue the high-power jobs, ensuring the system remains within the specified limit. A similar work [42] proposed by Yang manipulates the job queue so that the power usage can be adjusted by queuing the jobs to reduce the energy price in peak hours. However, it only employs job queuing to achieve the objective and does not consider application malleability. Mammela et al. [148] used DPM mechanisms in addition to turning off compute nodes in the popular scheduling algorithms like First In First Out (FIFO), backfill first (BFF) and backfill best fit (BBF) to make it energy aware. E-FIFO, E-BFF, and E-BBF were the corresponding energy-aware versions of the algorithms. In contrast to the proposed work, these algorithms cannot react to the dynamic energy requirements of the system since their primary objective is to reduce energy usage. In the proposed work, the power-aware scheduler can hold the jobs and bring power usage to the desired limit.

Furthermore, algorithms like energy-aware task scheduling algorithm (EAMM) [149], Min-Min [150], min-energy-max-execution (MEME) [151], energy-aware scheduling by minimising duplication (EAMD) [152], energy-aware forward list scheduling (eFLS) [153], EDL [154], EMRSA [155], energy-aware service level agreement (EASLA) [156], energy dynamic level scheduling (EDLS) [157], and power-aware algorithm for scheduling (PAAS) [158] focus on maximizing energy savings. These algorithms are explicitly tuned for maximizing energy efficiency and reducing power usage, considering various metrics like execution time, energy consumption, and max and min execution time of the jobs. The power models are created, and the jobs are scheduled to minimise energy usage. The similarity with the proposed works is that the proposed work also uses energy information for job scheduling. However, the energy information is used along with the job moldability to make scheduling decisions which are not covered in these works. In addition, the scheduler dynamically reconfigures the jobs to maintain, reduce or increase the power usage; meanwhile, the above-listed works are used primarily to reduce the power usage. Nevertheless, job malleability is not covered in any of the works mentioned above, and is unique to this work.

3.3 Checkpointing

The race for exascale machines is in full force [18–20], and the road to zettascale is already getting paved [159] in the HPC community. The component complexity associated with the exascale is massive and is expected to only increase along the road. However, the component's reliability is not increasing at the same rate as the complexity of the components [160]. As a result, resilience to failure has become a multidimensional problem, and the solution spans domains like hardware and software, applications and system software. The field of fault tolerance approaches is extensively studied and discussed heavily in the literature [160–164]. There are numerous fault tolerance techniques based on the concepts of redundancy (hardware and software), migration methods, failure semantics, failure masking and recovery. Each of these techniques is described below:

- **Redundancy**: In this technique, the redundant components or processes are added to tolerate the failures [165, 166]. In hardware redundancy, the critical components are replicated, while in software redundancy, multiple instances/versions of the software are executed [164].
- **Migration**: The advancement in virtualization technologies is exploited for process-level migration [167] and virtual machine migration [168] to address the failures. This technique's fundamental premise is to avoid failure by taking preventive action by migrating the application to a safe node.
- **Failure Semantics**: This technique [169] is about how a system designer anticipates failure and provides the failure handling strategies to thwart the anticipated failures. This strategy is tailored to the system, and predefined recovery action is taken when the anticipated failure occurs.
- **Failure Masking**: It ensures the resilience of a system by providing services to the clients using redundant services without the client recognising that a failure has occurred [170].
- **Recovery**: As the name suggests, the application is recovered from the failures in the recovery technique. Here the erroneous state of the application is replaced by a valid state. There are two categories of error recovery techniques based on their characteristics. They are ForwardError Recovery and Rollback Recovery.

  1. **ForwardError Recovery**: The system is brought to a valid state without repeating the previous computations executed in this technique. This strategy is used in mission-critical environments where high levels of accuracy can be sacrificed for immediate recovery. This technique is not useful with a highly complex software system with many valid states [171].
  2. **Rollback Recovery**: Rollback recovery comprises of three parts ❶ checkpoint, ❷ failure detection and ❸ recovery/restart. A checkpoint contains the snapshot necessary to restart the application in a valid state when a failure occurs. Rollback recovery is one of the widely used fault tolerance mechanisms in HPC Systems and can be attributed to the use of distributed memory programming models in creating applications [171]. Most applications are written using MPI, and MPI implementations do not have inherent fault tolerance capability. As a result, the failure in a single process can cascade to the entire application. In such instances, rollback recovery can be used to recover the application from the previously saved state. There are two techniques to implement the rollback recovery mechanism. They are log-based rollback recovery and checkpoint-based rollback recovery.

     – **log-based**: In the log-based recovery technique, the messages transferred across processes are logged and used in case of failure to recover the application. The messages are replayed to construct a valid state.

– **checkpoint-based**: The application is reverted to the most recent stable state by leveraging the checkpointed data in the checkpoint-based recovery technique. The checkpoint-based rollback recovery is one of the widely popular techniques used in HPC and the proposed work also utilises the checkpoint-based rollback recovery technique.

The checkpoint-based rollback recovery mechanism is called the checkpoint restart or C/R technique, or simply checkpointing. There are different techniques and types in the checkpoint restart strategy. Based on the implementation technique, the checkpointing systems can be classified as ❶ application-level, ❷ user-level, or ❸ system-level [162, 172]. In application-level checkpointing, the programmer or pre-processor injects the checkpointing blocks into the relevant parts of the application. In user-level checkpointing, user-level libraries are used to checkpoint the application. It is semi-transparent compared to application-level checkpointing because the call to the user-level library will save the application's state. However, in application-level checkpointing, the programmer is in charge of checkpointing every relevant data needed in the event of restart from failure. Nevertheless, user-level checkpointing can be considered conceptually synonymous with the application checkpointing technique since the application programmer needs to play a role in employing the checkpointing technique. On the contrary, the system-level checkpointing is fully transparent to the user, and the OS or resource manager will checkpoint the entire process state that can be used in the event of a failure.

Checkpointing can be further classified as single-level or multilevel, coordinated or uncoordinated, blocking or non-blocking and process recovery or data recovery[162, 172]. They are summarized below:

- **single-level vs multilevel**: The primary difference between single-level and multilevel is the number of memory hierarchies used to store the checkpoint. In a single level, the checkpoint is stored in only one place (for example, HDD). In multilevel, the checkpoints are stored in multiple storage hierarchies (for example, RAM and HDD).
- **coordinated vs uncoordinated**: In coordinated checkpointing, all the processes store the checkpoint in a synchronized manner, while in uncoordinated checkpointing, the processes checkpoint the data independently. Different processes have different versions of the data in uncoordinated checkpointing.
- **blocking vs non-blocking**: In the blocking technique, the process waits until the data is completely checkpointed. In contrast, the process resumes the execution immediately after calling the checkpoint library in the non-blocking technique. The checkpointing happens in the background.
- **process recovery vs data recovery**: In process recovery, a new process is started immediately to replace the process and resumes the application execution, while in data recovery, a single process failure kills the entire application. As a result, new set of processes are launched, and the data is restored from the checkpoint to continue the operation. To support the process recovery, special fault tolerant MPI needs to be used [60].

Considering various use cases and programming models, the classification extends beyond the above mentioned categories, including additional classes such as fault-tolerant MPI implementations. Given the diversity of use cases and programming models, the checkpointing classification encompasses additional types beyond the above-mentioned categories, such as fault-tolerant MPI implementations. There is an extensive array of checkpointing research that covers these various categories. [60–63, 66, 160–163, 173–188]. Here we only consider works most closely related to iCheck.

iCheck is a ***multilevel asynchronous adaptive application-level*** checkpointing system with remote direct memory access capability that provides fault tolerance together with the data distribution service. Compared with iCheck, the existing application-level checkpointing libraries (for example, [186–188]) cannot readjust their resources across different applications to optimise and improve their checkpointing activity.

iCheck can globally control checkpoint transfers and enhance the overall checkpoint performance by scaling the checkpointing resources associated with the application. The following three paragraphs compare the most relevant related work to iCheck based on the features mentioned above (***multilevel asynchronous adaptive application-level***). The first paragraph covers the related work closest to ***multilevel asynchronous*** checkpointing capability. The second paragraph compares the ***adaptive*** aspect of the application level checkpointing. Lastly, the latest ***application-level*** checkpointing systems are compared.

The closest work to our system in the area of ***multilevel asynchronous*** checkpointing capability is by Sato et al. [66]. This work proposes an RDMA-based, non-blocking, multilevel checkpointing system that incorporates staging nodes (remote nodes). Within the staging nodes, a staging server acquires the checkpoint from a staging client, which operates within the compute nodes of the workload. There are fundamental differences between iCheck and [66]. Firstly, iCheck is an adaptive system that dynamically increases or decreases the iCheck nodes (equivalent to staging nodes). Moreover, iCheck offers the flexibility to dynamically adjust its agents (analogous to staging servers) based on the needs at hand. Secondly, In contrast to the system described by Sato et al., a single iCheck node can provide checkpointing services to multiple applications concurrently, a feature not demonstrated in the referenced work. Lastly, iCheck eliminates the necessity for a separate staging client, as applications directly register their memory regions with iCheck agents on remote nodes for RDMA operations. This allows an application to notify the agents through a straightforward library call when a checkpoint is ready, preventing the need for an additional thread on the compute node dedicated to staging.

The checkpointing system VeloC [185] has similarities to iCheck with respect to its ***adaptive*** qualities in application-level checkpointing.VeloC functions as an asynchronous checkpointing system and selects the storage locations for local checkpoints dynamically by utilizing various underlying heterogeneous storage options. This approach is designed to circumvent potential bottlenecks in local I/O, enhancing efficiency and performance. In contrast, iCheck's adaptivity pertains to the system's resource dynamism employed by horizontal (dynamically adding more checkpoint nodes) and vertical scaling (dynamically adding more agents for checkpointing). VeloC, on the other hand, utilises IO threads within nodes to write checkpoint data into the desired storage solution.

Lastly, two latest application-level checkpointing libraries (FENIX [187] and CRAFT [186]) and a state-of-the-art application-level checkpointing library Scalable Checkpoint Restart (SCR [188]) are compared with iCheck. FENIX and CRAFT differ fundamentally from iCheck in their checkpointing approaches: FENIX performs checkpointing on neighbouring application nodes, whereas CRAFT is oriented towards storing checkpoints directly into the parallel file system. SCR stores checkpoints in both in-memory (or neighbouring nodes) for faster restores and writes to the parallel file system (PFS) based on the user's configuration. Like SCR, iCheck also writes the checkpoint into PFS, making it a multilevel checkpointing system. Nevertheless, the iCheck system differentiates itself by being adaptive and conducting RDMA-based in-memory checkpointing, utilising reconfigurable threads (agents) on remote dedicated compute nodes. This is a deviation from the methodologies employed in the above-mentioned related works. Additionally, iCheck agents perform second-level checkpointing to parallel file systems from these remote nodes instead of utilsing the application nodes (as done in SCR). This significantly reduces the overhead linked with file system transfers from the application.

Another unique aspect of iCheck is the integration with an adaptive resource manager (mentioned in the chapter 7.4.4). As a result, iCheck can get more resources (compute nodes) from the resource manager and scale its resources as per the application's need. This resource scaling ability makes iCheck unique compared to other checkpointing systems integrated with resource manager [173, 174, 188]. These systems use the resource manager to coordinate the checkpointing activities. For example, Berkeley Lab Checkpoint Restart (BLCR [174]) is integrated with Slurm for providing transparent system-level checkpointing activities, while SCR [188] uses the resource manager to relaunch the application whenever the application fails.

In addition, iCheck also provides a data redistribution library for malleable applications. iCheck posits that the checkpointing system can also act as a data redistribution service for malleable MPI applications during the resource change. Though there are many data distribution libraries like LAIK [189], ESPRESO [190], and Hitmap [191], iCheck demonstrates the proof-of-concept that the already available checkpointing system can be used for data redistribution. Since checkpointing systems store relevant data needed to restart the application in case of failure, it can be extended to support the data redistribution for a malleable application.

4

Adaptive Batch Scheduling

This chapter describes in detail the first contribution of this work, an adaptive batch scheduling system. At first, an overview of the functioning of the proposed batch scheduler is provided, followed by different strategies employed to perform the resource adaptation. Lastly, the chapter describes the performance-aware batch scheduler produced as part of this work.

As seen in Section 2.2.1.1, the Slurm workload manager was designed only to support the execution of rigid applications. Ureña [101] et al. brought malleability support into Slurm by creating a runtime system that can expand and shrink the running `iMPI` applications based on their resource management policy. The default Slurm Controller `slurmctld` was modified to create a new Invasive Runtime Scheduler (`iRS`). This also required the modification of other Slurm binaries that were relevant to support the dynamic resource reconfiguration. However, the system from Ureña [101] only supported using interactive `srun` instances to bring the resource elasticity. As a result, the system was not equipped to handle and differentiate the incoming jobs based on their resource elastic characteristics (see Section 2.2.1). The system could only recognise the application's malleability during its execution. The malleability was primarily utilised for improving the application performance. As a result, it hindered the system-level optimisations otherwise possible with a malleable infrastructure, such as launching moldable jobs to utilise the idle nodes and reconfiguring the resources in running applications by considering the pending jobs. With the adaptive batch scheduling implemented in this work, the full potential of application malleability along with the system benefits can be considered. Towards that, an Adaptive Batch Scheduler (`ABS`) is proposed that considers the applications in the queue along with running malleable applications. This opens up lot of benefits to the overall system-level metrics.

4.1 The Adaptive Batch Scheduler

To implement `ABS`, the default Slurm scheduler was added with two queues to differentiate rigid and malleable jobs. This is visible in the architecture diagram of `ABS` in Figure 4.1. `ABS` is a scheduling plugin (see Section 2.2.1.1) that is built on top of the default Slurm scheduler, and the algorithm of the `ABS` scheduling is shown in Algorithm 1. `ABS` gathers information about the current resources in the system (Line 2). This includes but is not limited to the total number of compute nodes, idle nodes, and the total number of nodes

```
1 #SBATCH --job-name adaptive_job_script
2 #SBATCH --time=00:30:00
3 #SBATCH --nodes=4
4 #SBATCH --ntasks-per-node=96
5 #SBATCH --min-nodes-invasic=2
6 #SBATCH --max-nodes-invasic=8
7 #SBATCH --node-constraints="odd"
8 #potential values are even, odd, ncube, pof2, #none
9
10 srun sample_app
```

Listing 4.1: Simple batch script for adaptive batch scheduling [55].

in the completing state. This information is collected using Slurm global variables called `bitmaps`. The `bitmaps` are changed by Slurm plugins during the job allocation to keep track of the available resources. In addition, the `bitmaps` are also modified based on the expand or shrink operation decisions made by `ABS` (Lines 21 - 24). `ABS` also gets information about the jobs submitted from users using the batch submission, like the total number of jobs and the requested resources. The jobs are differentiated based on the entries provided in the batch script.

As part of this work, special parameters are added to the batch script to define the malleable aspects of the application. A sample batch script is shown in Listing 4.1. The jobs are added to the elastic queue whenever entries are associated with the fields `min-nodes-invasic` and `max-nodes-invasic` (Lines 5-6). Additionally, users can leverage the `-node-constraints` (Line 7) option to pass hints regarding the application to `ABS`. Currently, `ABS` supports options like power-of-two, even, odd and cubic number of nodes (Line 9). This is relevant for applications like Lulesh [192], which require a cubic number of processes for execution. `ABS` can consider these constraints while making resource adaptation decisions. The absence of such hints can result in `ABS` allocating an incompatible number of nodes to an application during the resource reconfiguration, resulting in an application failure. Furthermore, the user can also specify the application's power consumption characteristics (Lines 6-7 in Listing 5.1) in the job script for making power-aware reconfiguration decisions described in Section 5.1. Upon receiving the submitted jobs, a First Come, First Serve (FCFS) policy is used by `ABS` to prioritise the jobs (Line 4 in Algorithm 1).

`ABS` attempts to schedule and run as many rigid and malleable (or resource-elastic) applications as possible based on their resource requests and job priorities (Line 4). Based on resources specified in the `--nodes` field in the batch script (see Listing 4.1), the user-submitted jobs are executed. During application execution, `ABS` expands or reduces the resources of an application based on the constraints set in the job script, like the minimum and maximum number of nodes indicated in the `min-nodes-invasic` and `max-nodes-invasic` fields as well as `-node-constraints`. After this, the scheduler updates its performance metrics (Line 5) based on the system's current state, such as system utilisation, job waiting and response times.

The scheduling loop in `ABS` (Line 6) is responsible for making dynamic resource reconfiguration decisions for the running malleable applications. Three different events trigger this loop.

- It is triggered on each `scheduler tick` seconds, which `ABS` reads from the Slurm configuration file `slurm.conf`.
- It is triggered when a new job (application) is added to the job queue (both rigid and malleable).
- It is triggered whenever an application finishes execution.

Whenever one of these events occurs, `ABS` receives new resource information regarding the system (Line 7), which may have altered due to previously scheduled jobs.

By iterating through every job in the system, ABS constructs four distinct lists of jobs (as indicated in Lines 9-10) categorised into running and waiting jobs, with each being further divided into malleable (or elastic) and rigid jobs. These lists play a crucial role in facilitating dynamic reconfiguration decisions, which are informed by the malleability and job management policies outlined in Section 4.2. Once the lists are created, ABS checks all the currently running malleable jobs to see if they have performance data. To do this, ABS checks a flag variable associated with the job in the `job_record` structure. If required performance data is missing, ABS requests it (Line 11-13). This technique obtains the performance data from malleable applications built using the Elastic Phase Oriented Programming Model (EPOP). For other malleable applications written using pure iMPI, the node-local performance data is obtained internally from iMPI by SLURMD daemons. A performance measurement handler was developed to aggregate and reduce values obtained from all daemons.

Algorithm 1. The Adaptive Batch Scheduler (ABS) Iteration [55].

```
1  while TRUE do
2      Get resource info from Slurm
3      Get workload info from Slurm
4      Schedule jobs (malleable and rigid) in priority order if required resources are present
5      Update the scheduler metrics
6      if a scheduling event occurs then
7          Update resource info
8          for each job in job list do
9              Update waiting malleable and rigid job list
10             Update running malleable and rigid job list
11             if no performance data available for running malleable job then
12                 request_perf_data(job);
13             end
14             Check if any running malleable job is adapting
15         end
16         if no malleable job is adapting and running elastic jobs > 0 then
17             Generate resource vectors (RSV) to expand or shrink running malleable jobs based on the
                 emplyoed job scheduling strategy
18             Update reduction job list
19             Update expansion job list
20             for each job in reduction job list do
21                 shrink_job(job);
22             end
23             for each job in expansion job list do
24                 expand_job(job);
25             end
26         end
27     end
28 end
```

The performance data used by ABS is called the MPI Time to Compute Time (MTCT) ratio. It is an intuitive heuristic criterion measuring the time spent in MPI calls versus the time doing relevant computation. It gives an outlook into the efficiency of an MPI application. The higher the ratio, the more time is spent on communication than computation for a particular application. This value can be successfully used to decide whether an application benefits from resource expansion or reduction. In performance-aware scheduling,

this metric can decide the application for shrinking (resource reduction) out of a pool of running malleable applications to launch a waiting job with highest priority from the queue (see Section 4.2.3). In addition, ABS also checks whether there are any pending job adaptations, i.e., expanding or shrinking (Line 14). It ensures consistency among compute resources and guarantees no new resource reconfiguration decisions are performed before all pending resource reconfigurations are completed. IS_ADAPTING flag is used to achieve the above objective.

Suppose no jobs are currently adapting, and malleable jobs are running in the system. In that case, ABS makes expand or shrink decisions based on current job management and adaptive scheduling strategy (Line 16). Multiple adaptive scheduling strategies for ABS are implemented and are described in detail in Section 4.2. The application reconfiguration decision from ABS is decided by examining the resource vector (RSV) data structure. RSV is an array sized to match the number of running malleable jobs and ordered based on the application arrival time. Each index of the RSV contains the node count associated with currently running applications. The change in RSV results in resource reconfiguration.

ABS iterates the RSV array and chooses to expand or reduce a job based on the ensuing criteria:

- *expansion*: If the corresponding entry in RSV array for the current job is greater than the currently allocated number of nodes.
- *reduction/shrink*: If the corresponding entry in RSV array for the current job is less than the currently allocated number of nodes.

For instance, if a malleable job executes on four compute nodes and the scheduling strategy determines that the job should be scaled down to two nodes, the updated RSV for that particular job in the index will be adjusted to two.

ABS makes two separate lists for jobs that are considered to be expanded and reduced. The expansion and reduction is performed as a two-step process. Firstly, ABS iterates through the reduction job list and performs resource reduction (Lines 20-22). Once all reductions are performed, ABS iterates through the expansion list (Lines 23-25) and triggers the expansion. To achieve that, for every job in these lists, ABS devises an RPC message request of type slurm_reallocation_message and is sent to the invasive runtime scheduler iRS. Essential information, like the number of preexisting processes to kill and nodes to retreat from, is contained in the message transferred for performing resource reduction. Meanwhile, for a resource expansion, the message includes information such as number of tasks to create and the compute nodes on which to launch them. Upon receiving this message, iRS performs the dynamic reconfiguration of batch jobs as described in Section 2.2.1.3. This can be seen in Figure 4.1.

iRS is responsible for maintaining a consistent system state while performing the resource expansion and reduction decided by ABS. To aid that, a reallocation handler was developed as part of this work. The new reallocation handler is accountable for modifying the context of a running job step, creating processes during a resource expansion, and terminating corresponding processes during a resource reduction. Using the existing reallocation handler is insufficient since the control flow of a batch job is different from an interactive job launch in Slurm. ABS iteration continues until the iRS is terminated.

4.2 Adaptive Scheduling Strategies

This section describes different scheduling strategies implemented in this work for executing and scheduling malleable applications. An adaptive scheduling strategy can be considered a combination of two policies ❶ a job management policy, and ❷ a malleability management policy. The former is mainly in charge of deciding ***whether*** to increase or reduce the resources of a running application. The latter decides on ***how*** to increase or reduce the resources.

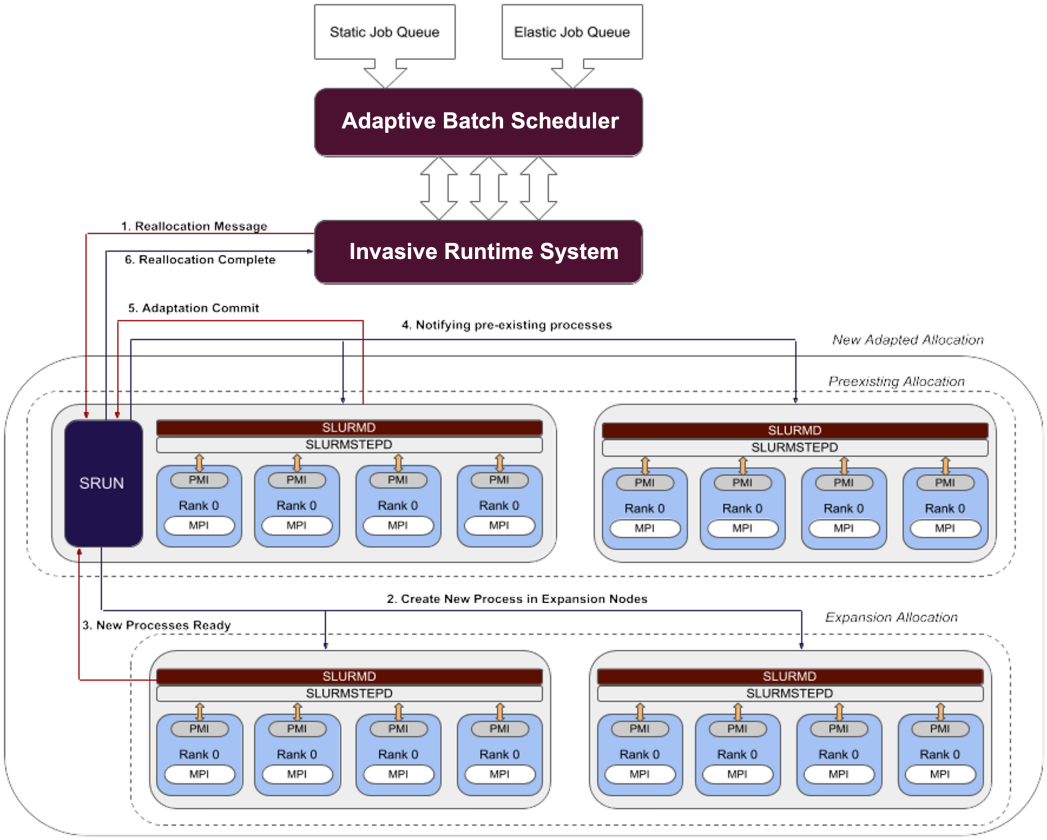


Figure 4.1: Adaptive Batch Scheduler Architecture [88]

4.2.1 Job Management Policies

The following job management policies are analysed in this work:

- Priority to running malleable applications (PRMA): In this strategy, a malleability management policy is employed to increase (expand) the resources of an already running malleable application whenever a compute resource becomes available. However, the highest priority job waiting in the queue is executed if the job's resource requirements become available.
- Priority to waiting malleable applications (PWMA): In this approach, a malleability management policy is used to reduce the resources of running applications to cater to the needs of a high-priority malleable application waiting in the queue. Shrink operations are the norm in this strategy. However, if sufficient resources cannot be gathered to start the waiting application, a malleability management policy performs expansion on running jobs.

4.2.2 Malleability Management Policies

This work extended the malleability management policies proposed by Buisson et al. [117] for multicluster grid systems to support HPC applications. Favor Previously Started Malleable Applications First (FPSMA)

and Equi-Grow Shrink (EGS) policies are adapted and extended in ABS. The procedures to expand and shrink the resources were added to both policies. The algorithm for FPSMA (Algorithm 2) provides an overview of the procedures.

4.2.2.1 Favor Previously Started Malleable Applications First (FPSMA)

As seen in Algorithm 2, the FPSMA_GROW procedure takes the elastic (malleable) job list and the current number of jobs as arguments and returns the filled resource vector (RSV). After that, ABS uses the RSV to expand resources on the running malleable jobs (see Section 4.1). Initially, the procedure arranges the malleable jobs based on their start time (Line 2) and records the current idle node count (Line 3) using the idle node bitmap (see Section 4.1). As seen in lines 6-7, the procedure iterates through the sorted job list to calculate the current node distribution and the remaining execution time of each job.

A job is deemed for resource expansion if the following conditions (Line 9) are satisfied :

1. If the current number of nodes is less than the value provided for the `--max-nodes-invasic` parameter in the job script (see Section 4.1).
2. The system has idle nodes.
3. The remaining execution time of the job is more than 60.0 seconds.

If the above conditions are unmet, the job is not considered for resource expansion (Line 8). If a job is selected for resource expansion, the constraints specified in the `--node-constraints` parameter (see Listing 4.1) are analysed (Line 10). For example, if the parameter's value is `--node-constraints='even'`, the current node count is two, and the remaining idle nodes are 8. Then, the procedure computes the appropriate *growValue* (Line 11), which is four in this case. The job is then marked for resource expansion by updating RSV with *growValue* (Line 12), and the new idle node count is calculated (Line 13). FPSMA_GROW performs the above steps iteratively until all jobs in the list have been considered.

Algorithm 2 for the FPSMA_SHRINK procedure takes an extra *waitingJob* parameter along with the arguments of the FPSMA_GROW procedure . The *waitingJob* parameter points to the queue's next highest-priority waiting job, which can be both rigid and malleable. This argument is necessary for the implementation of PWMA job management policy. FPSMA_SHRINK procedure sorts the malleable job list in the decreasing order of the job start times (Line 20). The procedure calculates the remaining idle nodes in the system and compute nodes needed by the waiting job (Lines 21-22). Similar to the FPSMA_GROW, the FPSMA_SHRINK iterates through the job list and calculates the current node count and the remaining job time of all the jobs. A running job is deemed for a resource reduction (shrink) operation if the following conditions are satisfied:

1. If the value provided in the job script for the parameter `--min-nodes-invasic` does not equal the current number of nodes.
2. The waiting job still requires some nodes.
3. The remaining execution time of the job is more than 60.0 seconds.

If all the above conditions are met, an appropriate *shrinkValue* is calculated for the job by considering the `--node-constraints` provided in the job script. It is similar to the FPSMA_GROW procedure. For example, if the parameter's value is `--node-constraints='odd'` and the node count of the job is nine. FPSMA_SHRINK calculates the next ideal *shrinkValue*, which is seven in this scenario (Line 29). The job is marked for resource reduction by updating RSV with new *shrinkValue* (Line 30). In addition, the number of the remaining nodes needed to launch the waiting job is updated (Line 31). FPSMA_SHRINK perform the above steps iteratively till all the jobs in the list are considered. Nevertheless, if the number of nodes needed to launch the high-priority jobs cannot be obtained, then no resource reductions are performed in this scheduler pass.

Algorithm 2. Adapted Favor Previously Started Malleable Applications First (FPSMA) job scheduling strategy [117].

```
1  Function FPSMA_GROW(RSV, malleableJobList, jobCount):
2      malleableJobList ⟵ jobs sorted in ascending order of their start time
3      remainingIdleNodes ⟵ number of idle nodes
4      Set RSVIndex to 0
5      for each job in malleableJobList do
6          jobRemainingTime ⟵ time remaining for the job to finish
7          currentNodeCount ⟵ current number of nodes allocated for the job
8          RSV[RSVIndex] ⟵ currentNodeCount
9          if currentNodeCount ≠ jobMaxNodes & remainingIdleNodes > 0 & jobRemainingTime > 60.0
            then
10             Analyse constraints of job wrt remainingIdleNodes
11             Calculate new growValue
12             Update RSV[RSVIndex] as growValue
13             remainingIdleNodes ⟵ remainingIdleNodes − growValue
14         end
15         RSVIndex++
16     end
17 return
18 Function FPSMA_SHRINK(RSV, malleableJobList, waitingJob, jobCount):
19     malleableJobList ⟵ jobs sorted in descending order of their start time
20     remainingIdleNodes ⟵ number of idle nodes
21     nodesRequired ⟵ waitingJobNodes − remainingIdleNodes
22     Set RSVIndex to 0
23     for each job in malleableJobList do
24         jobRemainingTime ⟵ time remaining for the job to finish
25         currentNodeCount ⟵ current number of nodes allocated for the job
26         RSV[RSVIndex] ⟵ currentNodeCount
27         if currentNodeCount ≠ jobMinNodes & nodesRequired > 0 & jobRemainingTime > 60.0 then
28             Analyse constraints of the job and find new shrinkValue
29             Update RSV[RSVIndex] as shrinkValue
30             nodesRequired ⟵ nodesRequired − shrinkValue
31         end
32         RSVIndex++
33     end
34     if nodesRequired > 0 then
35         Do not reduce any job.
36     end
37 return
```

4.2.2.2 Equi-Grow Shrink (EGS)

The second malleability management policy, Equi-Grow Shrink (EGS), also has two procedures associated with dynamic resource reconfiguration. Similar to `FPSMA_GROW` and `FPSMA_SHRINK` procedures, they are `EQUI_GROW` and `EQUI_SHRINK` procedures. Currently, the `--node-constraints` parameter is not considered in EGS. In the case of `EQUI_GROW`, the idle nodes are distributed equally amongst all the running applications by computing a suitable *growValue*. If the idle nodes cannot be divided equally among running applications, the remainder is distributed among applications based on their job start time. In the `EQUI_SHRINK` procedure, the number of compute nodes needed by the waiting application with highest priority are equally reclaimed

Algorithm 3. The ABS Performance-aware Scheduling function [55].

```
1  Function Perf_Aware_Schedule:
2      Get resource info from slurm
3      Get workload info from slurm
4      while requested resources available do
5          Launch rigid and malleable jobs using the employed priority scheme
6      end
7      if highest priority waiting job cannot be started then
8          for each running malleable job do
9              if job is adapting then
10                 set any_job_adapting
11             end
12             else if no performance data available for job then
13                 request_perf_data(job);
14             end
15         end
16         if no malleable job is adapting and number of active malleable jobs > 0 then
17             Check if active malleable jobs can be reduced wrt decreasing MTCT ratios to start
                highest priority waiting job
18             if enough nodes were found then
19                 Reduce nodes from the selected malleable jobs.
20                 Start the highest priority waiting job.
21             end
22             if insufficient resources found or idle nodes available then
23                 Check active malleable jobs for expansion wrt increasing MTCT ratios.
24                 Increase nodes in selected jobs.
25             end
26         end
27     end
28 return
```

from running applications by computing suitable *shrinkValue* for all running applications. Nevertheless, no reduction operations are performed if sufficient nodes cannot be obtained to launch the application, similar to `FPSMA_SHRINK` procedure.

In addition to the above expansion and shrink procedures, a heuristic-based resource reconfiguration procedure was also implemented in `ABS`. The metric `MTCT` ratio is used as a criterion to choose an application for dynamic resource reconfiguration. A resource expansion is performed on the running application only if the application has an `MTCT` value below a predetermined threshold. On the contrary, if the `MTCT` value is above the threshold, the application is marked for resource reduction. The `MTCT` threshold is set using the slurm configuration file (`slurm.conf` file) and are read by `ABS` during startup. The heuristic-based procedure is explained in Section 4.2.3.

4.2.3 Performance-aware Scheduling of Malleable Jobs

The methodology for performance-aware scheduling of rigid and malleable jobs is described in Algorithm 3. The `Perf_Aware_Schedule` function is in charge of starting jobs and making dynamic resource reconfiguration decisions for running (or active) malleable applications. It is event-triggered and executes on three distinct events as detailed in Section 2.2.1.2. At first, `ABS` acquire details about the current state of the available resources in the system and the information about the jobs submitted by users in the malleable and rigid job queues (In lines 2-3). Then, it attempts to schedule and launch as many malleable and rigids jobs as possible, based on priority of jobs and resource constraints and requirements (Lines 4-6). For every job, priorities are determined based on their arrival time. Malleable jobs are started with nodes requested in the `--nodes` parameter of the job script. They are later reduced or increased depending on the minimum and maximum number of nodes requested in the provided batch script. It is worth noting that the `--nodes` parameter can differ from the minimum nodes requested in the job script, giving more flexibility to the batch system regarding expand/shrink operations. This is different from previous strategies [113, 117] where already expanded jobs were only considered for resource reduction.

If resource unavailability leads to the holding of a high-priority job in the queue, then no lower-priority application is selected for scheduling (Line 7). As a result, the algorithm iterates through the list of active malleable jobs in the system and queries whether any resource adaptations are ongoing (Lines 8-9). The `job_state` variable provides information about current adaptations of the system. Ongoing adaptations in the system are marked using a flag *any_job_adapting* (Line 10). In the event of no ongoing adaptations, the algorithm looks for the performance data of active applications. If no performance data is available, the algorithm requests the data for the `iMPI` application (Line 13). The adaptation check in Line 16 guarantees the system's consistency by ensuring that new resource change decisions are only made after finishing pending adaptations.

The expansion/reduction decision phase is started in the algorithm if there are no ongoing adaptations and at least one malleable job is actively running in the system (Line 16). The priority is given to waiting jobs in this scheduling policy. The job with the highest priority in the waiting queue is chosen, and the scheduler attempts to launch it by taking away resources from active malleable applications. This is a variation from previous strategies where expansion/reduction operations were only performed to consider the efficiency of running applications.

The algorithm attempts to reduce resources of running applications based on their `MTCT` ratios so that the higher-priority job will be launched. The algorithm reduces running jobs in the decreasing order of their `MTCT` ratios to obtain resources to launch a higher-priority waiting job (Line 17). These reduction operations consider the node requirements (minimum and maximum values) and constraints (odd, even) on the number of nodes provided in the job script. For an entry in the `--node-constraints` parameter, the reduction operation attempts to decrease the number of nodes to the following lower constraint based upon the number of nodes needed to launch the waiting job. For example, if the node constraint is `even`, the number of nodes allocated to a job is ten, and remaining nodes required to start the waiting job are eight, then the reduction operation will attempt to reduce the node count to two. Nevertheless, selected jobs only encounter a resource reduction operation if all essential resources are readily available for the top priority waiting job (Lines 18-19). The waiting job is started immediately after the resource reduction operation is completed (Line 20).

If idle nodes are available in the system after the resource reduction, the algorithm moves into the expansion phase for running malleable jobs. Running malleable jobs are prioritised for expansion based on their `MTCT` ratios (Line 23). This increases system throughput, as demonstrated in Chapter 9. Like the resource reduction, the expansion algorithms consider the `--node-constraints` parameter while making the resource change decision. Finally, the jobs are expanded after distributing all idle nodes between chosen running jobs (Line 24).

5

Power Corridor Management

This chapter explains the second contribution of this work, a power-aware batch scheduler. At first, the core concepts regarding power-aware batch scheduling are presented and disclose how dynamic resource management can be leveraged for power management. Later, an explanation of the implementation aspects of the work is provided, where the power-aware scheduler algorithm is introduced. Finally, this chapter is concluded by describing the dynamic power management feature provided in the work.

Electricity contracts between energy companies and computing centres are often structured around a power corridor, setting specific upper and lower power consumption limits that the centres must adhere to [193]. Non-compliance could lead to penalties imposed by the energy providers. In some cases, computing centres also serve as stabilizers for the grid load [194], where dynamic adjustments to the power corridor could be a contractual element, potentially triggered by requests from the electricity company. Such arrangements benefit the computing centre economically by reducing electricity costs. While techniques like power capping are commonly employed to enforce upper power limits [121], they are not applicable to maintaining the lower limit, i.e., increasing power consumption when necessary. In these instances, dynamic resource management emerges as a viable strategy for managing the system's power consumption within the defined corridor.

The effectiveness of this strategy hinges on understanding the power usage characteristics of various applications, which can differ significantly [195]. Applications can generally be classified as high-power or low-power consumers. By reallocating resources between these types of applications during runtime, it's possible to adjust the total power usage of the computing system to align with the corridor constraints. This approach necessitates a power-aware resource and job management system (RJMS) and malleable applications that can dynamically adapt their resource usage.

The power-aware RJMS should be able to ❶ identify the power usage of the compute resources, ❷ find out the power profile of running applications (scaling and non-scaling), ❸ compute new node configuration according to the power profile of running applications, ❹ reconfigure the resource distribution of running applications, ❺ schedule new jobs that do not violate the power requirements, and ❻ adapt to the varying power corridors. Towards this, a power-aware runtime system and a power-aware batch scheduler were developed on top of `iRM` (See section 2.2.1). This work utilise and extend the components `iRS` (See section 2.2.1) and `ABS` (See section 4.1).

5.1 Power-aware Batch Scheduler Concepts

When a user submits a job to an HPC system, it is immediately added to the job queue if not enough resources are available to schedule the job. When the resources become available, the scheduler schedules the job with the highest priority. Priority is often assigned based on the site-level policy employed in the center. This is the typical behavior of a batch scheduler in an HPC system. The proposed power-aware batch scheduler changes this behavior of `iRM` to adhere to the power bound set by the site-level policy.

During a job submission, the power-aware scheduler must have a different operating procedure than the performance-aware scheduler. This can be attributed to the power usage characteristics of different applications. If the system is ideally in the power corridor, scheduling a new job upon arrival might lead to a power corridor violation if necessary steps are not taken. The power-aware scheduler must consider the following when a job is submitted:

- A submitted job must be able to be held by the scheduler even if the resources are available.
- The power usage characteristics of the job should be analysed before scheduling the job.
- Analyse the job queue and set priority according to the system-level energy requirements.

During the end of a job, the scheduler must consider the following:

- Try to schedule a new job that satisfies the power corridor.
- Find a resource reconfiguration that can maintain the power corridor and schedule the new job.

Listing 4 showcases the algorithm of the power-aware scheduling function. Before analyzing the power-aware scheduling algorithm, it is imperative that the core concepts behind the resource reconfiguration for power corridor management be understood. A linear programming model is proposed (See section 5.1.1) to calculate the new system configuration. The new resource configuration is calculated using the model, and `iRS` will take the necessary steps to enforce the new configuration.

The architecture of the power-aware scheduler (Figure 5.1) is similar to the adaptive batch scheduler. Instead of an adaptive batch scheduling plugin, the power-aware scheduling plugin is utilised for power corridor management. The changes performed in this work only touched the scheduling element, not the runtime configuration. Nevertheless, the messages sent and operations performed to perform the adaptation are the same as the `ABS` defined in Section 4.1.

5.1.1 Linear Programming Model for Resource Reconfiguration

When a power corridor violation is encountered, the Linear Programming (LP) model provided in Equation 5.3 is utilised to enforce the power corridor. The core objective of the proposed model is to enhance system utilisation by reducing the idle nodes in the system. The LP model is designed to use both active (currently running in the system) and pending jobs (waiting in the queue) to form a new resource configuration that fulfills power corridor requirements of the system.

The following assumptions are made in the model to simplify the analysis:

- the power consumption per node is uniform (i.e., the same amount of power is consumed by all nodes for a particular application).
- the system is homogeneous (i.e., all idle nodes use the same amount of power).

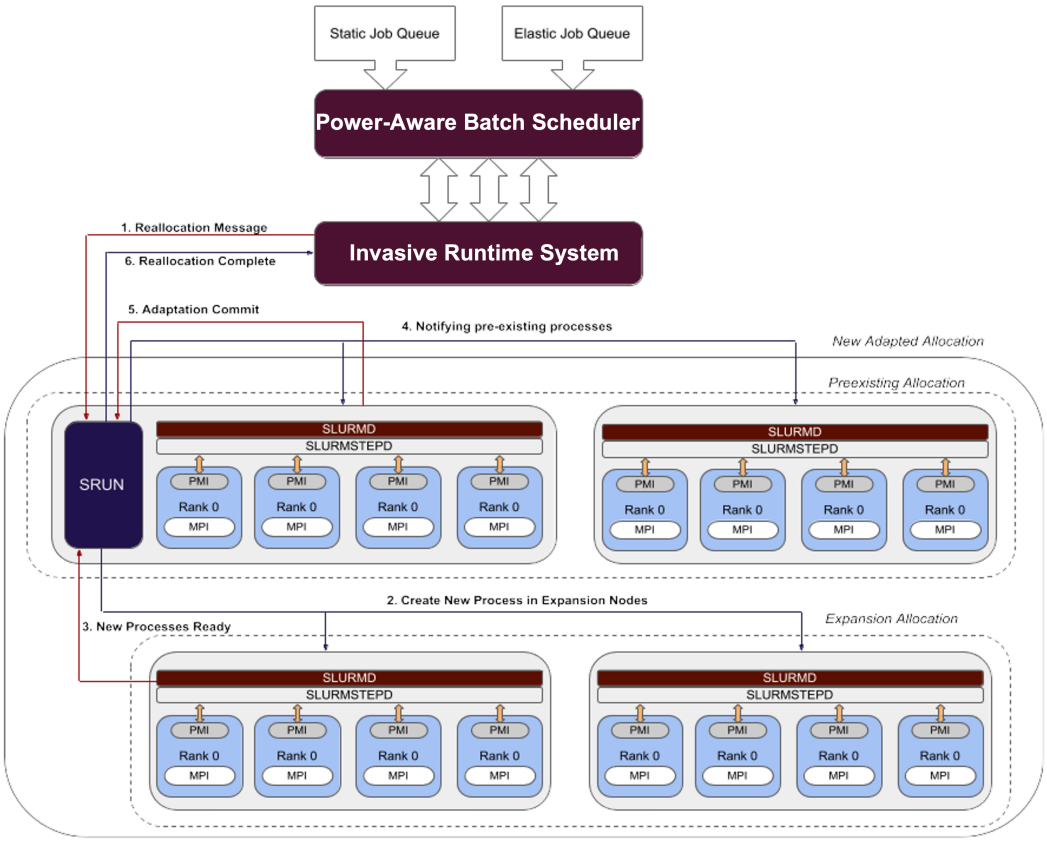


Figure 5.1: Adaptive Power-aware Batch Scheduler Architecture [88]

A power corridor is said to be broken when the total power usage of the system is above or below the boundary. The lower corridor violation can be expressed formally as:

$$l \geq \sum_{i=1}^{K} k_i * P_{min}^{(i)} + k_{idle} * p_{idle} \tag{5.1}$$

The upper corridor violation can be expressed formally as:

$$u \leq \sum_{i=1}^{K} k_i * P_{max}^{(i)} + k_{idle} * p_{idle} \tag{5.2}$$

Where, l and u denote the lower and upper power boundaries, $P_{max}^{(i)}$ and $P_{min}^{(i)}$ represent the total power used by Job `i`, and k_{idle} and p_{idle} represent the number of idle nodes and the power used per idle node. In such a case, a new node distribution $D_{new} = [k_1, k_2, .., k_K, m_j]$ should be found, where k_i represents the new node distribution of the i^{th} Job, K represents the total number of jobs running in the system and m_j represents the j^{th} job in the waiting queue.

By assuming that power consumption per node is uniform, an application's minimum and maximum power

usage per node can be calculated as $p^{(i)}_{min,max} = P^{(i)}_{min,max}/k_{old}$, where k_{old} is the old number of nodes. This information can be used to generate arrays of minimum and maximum power consumption of running applications per node which can then be solved as an LP optimization problem to determine the node distribution that satisfies the power corridor.

Considering that the potential maximum and minimum power consumption for an application per node can be derived by merely calculating $p^{(i)}_{min,max} = P^{(i)}_{min,max}/k_{old}$, we can then construct two arrays of maximum and minimum power consumption per node, $p_{idle}, p^{(1)}_{min,max}, p^{(2)}_{min,max}, \ldots, p^{(N)}_{min,max}$, each of size N. This creates a Linear Programming Optimization problem (LP) which is shown in our model. The following restrictions are added to the LP model:

- The total power consumption in all nodes must be within a lower value l (lower corridor) and upper value u (upper corridor).
- Every job needs to be allocated a minimum of one node. ($k_i \geq 0$).
- Largest job can have up to $N-K$ nodes, where N is the total number of available nodes in the system, and K is the number of jobs.
- Number of idle nodes (k_{idle}) can range from 0 to $N-K$.
- Use as few idle nodes as possible.

The LP model with the above restrictions will generate a new node distribution, which the power-aware scheduler can use to enforce the power bound.

$$
\begin{aligned}
&Minimize: \\
&\quad f(k_{idle}) = k_{idle} * p_{idle} \\
&Subject\ To: \\
&\quad l \leq \sum_{i=1}^{K} k_i * p^{(i)}_{min} + k_{idle} * p_{idle} + m_j * p^{(j)}_{min} \\
&\quad u \geq \sum_{i=1}^{K} k_i * p^{(i)}_{max} + k_{idle} * p_{idle} + m_j * p^{(j)}_{max} \\
&\quad k\_min_i \leq k_i \leq k\_max_i,\ k_i \in \mathbb{N} \setminus \{0\},\ i = 1, \cdots, K \\
&\quad 0 \leq k_{idle} < N-K,\ k_{idle} \in \mathbb{N}
\end{aligned}
\tag{5.3}
$$

5.1.2 Guarantees

In some cases, there might be no solution for the above-defined linear programming optimization problem. The LP problem, as defined, might be infeasible to solve. The power-aware scheduler can only guarantee a solution if particular criteria are met.

If the total number of nodes is N and K applications are running in a system,

- the power-aware scheduler will ensure compliance with the upper power corridor limit, u, only if the total system power consumption, when each application operates on a single node, remains below the upper bound u. This condition is formalised in Equation 5.4.

$$
u \geq \sum_{i=1}^{K} p^{(i)}_{max} + (N-K) * p_{idle} \tag{5.4}
$$

- lower power corridor l enforcement from the power-aware scheduler is only guaranteed if the system's power consumption exceeds the lower power corridor boundary when the most power-hungry application A is running on $N-(K-1)$ nodes. This is shown in Equation 5.5.

$$l \leq \sum_{i=1}^{K-1} p_{max}^{(i)} + (N-(K-1)) * k_A * p_A \tag{5.5}$$

As seen in the above LP model, one of the vital inputs to the LP model is the power consumption per node. `iRS` is extended to obtain this information directly from the application.

5.1.3 Power Measurement

Accurate power values of applications at regular intervals are essential for the power-aware scheduler to make proper scheduling decisions. The proposed system has two kinds of jobs, as seen in the above LP model. The former category is the jobs waiting in the queue to be scheduled, and the latter is the currently running jobs in the system.

The application developer can pass the approximate power usage for the waiting jobs by leveraging the special configuration parameters during the job submission. The proposed batch system provides two parameters `--min-power` and `--max-power` to developers (See lines 6-7 in Listing 5.1). If no values are given, the scheduler searches for the historical data. The job name and the number of resources requested are compared with historical data to find a match.

For the actively running jobs, the power values are obtained directly from the application. They are obtained regularly, and the power-aware `iRS` can also dynamically change the frequency and number of power measurements from applications written in `EPOP` model. On Intel systems, estimations of energy consumption are facilitated through Running Average Power Limit (RAPL) sensors, as detailed by Khan et al. [196]. These sensors provide a mechanism to monitor energy usage, with the data accessible through Model Specific Registers (MSR). To determine power consumption, the energy consumed is divided by the interval between successive measurements. There are various libraries available to interact with these registers, such as LIKWID [197], which enable the extraction of these energy readings. The obtained power values are preserved for future job submissions and later used to predict power usage.

5.1.4 Forecasting

There are two approaches to dynamic power management. The first approach is a proactive approach where the potential power usage is predicted and takes necessary steps to manage the power. In the second, a reactive approach is used where the necessary steps are taken to increase or decrease the power usage once a threshold is crossed. The proposed power-aware runtime system follows both proactive and reactive power management approaches. In the proactive approach, power measurements are passed to the forecast module to predict the maximum and minimum power consumption. If the predicted consumption exceeds the power corridor, Equation 5.3 is solved using the prediction. However, in the reactive approach, the forecast module is not utilised; rather, actual power measurements are used whenever there is a violation.

The forecasting module predicts whether the system will deviate from the predefined power corridor using the time series analysis technique. Time series analysis can unveil the inherent patterns within data sets, such as power consumption values given,

```
#SBATCH --job-name test_job_script
#SBATCH --time=00:30:00
#SBATCH --nodes=4
#SBATCH --min-nodes-invasic=2
#SBATCH --max-nodes-invasic=8
#SBATCH --min-power=50 # Watts
#SBATCH --max-power=100 # Watts
srun sample_app
```

Listing 5.1: Example batch script with additional options for power-aware scheduling [55].

- A valid time series: A valid time series is constructed in this work using the techniques mentioned in Subsection 5.1.3.
- A specific method to examine the data: For this, three techniques are used to model the power consumption data. They are,
  - ARIMA [198]: The AutoRegressive Integrated Moving Average (ARIMA) model combines an Integrated component, responsible for transforming data into a stationary series, with an ARMA component, which models this stationary data. The ARMA component is further split into an AutoRegressive (AR) part, illustrating the relationship between a time series value and its predecessors, and a Moving Average (MA) part, reflecting the impact of random shocks. The integration of these components forms the ARIMA model, which is apt for analysing and forecasting data points in time series.
  - SARIMAX [199]: Seasonal ARIMA with Exogenous Regressors (SARIMAX) expands upon the ARIMA model by accommodating time series data with a seasonal pattern and integrating the effects of exogenous variables, which are external influences outside the time series.
  - Holt-Winters [200]: This method, also known as Third Exponential Smoothing, applies exponentially decreasing weights to past observations, making it suitable for data with trends and seasonality. It can handle both additive and multiplicative seasonal effects.

Figure 5.2 shows the power prediction of a time series using the above three different techniques. Even though the model is not accurately predicting the exact power usage at a time, it is still sufficient for power corridor management since the LP model deals with maximum and minimum power usage. As can be seen in the figure, the maximum and minimum power usage during the predicted interval is similar to the maximum and minimum of the actual time series for the same interval.

5.2 Power-Aware Batch Scheduler Implementation

Initially, power-aware `ABS` acquires information regarding the status of jobs and resource availability in the system. The jobs are then started based on the power values per node (min and max provided in the job script) given by the user. After the application launch, `ABS` periodically updates the power values corresponding to the application and watches for potential violation of power corridors of the system. Algorithm 4 details the power-aware job scheduling strategy for dynamic power corridor management. Like the performance-aware strategy, the `Power_Aware_Schedule` function is triggered on the events detailed in Section 2.2.1.2. It is responsible for scheduling jobs that adhere to the power bounds set in the system and making resource reconfiguration decisions to control the system power and maintain it within the power budget.

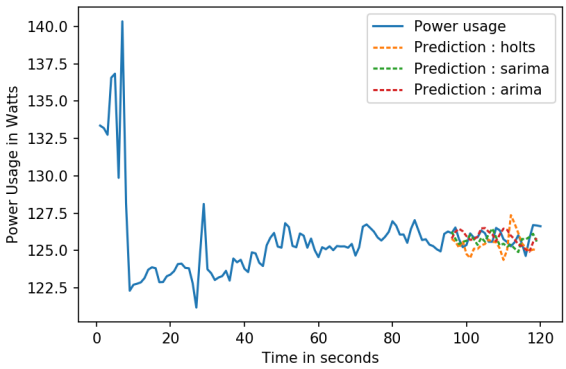


Figure 5.2: Forecasting using different techniques

The implemented power-aware scheduler works as follows. At first (Line 2 in Algorithm 4), the monitoring infrastructure analyses the existing power budget, the user-submitted approximate minimum and maximum power usage (see lines 6-7 in Listing 5.1), and any historical power data associated with the submitted job. A job is launched if the requested number of nodes is available and it satisfies the power corridor constraints (Line 4). Otherwise, it is added to the waiting queue. After launching jobs, `ABS` periodically updates the information regarding the amount of power consumed by corresponding jobs (Line 3). For every job, the average power consumed during the execution is stored along with its name in an array to facilitate reruns. Then the `ABS` watches for a power corridor violation (Line 4, 5) if there is availability of active malleable jobs in the system and there are no pending adaptations. If a power corridor violation is recognised, the LP model provided in Equation 5.3 is utilised to calculate new resource redistribution for enforcing the power corridor.

The LP model utilises information about both running and waiting jobs within the system to determine a new resource allocation that aligns with the system's power budget. The model selects the first viable resource configuration that incorporates the waiting job. Then, it adjusts the resources allocated to currently running jobs through either expansion or reduction to meet the specified power constraints. The solution to the system is found using the Coin-or Branch and Cut (CBC) solver via Pulp, a Python Integer Programming Solver module [201]. The solving time of Pulp for a system with applications $K = 2, 4, 8$, and 16 was always under 0.5 seconds and is fast enough since a decision has only to be made from one schedule pass to the next. Finally, a resource change is triggered by `Power_Aware_Schedule` with a valid node distribution found by Pulp. Additionally, the selected waiting job is launched. If no valid distribution is found, the current distribution is maintained. `iRS` monitor the available power budget and launch the jobs (jobs with estimated power usage that can adhere to the power corridor) in priority order if power usage is in the valid bounds.

The new valid configuration can contain resource reduction and expansion for a set of applications and no resource change for another set of applications. `Power_Aware_Schedule` will utilise the `iRS` to grow and shrink the resources of the applications based on the configuration provided by PuLP. To do that, the valid configuration is copied into the corresponding application indices of `RSV` array. `RSV` is not modified for application with no resource change. For application with resource change, `RSV` is modified as described in Section 4.1. Following that, all the resource reduction operations are performed simultaneously. Following the reduction, expansion operations are performed for the applications. Once both expansions and reductions are performed, the application from the waiting queue is launched. This expansion and reduction is performed similarly to the `ABS` mentioned in Section 4.1. `iRS` is utilised for the runtime reconfiguration of the running application.

Algorithm 4. The ABS power-aware scheduling function [55].

```
1  Function Power_Aware_Schedule:
2      Get resource and workload info
3      Update the power usage information for each running job
4      if no malleable job is adapting and number of running malleable jobs > 0 then
5          if power corridor is broken then
6              Hold the scheduling of jobs.
7              if strategy = favour elastic jobs then
8                  Pick malleable jobs from waiting queue
9              end
10             for each waiting job j in priority order do
11                 Calculate new resource distribution using LP with running jobs and waiting
                    job j.
12                 if feasible configuration found then
13                     Redistribute resources.
14                     Start the job j.
15                     break;
16                 end
17             end
18             Pick new job from the waiting Queue that satisfies the power requirements.
19         end
20         if power corridor is not broken then
21             If possible, pick a new job from the waiting Queue.
22             Equally distribute available idle nodes among selected jobs.
23             Find the highest priority job(s) that satisfies the power constraints
24             Start the selected jobs.
25         end
26     end
27 return
```

5.2.1 Dynamic Power Corridor Management

One of the necessities of power corridor management is to adapt to the varying power supply from the electricity providers or utilise different energy sources. For example, an HPC center can increase the power budget while using electricity from renewable energy sources. To faciliate such scenarios, the system should support and react to the modification in power corridors.

To enable dynamic power corridor management, the site-level administrators can provide lower and upper corridor values to the power-aware scheduler. Initial values can be given via the configuration file `slurm.conf`. Since the configuration file is read only once by Slurm, the power-aware scheduler provides another special configuration file `power-slurm.conf` that can be leveraged to input the dynamic power corridor values. The value provided is read by `Power_Aware_Schedule` during every `scheduler tick` seconds.

The proposed power-aware scheduler allows the change of lower and upper power corridors externally and dynamically. During each scheduler pass, the scheduler will check for the power corridor change. If a change is detected, the scheduler immediately calls the LP model with the new corridor values and calculates a new resource distribution. As a result, the system can cater to the dynamic power corridor situations.

5.2.2 Limitations

The critical limitation of this approach is the availability of malleable applications. The power-aware scheduler can only keep the system in the power corridor if there are sufficient malleable applications, and resource changes in the available malleable applications significantly impact the overall power usage of the system. The LP model only uses the malleable application for calculating the new resource redistribution. As a result, `Power_Aware_Schedule` will fail to calculate valid resource distribution if there are not enough malleable applications. Hence, the system will continue in the same state with no change in power consumption, which might result in the power corridor violation.

6

iCheck – Invasive Checkpointing System

This chapter presents the **major contribution** of this work, an **invasive (adaptive) checkpointing system**. It begins by introducing the core concepts underlying the proposed checkpointing system. Next, the complete workflow of checkpointing using this system is explained. Following that, the library developed as part of this work is introduced. The various features supported by the checkpointing system are then discussed in detail. Finally, the chapter concludes with an exploration of potential failures in the checkpointing system and the techniques employed to mitigate them.

The state-of-the-art works on application-level checkpointing view checkpointing as a function of data transfer between an application and a memory hierarchy and provides efficient libraries for it. Multiple optimizations and improvements are added to these libraries to improve the individual performance of the applications. The proposed checkpointing system changes the perspective on application-level checkpointing from a library to a holistic data and resource management system. Applying invasive computing concepts (introduced in chapter 2) introduces the dynamic resource management capability to the checkpointing system. Towards that, an **Invasive Checkpointing System (iCheck)** is proposed. This system is specifically designed to offer invasive (malleable) checkpointing services to applications. Its primary design goal centers on

- Allocating resources (such as memory and endpoints) in accordance with the checkpointing needs of applications.
- Adjusting its checkpointing resources (compute nodes) by scaling them up or down based on requirements.

To support the former, iCheck should have malleable library support that the application can use. The iCheck system should be scalable and have flexible components for the latter. iCheck is designed to accommodate these design objectives.

6.1 Architecture

iCheck consists of a core system and a library, as depicted in Figure 6.1. The iCheck core system operates on a set of dynamically configurable dedicated nodes (iCheck nodes) within the HPC system. Using the provided library, the application communicates with the iCheck core system. The application contacts iCheck

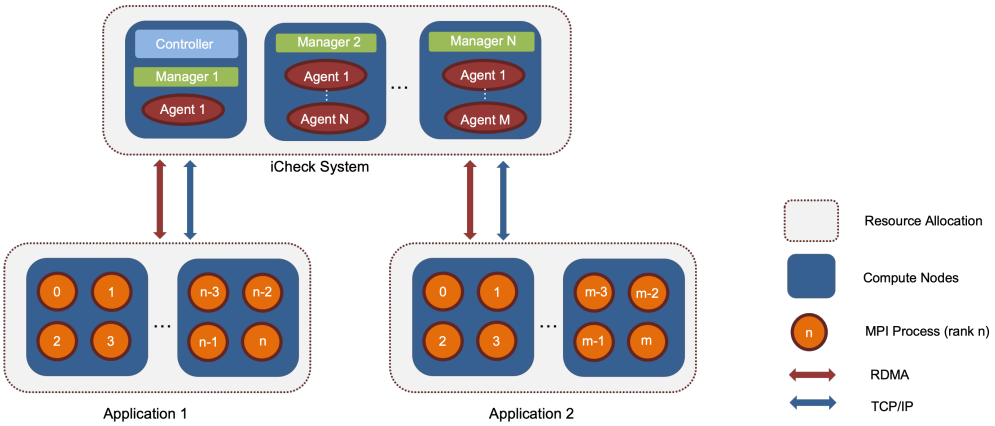


Figure 6.1: iCheck system architecture [77]

for checkpointing as well as configuration needs. iCheck leverages the high-performance high-bandwidth network of HPC systems for checkpoint transfer. iCheck core has different checkpoint transfer mechanisms to utilise the network resources optimally. The subsequent subsections delve into a detailed examination of the various components within iCheck and their functionalities.

6.1.1 iCheck Core

The iCheck core system is a modular system with three components: Controller, Manager, and Agent. They are decoupled to improve scalability, and they communicate with each other via message passing and shared memory. Components communicate asynchronously and are not blocked waiting for messages from each other. As seen from Figure 6.5, iCheck has a hierarchical structure even though the components are decoupled functionally. Each component has well-defined objectives and is defined in the following sections in the increasing order of component functionality.

6.1.1.1 Agent

The agent performs the read/write operation of checkpoint data from/to a connected application. The interaction between agents and an application is facilitated through Remote Direct Memory Access (RDMA) [202] operations and TCP/IP[203]. Agents use RDMA for low latency, high bandwidth checkpoint transfers, and TCP/IP for configuration messages. The agent can transfer checkpoints synchronously (blocking) and asynchronously (non-blocking). An agent can also perform other services like data compression and IO operations to the Parallel File System (PFS) [204] (See Figure 6.2). There can be multiple agents of multiple applications in an iCheck node. The mapping of agents to applications and to iCheck nodes can be defined as $p : q$, i.e., p agents can be used to perform checkpointing on q applications, and p agents can be mapped to q iCheck nodes in any format. The number of agents are not fixed and can be scaled as per the requirements. Checkpoints of applications are stored in the memory of iCheck nodes.

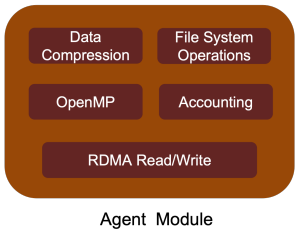


Figure 6.2: iCheck Agent

6.1.1.2 Manager

Every iCheck node is equipped with a manager component that is responsible for overseeing the node-specific activities of the software. This includes launching agents, tracking and forecasting node usage metrics (like memory and bandwidth utilisation) within iCheck nodes, and facilitating communication with the controller. The manager regularly conveys the status of the node, derived from the monitoring infrastructure, to the controller. Additionally, the manager does not engage in direct interaction with the applications.

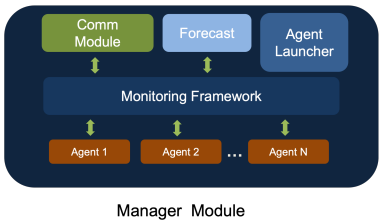


Figure 6.3: iCheck Manager

6.1.1.3 Controller

The controller has a global view of the system and its resources. There is only one controller per iCheck system, and its primary responsibilities include selecting iCheck nodes and scheduling agents. By analysing various metrics such as bandwidth, available memory, and checkpoint frequency, the monitoring framework within the controller assists in determining the optimal number of agents and the specific iCheck nodes to deploy them. Additionally, the controller plays a vital role in liaising with a malleable resource manager. iCheck can operate independently or integrate closely with `ABS`, as discussed in Chapter 7. In an integrated setup with a malleable resource manager, the controller has the capability to request extra nodes or release current ones based on the state of the system. To ensure reliability, a secondary controller is activated in the event of the primary controller's failure.

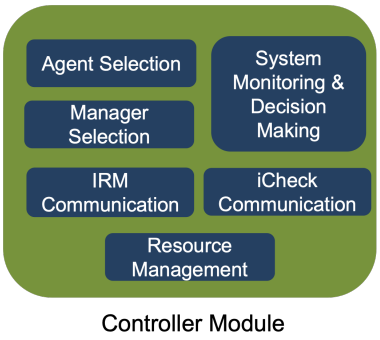


Figure 6.4: iCheck Controller

6.1.2 iCheck Workflow

The components of the iCheck core and an application interact during different phases of the application execution and a broad overview of these interactions is provided below:

During the application start:

1. The application engages with iCheck by registering itself with the controller.
2. The controller determines the initial number of agents and the iCheck nodes to launch these agents.
3. The controller communicates the agent information (identifiers and number of agents to launch) with the managers in selected iCheck nodes.
4. The managers launch the specified number of agents and inform the controller.
5. The agents are now ready (and waiting) to connect with the application processes.

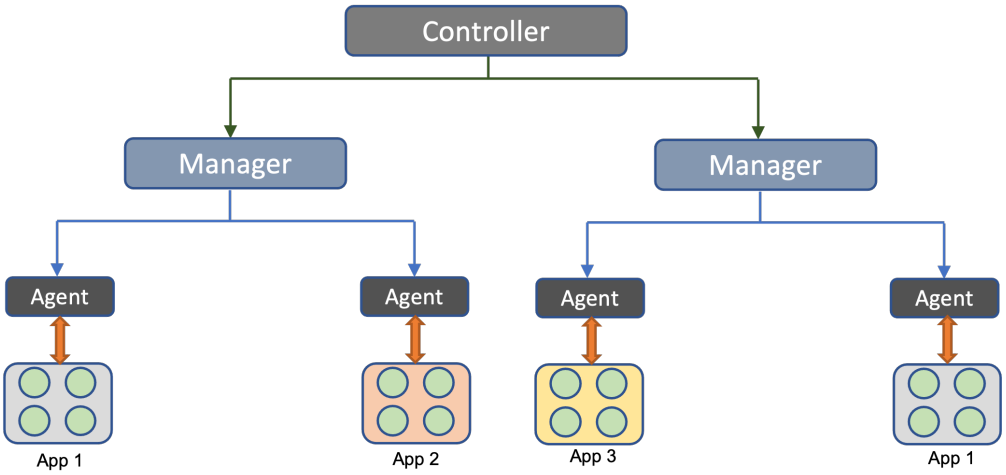


Figure 6.5: iCheck architecture hierarchical view

6. The controller transfers the agent list to the application.
7. The application then registers with the agent.
8. The application and the agents prepare for RDMA operations by registering the memory regions.
9. The application and the agents engages in checkpoint transfer.
10. Application exits after completion (or exit with an error).

During the restart:

1. The application connects with the controller to obtain checkpoint information.
2. The controller transfers the information regarding agents.
3. Application can either retrieve the latest checkpoint from the agents or start fresh.

Applications interact with the iCheck core system using the API provided by the iCheck library.

6.1.3 iCheck library

The design principle behind the iCheck library focuses primarily on three objectives.

- API function calls must be simple
- Support malleability
- Provide fast data transfer with less overhead

The subsections below give an in-depth overview of how these design objectives are attained via the proposed iCheck API and library.

6.1.3.1 API for Checkpoint Transfer

iCheck offers a simple API, explicitly designed for application-level coordinated checkpointing, necessitating only six iCheck API functions to effectively perform checkpoint writing and reading. These include two functions each dedicated to configuration, checkpointing, and restarting processes.

The following two API function calls are used for the configuration of the library:

1. `icheck_init(argv[1], "App Name", Communicator, status);`

   It initializes the library with initial configuration parameters. This call contacts the iCheck controller to get the agent's information. Upon getting the information, the application is connected to the agent. This call must be the first iCheck API function call in an application.

2. `icheck_finalize(IC_PERSIST);`

   It is used to finalize the checkpointing activities in an application. The application informs the controller that the application is finished. Using this information, the controller can update the metadata of the application. Finally, the application will clear the data associated with iCheck. This function call must be the last iCheck API function call in an application.

After the initial configuration, the application uses the two API functions for checkpoint management:

3. `icheck_add("LABEL", data, SIZE, TYPE);`

   It marks data that needs to be checkpointed. The application should specify the pointer to data, size, and type and add a label for the data. The application can add multiple data types to be checkpointed simultaneously. It is only required to call this function once per data (arrays, variable) in an application since it only labels the data, which the commit function will then use for checkpoint transfer. The type of data structure can be `ICHECK_FLOAT`, `ICHECK_INT`, `ICHECK_CHAR`, or `ICHECK_BINARY`.

4. `icheck_commit();`

   This API function call transfers the checkpoint from an application to the agent. The agent will transfer the data associated with data structures marked and labelled using the `icheck_add()` API function call to the iCheck node. In the case of multiple agents, the data is transferred in parallel to the iCheck node. This function should be called regularly to update the checkpoint in the agent.

In the event of an application failure or restart, the application can restore its checkpoint by using the following two API function calls:

5. `icheck_restart();`

   This call will transfer checkpoints from agents to the iCheck library in the application during the restart. It restores the checkpoint from the most recent `icheck_commit()` call. This call should only be called once in the application.

6. `icheck_restore("LABEL", data, SIZE);`

   This call will copy the checkpoint that matches labels (added via the `icheck_add()` call) to the application data structure from the iCheck library. A special API call `icheck_restart_all()` can combine the above two API calls into a single call. It will transfer all the data checkpointed using the `icheck_add()` call at once to the application. The prerequisite for this API call is the `icheck_add()` call.

The above-detailed API calls are utilised to integrate iCheck into a simple application, as illustrated in the pseudocode of a basic iCheck-enabled application in Listing 6.1. Initially, the application includes the `icheck.h` header file. The function call `icheck_init()` (shown on line 5) is crucial as it registers the application with the iCheck controller and conveys any available hints about it. The `icheck_init` function requires the `MPI_COMM_WORLD` communicator, the application's name, and its status as arguments. In cases where the application is not MPI-based, `NULL` can be substituted for the communicator. Subsequently, the application is provided with a list of agents from the controller, enabling it to register with these agents to use it for future RDMA operations.

`icheck_add` (in line 7) registers the array *data* to the iCheck library, its size, and assigns a *label* for this data. In line 17, the `icheck_commit()` function signals the library to transfer all data previously specified using `icheck_add` into a designated buffer for remote data transmission and informs the agents that the checkpoint data is ready. Following this, the agents remotely access and retrieve the checkpoint data. Later, in line 22, the `icheck_finalize()` function is called to notify the controller that the application's execution has concluded. The argument provided in the finalise call specifies whether the checkpoint data should be preserved in the agent's memory post-application termination. Utilising `IC_PERSIST` as an argument directs iCheck to maintain the checkpoint data within the agent's memory. Otherwise, the agents are killed during the application finish, and the checkpoint is transferred to the file system. `icheck_restart` (in line 10) will transfer the checkpoint from the agent to the library. The `icheck_restore()` (in line 11) will transfer the data associated with the *label* from the library to the data structure reference passed in the call.

```
1  #include<icheck.h>
2  int main() {
3    MPI_Init(NULL, NULL);
4    float data[size];
5    icheck_init(argv[1], "testApp", MPI_COMM_WORLD, status); // Initialising iCheck
6    /* Registering the data to be checkpointed */
7    icheck_add("mydata", data, size, ICHECK_FLOAT);
8    if(checkpoint_available) {
9     /* In the event of application restart */
10    icheck_restart();
11    icheck_restore("mydata", data, size);
12   }
13   /* Perform computatio n*/
14   for(i = 0; i<N; i++) {
15    /* Modify the values in data[] */
16    if(i%10)
17     icheck_commit(); // Performing checkpoint transfer
18    /* Perform computation */
19    if(i%2000)
20     icheck_probe_agents(hints); // Looking for resource change
21   }
22   icheck_finalize(IC_PERSIST); // End of iCheck
23   MPI_Finalize();
24 }
```

Listing 6.1: Pseudocode of a simple iCheck enabled MPI application [77].

6.1.3.2 API for Malleability Support

One of the design objectives of the iCheck library is to support the malleability or dynamism of resources. As seen from Subsection 6.1.1.1, agents are scalable and in charge of checkpoint transfer. As a result, this dynamism in agents can be utilised to improve the application performance. For the application to benefit from such a malleable design and to evoke dynamism in the iCheck system, icheck offers the following special API function that allows applications to adapt to changes in iCheck's allocation of resources.

- `icheck_probe_agents(hints);`

  This call will contact the iCheck controller to enquire about any agent change. If the controller triggers agent change, this call will reinitialize the library to use the new number of agents. Hints about application characteristics can be passed to the controller using the hints section. As a hint, an application can pass the time taken for the most recent checkpointing operation.

In line 20 of the Listing 6.1, the usage of this API call can be seen. When the application invokes the probe function, it obtains updates regarding any changes to the agents. If agent changes occur, the next `icheck_commit()` call will transfer the checkpoint using the new number of agents.

6.1.3.3 API for Faster Data Transfer

The last design objective is to provide fast data transfer with less overhead. To reduce the impact of checkpointing calls on the application execution, iCheck supports asynchronous checkpoint transfer. This can be turned on and off dynamically at any point in application execution using a single API call.

- `icheck_enable_asynchronous(FLAG);`

  `icheck_enable_asynchronous(true)` call will notify the iCheck library to perform the checkpoint transfer in a non-blocking manner till `icheck_enable_asynchronous(false)` is encountered. A caveat is that the application developer must ensure that the application does not overwrite the data until the checkpoint transfer is complete.

If enabled, the `icheck_commit()` call will return immediately, and the iCheck library will perform the data transfer in the background. This makes sure that the overhead is minimal with iCheck.

To support faster data transfer, iCheck uses Remote Direct Memory Access in `icheck_commit()`, thus leveraging the underlying high bandwidth hardware of the HPC system to get the maximum performance during the checkpoint transfer. The following subsection gives an overview of how RDMA is used in iCheck.

6.2 Data Transfer in iCheck

RDMA enables a remote process to access data in pre-registered memory regions without CPU involvement, thereby enhancing throughput and minimising roundtrip latency. To integrate RDMA support into iCheck, we utilise the libfabric library [205]. The OpenFabrics Interfaces (OFI) [206] offer programming interfaces designed for crafting high-performance and scalable remote memory access operations. Libfabric provides access to the user-space API of OFI services, facilitating direct access to these advanced networking capabilities. `fi_mr` routines are used for memory region registration, and `fi_read` and `fi_write` remote memory access operations are utilised by the iCheck agents and the library for the checkpoint transfer. Libfabric

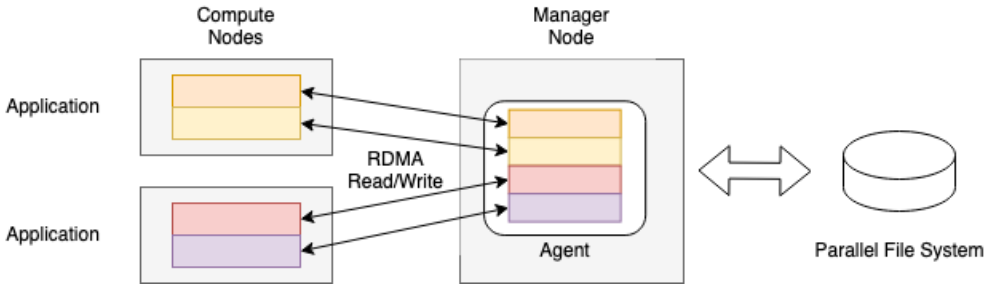


Figure 6.6: RDMA in iCheck

supports a wide variety of networking hardware and high-performance fabrics, including Omni-Path [207], used in SuperMUC-NG [56], where the iCheck system is evaluated.

In iCheck, the execution of Checkpoint and Restart operations can be conducted in two different forms, which are based on the RDMA operations that libfabric supports: read and write (as illustrated in Figure 6.6). iCheck employs a blend of these operations to facilitate the storage and retrieval of checkpoint data. Specifically, iCheck supports two techniques for data transfer during the checkpoint/restart processes: *Push* and *Pull*. These techniques dictate how data is moved between the application and the checkpoint storage during the checkpointing and restarting phases. Though read and write operations are used in both techniques, the core difference between these two techniques is around the initiator of the checkpoint transfer.

- *Push Technique*: In this technique, the application writes the checkpoint data in to the memory of agents during the `icheck_commit()` call. During the application restart, inside the `icheck_restart()` call, the application reads the checkpoint directly from the agent's memory.
- *Pull Technique*: In this technique, the agents read the checkpoint from the application's memory during the `icheck_commit()` call and write the checkpoint into the application's memory during the `icheck_restart()` call.

Furthermore, agents can leverage multithreading to parallelise the checkpoint operations. These strategies are configurable in iCheck and can be used interchangeably during the application execution. The *Pull Technique* presents many optimization opportunities that are discussed in Section 7.1.

For both techniques, the concept of memory regions in libfabric is employed. The agents register memory regions in iCheck nodes to store the checkpoints and provide permission to the iCheck library to access the corresponding memory regions. Similarly, from the application side, the iCheck library must register the memory regions and give access to the agents. It can be accomplished in two ways and is configurable per app. The first technique is the Memory Region approach (*MR* approach), which uses memory regions in the individual processes, and the second is Shared Memory Region approach (*SHMR* approach), where shared memory is used for the memory regions.

6.2.1 MR Approach

This technique registers the memory regions of all the participating processes so that agents can directly read the checkpoints. For example, consider an application with P processes, N nodes, and iCheck with M agents. There will be $P : M$ mapping, where P memory regions will be registered across M agents, and P RDMA operations are performed across M agents during every checkpoint transfer. If t_i is the time taken for

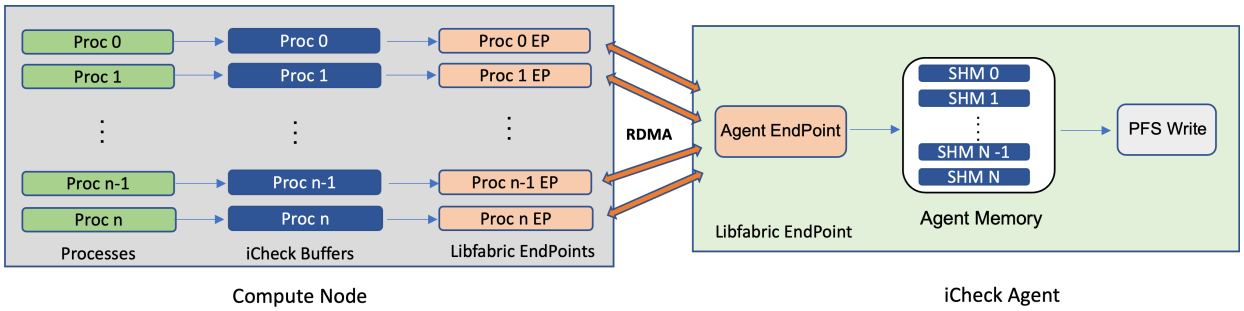


Figure 6.7: Buffer Management in MR based checkpointing

data transfer from the i^{th} process to an agent, the overall time taken for synchronous checkpointing (t_{sync}) is

$$\begin{aligned} t_{\text{transfer}} &= \max\{t_1, \ldots, t_P\} \\ t_{\text{sync}} &= t_{\text{msg+copy}} + t_{\text{transfer}} \end{aligned} \tag{6.1}$$

where, $t_{\text{msg+copy}}$ is the time taken for sending the checkpoint ready message to agents (t_{msg}) and copying the message from the application data structure to the iCheck allocated memory in the library (t_{copy}).

6.2.2 SHMR Approach

In this technique, one process per node will create a shared memory region and provide the agents access to the shared memory to retrieve the checkpoints. For example, consider an application with P processes, N nodes, and iCheck with M agents. There will be an $N:M$ mapping, where N memory regions will be registered across M agents, and N RDMA operations are performed across M agents during every checkpoint transfer. If t_i is the time taken for data transfer from the i^{th} node to an agent, the overall time taken for synchronous checkpointing is

$$\begin{aligned} t_{\text{transfer}} &= \max\{t_1, \ldots, t_N\} \\ t_{\text{sync}} &= t_{\text{msg+shmcopy}} + t_{\text{transfer}} \end{aligned} \tag{6.2}$$

where, $t_{\text{msg+shmcopy}}$ is the time taken for sending the checkpoint ready message to agents (t_{msg}) and copying the message from the application data structure to the iCheck allocated shared memory in the compute node (t_{shmcopy}).

6.2.3 Buffer Management in iCheck

The buffer management for both the MR approach and the SHMR approach are visualised in Figures 6.7 and 6.8, respectively. The fundamental concept behind both approaches is described in the above sections. As seen in both figures, there is a libfabric endpoint in the agents for both strategies in an iCheck node. Meanwhile, on the library side, there is a single libfabric endpoint per compute node in the SHMR approach and an endpoint per process in the MR approach. The libfabric library manages data transfer between these endpoints, and iCheck initiates (read/write) data transfers using remote memory access calls and tracks the progress using the libfabric counters. In libfabric, endpoints are communication portals that function at the transport level [208]. There are two types of endpoints in libfabric, namely passive and active. The former is used for listening to connection requests, while the latter is used for performing data transfers.

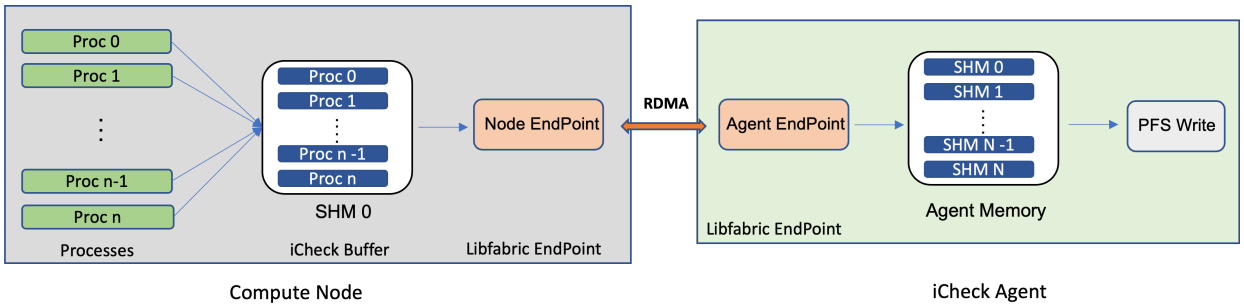


Figure 6.8: Buffer Management in SHMR based checkpointing

iCheck creates and utilises the active endpoints for checkpoint transfers. In both the SHMR and MR approaches, the iCheck library allocates a buffer in the user space of the compute node, and the iCheck agents allocate memory with the corresponding size in the agent. These allocated buffers are then registered as memory regions in the fabric domain and bound to the endpoints. Hence, performing a read/write from the agent will result in remotely accessing these memory regions inside the application to send and receive the data. However, this induces an issue where sufficient memory is unavailable in the allocated buffer (or memory region) to accommodate all the checkpoint data. Towards that, a pipelining technique is employed by iCheck, where iCheck creates a limited buffer size, and the checkpoints are transferred in chunks iteratively between the application and agents. A brief description of the pipelining technique in iCheck is provided in the section below.

6.2.3.1 Pipelining

The iCheck library creates a buffer of limited size (the buffer size will be a factor of the checkpoint size) in the application process and is registered as a memory region in the libfabric during the initialisation. Upon preparing for a checkpoint transfer, the library communicates with the iCheck agent out-of-band, detailing the total size of the checkpoint, the size of each chunk to be read, and the total number of reads required to transfer the checkpoint data fully. iCheck agent reads the remote memory iteratively till all the data is transferred to the agent. The iCheck agent iteratively retrieves the data from remote memory, continuing the process until the complete dataset is transferred. Depending on the availability of memory on the agent side, the data can either be stored entirely in memory or written incrementally to the file system as each chunk is read. This process is illustrated in Figure 6.9.

For scenarios where the initial iCheck buffer cannot accommodate the entire checkpoint, the Checkpoint Manager (CP Manager) and a Coordinator are employed to orchestrate the segmented transfer of the checkpoint data. The CP Manager on the application side segments the data and feeds it into the iCheck RDMA buffer, while the Coordinator signals the agent to retrieve the data and informs the CP Manager when to load new data chunks into the buffer. On the agent side, the coordinator oversees the iterative data retrieval, guided by the total chunk size and updates from the application's coordinator. If the agent lacks sufficient memory, the CP Manager is equipped to write the data chunks directly into PFS. This pipeline technique facilitates a structured and efficient checkpoint commit operation across both the application and agent sides, as detailed in the subsequent workflow description.

Application side: The pipelining technique in iCheck for SHMR strategy can be abstracted in the following seven steps:

1. Application process calls the checkpoint (`icheck_commit`) operation.

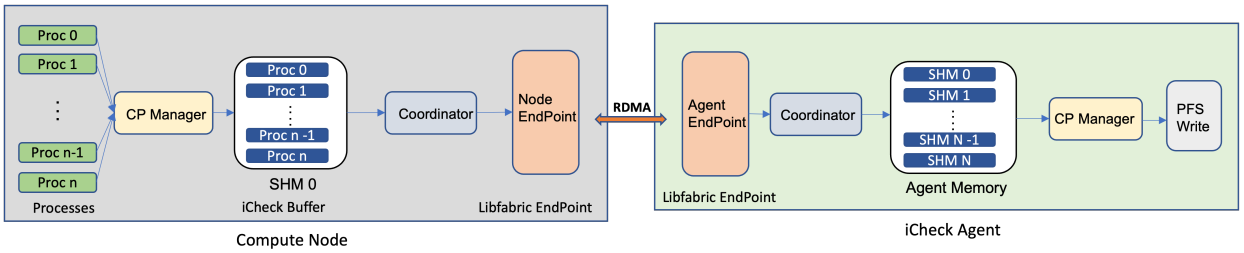


Figure 6.9: iCheck Buffer Management in SHMR based checkpointing using the CP Manager and Coordinator

2. iCheck library inside the application process passes the checkpoint metadata (checkpoint size, pointers to the data structures) to the Checkpoint Manager (CP Manager).
3. CP Manager, based on the total iCheck buffer size, copies the chunk of data from the data structures and notifies this information to the iCheck coordinator.
4. iCheck coordinator will signal the agent that the checkpoint is ready to be copied.
5. The agent receives the signal, copies the checkpoint data and signals the coordinator data is read.
6. Coordinator informs the CP Manager that the buffer can be reused.
7. CP Manager copies the next chunk into the RDMA buffer.
8. Steps 4 - 7 will continue until all the checkpoint data is transferred to the agents.

Agent side: The pipelining technique can be abstracted in the following six steps:

1. Agent receives checkpoint metadata such as total size, chunk size, and number of reads to perform.
2. The coordinator in the agent reads the initial chunk of data, copy into the agent's memory.
3. The coordinator notifies the application coordinator and waits for the next signal to read.
4. CP Manager, based on the agent's available memory, copies the chunk into the PFS or remain idle.
5. Steps 2 - 4 will continue until all the checkpoint data is transferred from the application.
6. CP Manager will update the version metadata (For example, version number, time, size) of the checkpoint.
7. CP Manager will duplicate checkpoint into the PFS (frequency of this operation can be configured) if the checkpoint was stored entirely in the agent memory in Step 4.

The pipelining process during the checkpoint restore operation mirrors the commit operation's conceptual framework, with a key difference in the direction of data flow. In this phase, agents write the checkpointed data back into the application in segmented chunks rather than reading it from the application. The Checkpoint Manager (CP Manager) merges these chunks into the application's data structures. Following each successful chunk transfer, the coordinator in the application signals the coordinator in agents to initiate the subsequent chunk transfer until all the checkpoints are restored. This ensures a systematic application checkpoint restoration, leveraging a reverse workflow that complements the initial data checkpointing process.

6.2.3.2 Versioning

Checkpoint versioning is essential for ensuring the validity and consistency of transferred checkpoints. It guarantees that agents have accurately transferred the necessary checkpoints from the application, allowing them to be utilised for recovery purposes if required. This aspect becomes crucial in scenarios where the application experiences a failure during the checkpoint transfer process. Within the agents, the Checkpoint Manager (CP Manager, as depicted in Figure 6.9) oversees the versioning of iCheck checkpoints. Following every successful checkpoint operation, the CP Manager updates the checkpoint's metadata, which encompasses the checkpoint's version, size, label and timestamp. The CP Manager increments the version count of the checkpoint with each successful operation. Additionally, the CP Manager is configured to save the entire checkpoint and metadata to PFS periodically. It overwrites the existing checkpoint in PFS only if all CP Managers align on the same version number and the previous version number is exactly one less than the current version. This systematic approach ensures checkpoint data integrity and reliability, facilitating smooth recovery.

6.2.4 Asynchronous Checkpointing

iCheck seamlessly incorporates asynchronous checkpoint retrieval into its library through its agent-based model. Utilising RDMA, the agents enable applications to offload data transfer tasks, allowing them to resume execution right after signalling the agents about the checkpoint availability. This mechanism ensures that the agents can fetch the data remotely without affecting the performance of the application.

Suppose the total time taken in an application to inform an agent about the new checkpoint and to copy the checkpoint to the registered memory region is $t_{msg+copy}$, the time taken by the agent to transfer data from the application is $t_{transfer}$, and the checkpoint interval inside the application is $CP_{interval}$. In that case, the checkpoint time t_{async} associated with non-blocking checkpoint restart can be defined as in Equation 6.3.

$$t_{async} = \begin{cases} t_{msg+copy}, & \text{if } t_{transfer} \leq CP_{interval}, \\ t_{msg+copy} + t_{transfer} - CP_{interval}, & \text{if } t_{transfer} > CP_{interval} \end{cases} \tag{6.3}$$

If the $CP_{interval}$ is less than the time taken for transferring the data from application to agent ($t_{transfer}$), then the benefit associated with asynchronous checkpointing is proportional to the $CP_{interval}$. As smaller the value of $CP_{interval}$ becomes, the t_{async} value will reach near the value of t_{sync}. The overall time taken for synchronous checkpointing (t_{sync}) is instead

$$t_{sync} = t_{msg+copy} + t_{transfer} \tag{6.4}$$

However, the performance advantage of asynchronous checkpointing may not be fully leveraged when the pipelining technique is utilised for checkpoint transfer in iCheck. Coordination needs to be done between application processes and the agents to ensure a valid checkpoint transfer, and the application should only modify the checkpoint data once the agents have copied the chunks associated with a particular process. Hence, the processes need to wait (or be synchronous) until the agents transfer the chunks associated with it. If the pipeline is unutilised, the application can fully leverage the benefit of asynchronous checkpointing.

6.2.5 Multilevel Checkpointing in iCheck

iCheck is a two-level checkpointing system. In the first level, iCheck keeps the checkpoints in the main memory of iCheck nodes. iCheck uses the underlying IO file system of the machine as the second level to store the checkpoint. Typically, there will be Parallel File Systems (PFS) to support quick IO transfer in HPC centers. For the first level checkpoint transfer, both iCheck and application are involved, while in the second level, only iCheck agents are involved in writing into PFS. Hence the second level is transparent to the application, and the performance of the application is not affected (See Figure 6.6).

During the application restart, iCheck agents will utilise the data stored in PFS. The controller can deploy these agents on any iCheck node, allowing them to load checkpoints from PFS and provide checkpoint services. This data acts as a backup, protecting the application against agent failure or checkpoint corruption. Moreover, iCheck can simulate agent migration using this layer. Agents can store data in PFS before termination, and new agents on a different iCheck node can then access this data, effectively mimicking a full agent migration.

Compression of checkpoint data is an efficient way to reduce memory usage[209]. iCheck can compress checkpoint data before writing it into the parallel file system. Compression and writing to PFS can be dynamically turned on and off inside the iCheck per-app basis and is configurable based on the application characteristics. For example, icheck can be configured to compress checkpoints before writing into PFS when the checkpoint size is greater than 1GB.

6.3 Monitoring

iCheck has a monitoring framework to obtain the performance data of a compute node. Plenty of performance data can be obtained from a compute node using various system tools in an HPC system, and most of them are not relevant in the context of a checkpointing system. A set of relevant metrics that can identify the appropriate characteristics of a node must be first defined. It is essential for fine-tuning the checkpointing system. Metrics associated with memory, data transfer, checkpoint operations, and power are relevant in iCheck optimization. Hence the monitoring framework in iCheck obtains data associated with these metrics and passes it to the controller. The metrics used in iCheck are defined in the below subsections.

6.3.1 Memory

Memory usage is one of the critical information needed by a checkpoint system. The iCheck controller must be aware of the memory usage in iCheck nodes to make the agent scheduling decision. Lack of memory during checkpointing can lead to an undefined behavior in iCheck. Hence the monitoring framework in iCheck will periodically keep track of the available memory (*avail_mem*) in each iCheck node.

The controller calculates the memory depletion rate with the available memory usage information *avail_mem* from managers. Suppose the available memory in a controller node at time t is $avail\_mem_t$, then the memory depletion rate after i seconds will be

$$MemoryDepletionRate = \frac{avail\_mem_t - avail\_mem_{t+i}}{i} \tag{6.5}$$

Using this information controller can perform horizontal scaling of iCheck nodes to replenish the memory pool. In addition to the monitoring, iCheck estimates the maximum and minimum memory usage for a time

interval using time series analysis. This forecast and the memory depletion rate can be used to identify the system's memory usage trend.

In total, the following four memory-related metrics are tracked for iCheck nodes.

- Available Memory
- Memory depletion rate
- Estimated minimum and maximum memory usage

6.3.2 Checkpoint Operations

After memory usage, another critical metric is the number of checkpoint operations happening inside an iCheck node. Since multiple agents belonging to multiple applications can perform checkpoint transfer in a single iCheck node, it is essential to comprehend and record the checkpointing events. Using this metric, the controller can find the most active iCheck nodes and use them for agent placement decisions.

The first metric it records is the total number of Checkpoint Operations (*ncops*) performed in an iCheck node. For N agents in a node with x number of checkpoints at a time t, *ncops* can be defined as

$$ncops = \sum_{n=1}^{N} x_n + ncops \tag{6.6}$$

In addition, the framework also calculates the interval between subsequent checkpoint operations (let cop_i denote the checkpoint operation at the i^{th} interval) in an iCheck node, which is defined as

$$\Delta cop = cop_{i+1} - cop_i \tag{6.7}$$

As the last metric, the number of checkpoint operations per minute ($ncops/min$) on an iCheck node is also calculated.

Analyzing these events will provide an overview of the nodes' activities and can be used later for agent configuration. These metrics are consequential alongside the memory usage because frequent checkpointing within a particular node can affect the performance of all applications checkpointing into that node. Even though available memory in that particular iCheck node might be high, adding more applications exacerbates the performance issues. As a result, it is necessary to maintain a careful balance between memory usage and checkpointing frequency during agent placement.

6.3.3 Bandwidth

Another crucial metric for checkpointing is the data rate or bandwidth. Bandwidth can be defined as the ratio of the amount of data transferred to the time taken. In the iCheck node, two categories of bandwidth come into play. ❶ The data transfer rate from an application to the agent and ❷ the rate of overall data transfer by all the agents in an iCheck node. The first information can be used to decide the number of agents, and the second information can be used to place the agents.

The monitoring module tracks the rate at which the agents read or write the checkpoint data. It will aggregate the agents' data and summarize the overall IO bandwidth usage in the node. If the time taken by an agent to transfer a checkpoint cp with the size $size(cp)$ is t, then the bandwidth (bw) can be defined as

$$bw_{agent} = \frac{size(cp)}{t} \tag{6.8}$$

The aggregate bandwidth of the manager with N agents can be defined as

$$bw = \sum_{n=1}^{N} \frac{size(cp)}{t} \tag{6.9}$$

6.3.4 Agent Count

The controller creates agents for an application and places them in different iCheck nodes based on the above performance indicators. There might be a scenario where multiple iCheck nodes have similar values, and the controller needs to find ideal candidates to place the agents. In such a scenario, the metric agent filling rate can be used.

If the number of agents that are launched at a time point t is $\#agents(t)$ and the new number of agents launched at time point $t + \Delta t$ is $\#agents(t + \Delta t)$, then the agent filling rate in an iCheck node ($agent\_fill$) can be defined as

$$agent\_fill = \frac{\#agents(t + \Delta t) - \#agents(t)}{\Delta t} \tag{6.10}$$

The controller keeps track of the number of agents across different iCheck nodes and calculates the agent filling rate periodically. The controller also sets a variable (max_agent_count) for every iCheck node. Ideally, this is set to prevent further allocation of agents again in a specific iCheck node. Values for maximum agent count max_agent_count can be provided in the configuration file provided by iCheck (See Listing 6.2 in Subsection 6.3.6).

6.3.5 Power Usage

The monitoring framework periodically tracks the iCheck node's power usage along with memory, bandwidth, and checkpoint operations. This information can be used for the power-aware scheduling of iCheck agents. The power value is obtained using the linux profiling tool `perf` [210].

6.3.6 iCheck Configuration File

iCheck utilises two kinds of configuration files, ❶ static and ❷ dynamic. They have different use cases in the iCheck ecosystem and are explained in detail below:

- Static configuration file: This configuration file handles the configuration data associated with communication in iCheck. Listing 6.3 provides a sample configuration file. Using this configuration file, information like the controller's address and port, the managers' address, and network information of the malleable resource manager can be provided to the iCheck system and applications. In addition, the library in the application reads this configuration file to get the information about the iCheck controller. Furthermore, the interval in which the heartbeat messages need to be sent between controller and manager nodes is also defined in this file (See $HeartBeatInterval$ in Line 5 of Listing 6.3. HeartBeat messages are used to ensure whether an iCheck node is alive or not. The static configuration file is only read once by the iCheck system.

```
readInterval = 30 #in seconds
max_agents_node = 80
max_agents_app = 80
max_mem_node = 90 #in GB
init_agent_count = num_nodes #default strategy
candidate_count = 5 #candidates to pick from for agent distribution
manager_count = 10
agent_selection = BW #BW for Bandwidth, CP for checkpoint frequency
node_selection = RR #RR for roundrobin, EQ for equal, BLK for block
is_dynamic_as = true #allows agent scaling
is_dynamic_hs = true #allows horizontal scaling
enable_multi_level = true
force_config = false #always take this configuration
```

Listing 6.2: iCheck dynamic configuration file.

```
ControllerAddress = hostname1
iCheckManagerNode = hostname1
iCheckManagerNode = hostname2
iCheckManagerNode = hostname3
HeartBeatInterval = 5
StatusMessageInterval = 5
iRMNodeName = hostname5
iRMiCheckPort = 9999
ControllerPort = 7777
```

Listing 6.3: iCheck static configuration file.

- Dynamic configuration file: This file is used to pass/update the configuration information to iCheck. This file is read periodically (every *readInterval* seconds) by iCheck. The interval in which this file must be read is passed via the dynamic configuration file itself. Initially, this file is read along with the static configuration file during the application launch. After that, based on the *readInterval* defined in the file (See Line 1 in Listing 6.2), it is read periodically. With this file, the administrator can pass the *agent_count*, *node_selection_strategy* and other relevant metrics that can affect the overall performance of the checkpointing services. Listing 6.2 shows some parameters that can be changed periodically.

6.4 Dynamism in iCheck

One of the key features of iCheck is that it is a fully malleable (adaptive) checkpointing system. iCheck is dynamic both at the system level and application level. iCheck offers system-level malleability by scaling iCheck nodes, tuning agent behavior, and application-level malleability by scaling agents and supporting malleable applications.

6.4.1 System Level

At the system level, iCheck can ❶ horizontally scale its resources and ❷ change the agent's behavior (turn features on/off) based on the system/application characteristics or requirements. An overview of these system level features is provided below:

- **Horizontal Scaling:** iCheck controller periodically tracks the metric *avail_mem* of all iCheck nodes. Whenever it observes that a node reaches its maximum memory capacity, the controller can add additional iCheck nodes dynamically to support new applications. iCheck also supports the removal of nodes on the fly if a node remains unused for a long time. In short, horizontal scaling requires a malleable resource manager to dynamically add/remove nodes from iCheck. iCheck uses `iRM` for malleability and it is discussed extensively in Chapter 7.
- **Agent Behaviour:** Multiple applications can be connected simultaneously to iCheck, and their agents are deployed across iCheck nodes. Agents offer various services like data compression, encryption, data distribution, and PFS write/read. These services are pluggable and are visualized in Figure 6.10. For example, agents of all applications do not need to call compression service, only the agents that checkpoint data above a limit (*MAX_CP_SIZE*). The initial value of *MAX_CP_SIZE* can be configured dynamically in iCheck (See Listing 6.2 in Subsection 6.3.6). Similarly, data distribution service is only relevant to malleable applications. These services are pluggable in iCheck and can be enabled/disabled using the iCheck configuration file (See Subsection 6.3.6).

  Multiple services can also be combined for one application but not for another. Consider two applications, *App A* and *App B*. *App A* can use a combination of compression and PFS, while *App B* can use data encryption and PFS. This information can be passed as hints to the agents during application initialization.

  In addition, the controller can decide to leverage services, such as enabling compression and writing data to PFS, based on the current system status. Nevertheless, these services are explicitly added to the agents if provided via application hints. Furthermore, new services can be easily added and extended due to the pluggable architecture of iCheck.

6.4.2 Application Level

At the application level, iCheck can ❶ dynamically change the agents of malleable and non-malleable applications and ❷ support malleable applications. Each of these attributes of iCheck are described in detail below:

- **Agent Dynamism:** iCheck can dynamically reconfigure the number of agents associated with a connected application by increasing or decreasing the number of agents (Figure 6.11). An application does not need to be malleable to utilise the dynamism offered by iCheck. As described in Subsection 6.1.3.2, the applications can utilise the `icheck_probe_agents()` function to check for any changes in their agent configuration. There are the following three possible outcomes to a probe call:
  - **Change in agent count:** An agent change can result in more or fewer number of agents. The controller decides this based on the agent count selection algorithm defined in Section 7.1.1. The objective of agent change is to improve the checkpoint performance.
  - **No change in agent count:** If there were no performance improvements due to the previous agent change, the controller decides not to change the agents further for this particular application unless it is a malleable application. Malleable applications can change their computing resources in the meantime, hence, change in agent count can be beneficial for their checkpoint performance.
  - **Change in agent location:** The controller can ask an application to connect to a different set of agents on different nodes if the controller needs to free up iCheck nodes. A potential scenario could be a malleable resource manager requesting the controller to give back some iCheck nodes.

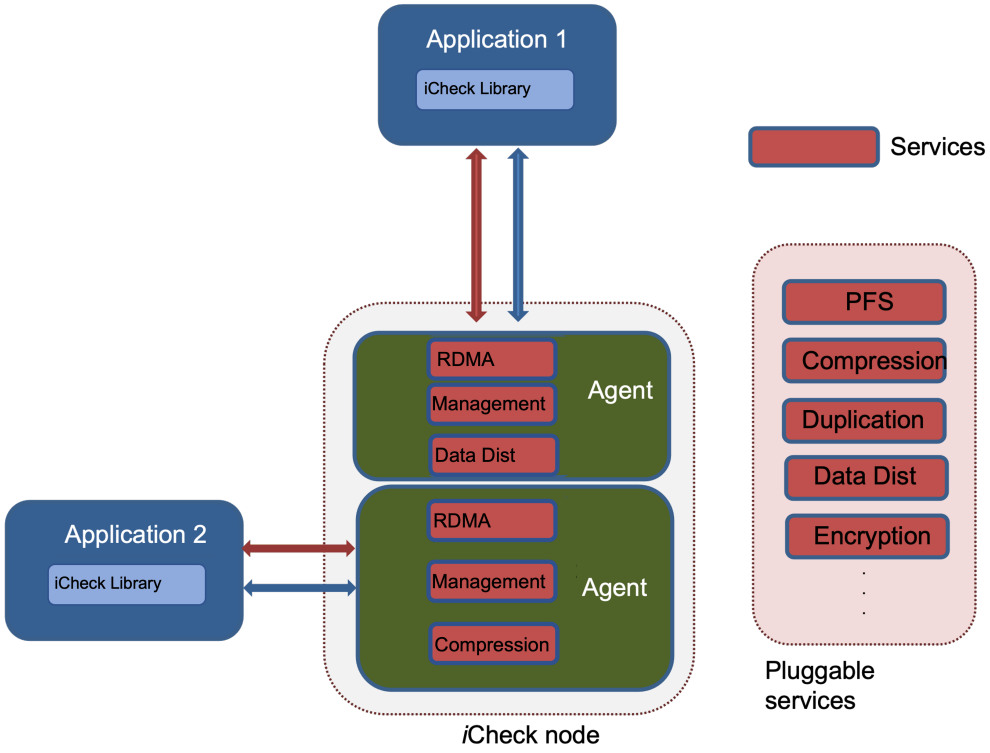


Figure 6.10: Pluggable services in an iCheck agent

Dynamic agents in iCheck will be beneficial under certain conditions, particularly in the context of performance during checkpoint operations. Consider the following scenario where t_{probe} represents the time required for dynamic agent reconfiguration, $t_{old\_transfer}$ denotes the time to transfer a checkpoint with the current number of agents and $t_{new\_transfer}$ is the time needed to transfer a checkpoint with the newly configured number of agents. The potential number of checkpoint operations ($ncops$) that could occur with the old set of agents while the new agents are being configured is calculated as follows:

$$ncops = \frac{t_{probe} + t_{new\_transfer}}{t_{old\_transfer}} - 1 \tag{6.11}$$

This formula helps to determine the threshold beyond which the new configuration starts to provide a performance benefit. The performance improvement from the dynamic agents will commence from the n^{th} checkpoint operation, where n is defined as:

$$n = \frac{ncops \times t_{old\_transfer}}{t_{new\_transfer}} + 1 \tag{6.12}$$

If the number of remaining checkpoint operations is fewer than n, then employing dynamic agents will not enhance the application's performance; instead, it could potentially worsen the application's performance. This analysis is crucial for determining when the adjustment of agent numbers will actually translate to a net gain in efficiency.

- **Malleable Applications:** iCheck supports malleable applications in the following ways:
  - Support checkpoints of varying size.

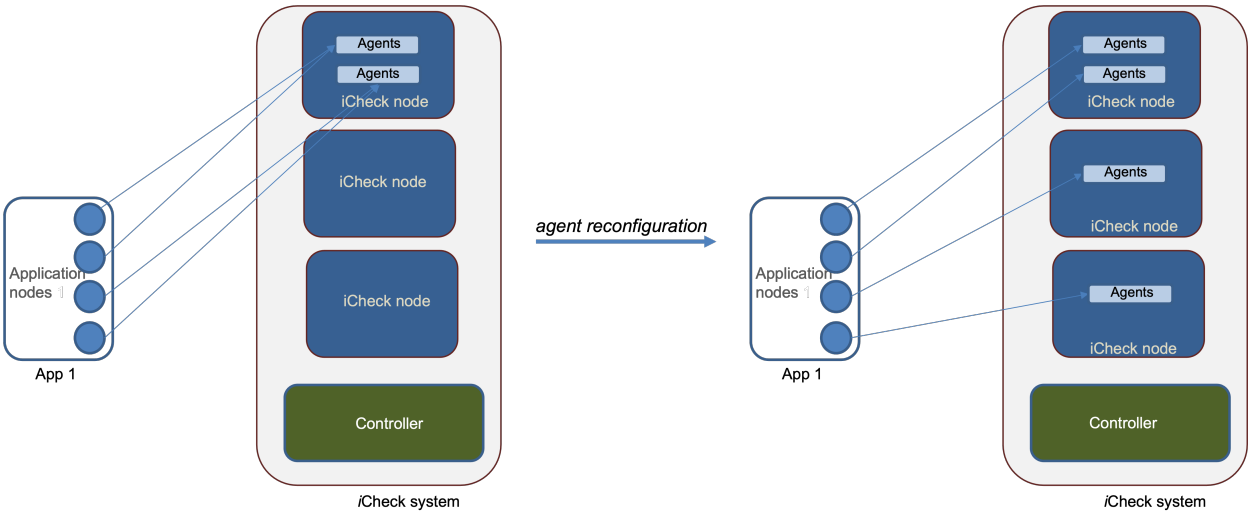


Figure 6.11: Agent reconfiguration

- Performs agent change during resource adaptation.
- Supports data redistribution during resource change.
- Performs agent change based on the application performance.

The malleability aspect of iCheck system is covered extensively in Chapter 7.

6.5 Failures in iCheck

The above sections offered a glimpse into iChecks checkpointing activities that provide fault tolerance to applications connected to the iCheck system. This section discusses the aspect of failures in iCheck and the techniques employed to overcome those failures. Following two subsections discuss in detail the scenarios where the failure of different iCheck components affects the execution of the application. The first section outlines the failure of iCheck from an application perspective, and the second section explores the failures from the component side of iCheck.

6.5.1 Application Perspectives to Failure

The iCheck library in an application connects to the controller using TCP/IP and RDMA. From the application perspective, a failure can be a communication issue. The communication issues can be manifested as an unresponsive controller or agent or incomplete data transfer.

Various communication issues can happen during the different parts of the application execution and are summarized below.

- **Failure during initialization:** The application must connect to the controller to get the essential information to avail checkpointing services from iCheck. However, the application might detect an unresponsive controller. It can be attributed to controller failure or networking issues. In such a scenario, the application can utilise the secondary controller. If the primary controller is unresponsive, the library will retry to connect to the secondary controller. If the series of attempts fails to connect to

the secondary controller, the iCheck library will write the checkpoint directly to the parallel file system without the agent intervention. iCheck on the library side is also equipped to perform checkpointing directly to the parallel file system and will be used in exceptional scenarios. It will not be as efficient as using RDMA, but it ensures that if the iCheck core system is unreachable, the application can continue its execution without failing.

- **Failure during agent connect:** If the agents are unresponsive to an application, the library will notify the controller, and the controller will provide new agent information. The application will revert to the fail-safe PFS technique if the agents are unreachable. It is an unlikely scenario since the controller will only hand over the agent information if the agents are launched successfully.

  If the application finds an agent unresponsive during the RDMA configuration after connecting to the agent, the library will try to contact the controller for new agent information. If the issue persists, the library will revert to the PFS technique.

- **Failure during agent checkpoint transfer:** This case can have two potential failure scenarios. First, the agent is unresponsive while connecting to the application to initiate RDMA transfer. After multiple retries, the library will decide to use the PFS technique and immediately notify all the application processes about this decision. Second, the agent fails while performing RDMA. In such a scenario, the library will inform all the processes to revert to the PFS technique even though only a single agent is failed.

- **Failure during restart:** Similar to the first scenario, if the communication with the controller is unresponsive, the library will connect to the secondary controller and continues the normal execution. If it fails, the library will read the checkpoint from PFS directly to resume the application execution.

6.5.2 System Perspectives to Failure

Different component failures in iCheck can induce communication failures in applications, as seen in the above section. iCheck takes necessary steps to counter such component failures to provide a fair checkpointing service to the connected applications. The steps taken to thwart various component failures are discussed below.

- **Agent Failure:** The manager detects an agent failure if no activities are happening inside the agent. Agents should periodically inform the manager about its status. Since the application will use the PFS technique if agent failure is detected, managers only need to communicate the agent failure to the controller. During the probe call, the controller can launch a new set of agents for the application.

- **Manager Failure:** Managers periodically send heartbeat messages to the controller to inform about the availability. The controller detects a missing heartbeat message from an active iCheck node (Node where the agents are loaded), the controller will mark the application for agent change, and new agents are given for the application during the next probe call. The controller tries to relaunch the manager, and if it fails, it indicates that the node is down.

- **Controller Failure:** In the event of the controller failure, the secondary controller will take control. The secondary controller is ideally placed in a different node and is functional to receive messages from the primary controller. Once the message from the primary controller fails, the secondary controller takes control and loads the application and agent information from the file system to its memory. The primary controller's resource management module updates application and agent metadata periodically to the file system.

To summarize, whenever the controller detects a component failure, the controller allocates new agents for the applications affected in a new node. The library will call the probe internally after a preconfigured (t_{wait}) time is elapsed after every communication issue and will be transparent to the application. So the application does not get impacted by the component failures in iCheck. If the time taken for a probe call is t_{probe} and to connect to new agents is t_{new_agent}, the overhead $t_{overhead}$ induced in the application with checkpoint frequency $cp\_frequency$, when a component failure occurs and the library can reconnect successfully after n tries can be estimated as:

$$t_{overhead} = t_{probe} + t_{new\_agent} + \frac{n \times t_{wait} \times (t_{pfs\_technique} - t_{rdma\_transfer})}{cp\_frequency} \tag{6.13}$$

where $n \times t_{wait}$ is the time elapsed before reconnecting, $\frac{n \times t_{wait}}{cp\_frequency}$ is the number of checkpoint operations performed directly into the PFS while waiting for iCheck to reconnect, $t_{pfs\_technique}$ is the time taken for the fallback checkpoint mechanism in the library, and $t_{rdma\_transfer}$ is the time taken for checkpoint transfer using RDMA.

Overhead induced if the application cannot reconnect till the end of the application can be estimated as:

$$t_{overhead} = t_{probe} + t_{new\_agent} + \frac{t_{elapsed} \times (t_{pfs\_technique} - t_{rdma\_transfer})}{cp\_frequency} \tag{6.14}$$

where $t_{elapsed}$ is the time taken to finish application execution from the point of initial communication error.

6.5.3 Node Failure

From the perspective of iCheck, both software and hardware failures will have the same impact. For example, when a controller encounters a manager unreachable scenario or does not receive heartbeat messages, the controller will follow the same procedure irrespective of the issue that caused it. The manager might be down because the entire iCheck node was down or because there might be some undetected software bugs that crashed the manager. In both cases, the controller will try to reconnect and launches a new manager. The controller will take steps defined in the above section to mitigate the issue associated with node failures.

To summarise, from an application perspective, whichever failure occurs under the hood (either a communication failure or node failure) is handled by the iCheck library and the checkpointing operation works. The whole internal failure recovery activities remain transparent to iCheck. However, the application performance will be affected since the fallback mechanism is to write into the parallel file system (This can be seen in Chapter 9. Nevertheless, iCheck ensures that the application is always resilient to failures.

7

Resource Management in iCheck

This chapter explores the resource management aspect of iCheck and discusses it in four sections. . The first section describes the agent management aspect of iCheck in detail. The second section outlines how iCheck leverages the agents to provide checkpointing services to malleable applications. The third section discusses how iCheck utilises the invasive resource manager to achieve horizontal scaling capability. Finally, the last section concludes the chapter with a discussion on how iCheck incorporates data distribution capabilities into the checkpointing system.

Chapter 6 briefly described the design of the iCheck system and introduced the core components of iCheck and its interaction. The following chapter provides deep insight into the most dynamic component of the iCheck – namely, the iCheck agent, how it is selected and where it is placed. Further, this chapter explores the **invasive** aspect of iCheck (**invasive** checkpointing system) and gives an overview of how the iCheck is

- providing checkpoint support for malleable applications developed using the invasive infrastructure.
- interacting with `iRM` to expand and shrink its resources.
- providing a data distribution library for malleable resource management.

7.1 Agent Management in iCheck

iCheck's performance is largely contingent upon its configuration setup. The interaction between agents and application processes for checkpoint transfer represents the core and most frequent task within iCheck. Hence, strategic placement of agents across iCheck nodes is essential for maximising application performance, offering significant opportunities for optimisation. To optimise and adjust iCheck effectively, three main parameters can be manipulated: the number of agents, the selection of iCheck nodes for agent deployment, and the arrangement of agents within these iCheck nodes. To fine-tune these parameters, the iCheck controller needs to have an overview of the current system status (For example, the available memory in an iCheck node and the total number of agents). The controller gets this information from managers in iCheck nodes.

As seen in Section 6.1.3, the `icheck_init()` will register an application with the controller and obtains the agent information. This agent information contains the number of agents and the iCheck nodes in which

the agents are launched. The application will later use this information to connect to the agents. From the controller side, deciding the number of agents and where to place the agents is not trivial since this decision can impact the application's performance. The following two subsections cover these decision-making mechanisms inside the controller triggered during the application's `icheck_init()` call.

7.1.1 Agent Count Selection

An application requires agent information from the controller to leverage iCheck's RDMA-based checkpointing capability, and the controller gives this information to the application in the following three instances.

1. During the application registration : `icheck_init()`
2. During the application restart : `icheck_restart()`
3. During the agent change query performed by the application : `icheck_probe_agents()`

The agent count selection is trivial for the controller in the first two instances. The controller determines the number of agents based on the agent selection algorithm shown in Listing 5. As described in the algorithm, the controller first obtains details such as the application ID (*app ID*), the number of nodes in the application, the characteristics of the application, and the RDMA strategy used (see subsection 6.2.1). When an application connects, the controller will check whether the *app ID* already exists. The absence of an *app ID* indicates that the app connects to the controller for the first time, and the controller has to find the initial agent count. The initial agent value was set as the number of nodes in an application based on the empirical results from the experiments (see Line 16 in algorithm 5). Values for initial agent *init_agent_count* can be provided in the configuration file provided by iCheck (See Listing 6.2 in Subsection 6.3.6).

In the event of an application restart, the app id will be present in the controller. Hence, the controller can immediately return the number of agents based on the previously stored value (see Line 18 of algorithm 5). Since iCheck also allows that agents change dynamically with `icheck_probe_adapt`, it is essential to differentiate between a restarting application and an agent change request. *is_active* flag can be used to differentiate between the two (see lines 21 - 23), and a restarting application will not have this flag set.

Selecting agent count is non-trivial in case of an agent change. The controller requires additional information about the current checkpointing process of the application to determine the agent count. The controller must know whether the strategy used is *mr* or *shmr* since the maximum number of agents allotted for an application can be *num_nodes* for the strategy *shmr* and *num_process* for the strategy *mr*. These limits should bind the new agent number. Furthermore, the checkpoint transfer time should be compared for an application if it has already undergone a previous agent change (Lines 3 - 9). For example, if an increase in agents has increased the checkpoint time, it is crucial that during the next agent change, the number of agents should be decreased but not increased (Line 4). In case of the first agent change, the controller will increase the agent by a $factor$ which can be configured (Line 6). The initial value of $factor$ is set as 2 and can be configured dynamically (See Listing 6.2 in Subsection 6.3.6). The controller will increase or decrease the number of agents during further changes based on the checkpoint performance variation. As per the iCheck design, an application is guaranteed to have at least one agent allocated to it (Line 4). Moreover, the maximum number of agents that can be assigned to an application is bounded by the total number of processes of the application (Line 6).

Algorithm 5. The agent count selection algorithm

```
1  Function get_num_agents:
2      Read value of factor
3      if t_sync > old_t_sync then
4          num_agents = max(num_agents/factor, 1 )
5      if t_sync < old_t_sync then
6          num_agents = min(num_agents * factor, #num_processes )
7          if shmr == true and num_agents > num_nodes then
8              Set num_agents = num_nodes
9      return num_agents
10 return
11 Function agent_selection:
12     Obtain app specific information app_id, number of nodes num_nodes, number of processes
         num_processes, application type is_active
13     Get the strategy mr or shmr
14     Get the total agents in managers total_agents
15     if app_id not exists then
16         Set initial number of agents for application num_agents as num_nodes
17     if app_id exists and is_active == false then
18         Retrieve previous old_num_agents
19         num_agents = old_num_agents
20     Obtain the current t_sync and previous old_t_sync values
21     if app_id exists and is_active == true then
22         Retrieve current num_agents
23         Get new num_agents by calling get_num_agents()
24     Update total_agents
25     Update the previous old_t_sync value with the current t_sync value
26     return num_agents
27 return
```

7.1.2 Agent Placement

Once the number of agents has been determined, the controller's next responsibility is to identify appropriate iCheck nodes where these agents can be deployed. The empirical analysis showed that agent placement significantly impacts the application's performance. Various metrics collected from the monitoring infrastructure will be used for making the agent placement decision. As seen in Section 6.3, the metrics collected were related to categories of bandwidth, memory, checkpoint operations, and power. The agent placement algorithm facilitates different strategies to pick the ideal iCheck nodes to place an agent.

The agent placement can be divided into three independent tasks and is detailed in below subsections. The first and foremost task is determining how to choose iCheck nodes (*candidate_list*) from the available node list (*candidate_pool*). The second task is deciding how to distribute the *agent_num* agents across the iCheck nodes. The final task is deciding the number of iCheck nodes (*candidate_num*) to place these agents. The default numbers for these tasks (*candidate_num* and *agent_num*) can be configured dynamically with iCheck configuration file (See Listing 6.2 in Subsection 6.3.6).

Algorithm 6. The agent placement algorithm

```
1  Function candidate_count:
2      read candidate_num from config file
3      return candidate_num
4  return
5  Function candidate_selection:
6      read selection_strategy from config file
7      Create candidate_list to select the iCheck nodes
8      if selection_strategy == MEMORY then
9          Select iCheck nodes with maximum available memory
10     else if selection_strategy == BANDWIDTH then
11         Select iCheck nodes with least network traffic
12     else if selection_strategy == FREQUENCY then
13         Select iCheck nodes with least checkpoint operations/s
14     else if selection_strategy == POWER then
15         Select iCheck nodes with low power usage
16     return candidate_list
17 return
18 Function agent_placement:
19     Obtain the application information
20     Get the agent count from agent_selection() function
21     Select the agent distribution strategy distribution_strategy
22     Choose the agent placement strategy selection_strategy()
23     Get the candidate_list from candidate_selection() function
24     read distribution_strategy from config file
25     Get the candidate count from candidate_count() function
26     if distribution_strategy == block then
27         Distribute agents in block across the candidate_list
28     else if distribution_strategy == roundrobin then
29         for each agent j in agent_list do
30             Select next iCheck node i in the candidate_list in the round-robin method
31             launch agent j in iCheck node i
32         end
33     else if distribution_strategy == filltocompletion then
34         for each iCheck node i in candidate_list do
35             Launch maximum agents in iCheck node i
36         end
37 return
```

7.1.2.1 Node Selection

iCheck provides four distinct strategies to determine the nodes for agent deployment (as seen in lines 5 - 16 in Algorithm 6) and they are based on:

1. Bandwidth: In this strategy, the controller will create a sorted list of iCheck nodes with their bandwidth availability. The metric bw_{agent} (defined in subsection 6.3) is used for the list creation. The controller then selects the iCheck nodes that transfer a low amount of data (or have higher bandwidth

available for checkpoint transfers) to place the agents. This strategy is beneficial for application that frequently checkpoints a huge size of data. As can be seen from the results (see section 9), it improves the application performance.

2. Available Memory: This strategy focuses on the available memory on an iCheck node. The controller creates a sorted list of iCheck nodes based on memory availability using the metric *avail_mem*. The controller can then choose the iCheck nodes with the maximum available memory to place the agents. This strategy becomes beneficial for applications that typically checkpoints a huge amount of data. This makes sure that the checkpointing process doesn't lead to out of memory issues in iCheck nodes. It can be seen in lines 8 - 9 of Algorithm 6.
3. Checkpoint Frequency: In this strategy, the controller will assemble a sorted list of iCheck nodes based on their checkpointing frequency. The metric $\Delta cops$ (defined in subsection 6.3) is used for the list creation. Then the controller selects nodes with the least activity. This strategy can benefit applications that checkpoints frequently.
4. Power Budget: This strategy focuses on the power consumption on an iCheck node. The controller constructs a sorted list of iCheck nodes with power usage using the metric *ic_power* and picks the iCheck nodes with the lowest power consumption for deploying agents. This approach is particularly useful when iCheck is required to operate within a defined power budget..

7.1.2.2 Distribution Scheme Selection

After identifying the appropriate iCheck nodes for agent deployment, the controller must decide how to distribute the agents across these nodes. iCheck offers following three different strategies for this purpose, detailed in lines 26 - 35 of algorithm 6:

1. Block Distribution: Agents are allocated in blocks across the iCheck nodes. If the block size is not specified, the distribution is even across all *candidates*. The block size can be adjusted using the iCheck dynamic configuration file, as illustrated in Listing 6.2 in Subsection 6.3.6.
2. Round Robin: This method distributes agents across the *candidates* in a round-robin fashion. The controller cycles through the selected iCheck nodes, assigning one agent at a time to each node until all agents have been allocated.
3. Fill to completion: Here, agents are assigned to nodes until reaching the node's maximum capacity. The goal is to cluster the agents as closely as possible within the nodes, ensuring a dense and efficient distribution.

7.1.2.3 Count Selection

Candidate count can be configured externally in iCheck. Selecting the candidate count based on the applied selection strategy is recommended. For example, the bandwidth-based selection strategy will not improve performance when the agents are distributed across all iCheck nodes. The agents should be distributed only across the limited number of candidates *candidate_list*.

The complete agent placement algorithm can be seen in Algorithm 6. Initially, the controller obtains the application information and gets the agent count from the $agent\_selection()$ function (as seen in Algorithm 5). The controller proceeds to pick a candidate selection strategy (Line 22 in Algorithm 6). Using the provided strategy, it creates a sorted candidate list by calling the $selection\_strategy()$ function (Line 23).

selection_strategy() will obtain the strategy from the iCheck configuration file (See subsection 6.3.6)) . Once the ideal candidates are obtained, the controller performs the necessary steps to distribute the agents. As a first step, the controller reads the distribution strategy *distribution_strategy*() provided in iCheck (Line 24). Similar to *selection_strategy*(), *distribution_strategy*() information is also obtained from the dynamic configuration file of iCheck. Then the controller decides the number of iCheck nodes by calling the *candidate_count*() function. The controller then gives agents unique ids and maps these agents to different iCheck nodes. As a next step, the controller launches these agents in different candidates chosen based on the distribution strategy. Finally, the controller stores information about the application (For example, agent count and candidate hostname) for bookkeeping.

The effect of various strategies on checkpoint performance is observable in Section 9. Different values can be provided for the number of iCheck nodes, how to pick the nodes, and how to place the agents across these nodes. They are dynamically configurable using a special configuration file (See Subsection 6.3.6). The number of agents can also be configurable via the same file. iCheck controller has dedicated threads to read and record the configuration parameters periodically.

7.2 Malleable Application and Agents

Malleable applications, as defined in Section 2.2.2, change their resources based on the resource manager's instruction. Malleable applications running without resource change are equivalent to rigid or non-malleable applications. Therefore, no additional support is needed from the checkpoint systems in such a scenario. The issue arises when the application changes its resources. Typically the checkpointing systems are mapped to the MPI world communicator (`MPI_COMM_WORLD`) and cannot change their mapping (size/process count) on the fly to adapt to the resource change. However, iCheck is designed to support resource changes in malleable applications out of the box.

The core idea of iCheck is agents retrieving checkpoints from applications (as seen in Chapter 6), and they have an $N:M$ mapping where N can be the number of processes or nodes (See the subsection 6.2.1) of the application and M is the number of agents. For a non-malleable application, the agent-to-resources mapping will always be $N:M$. For a malleable application with K resources changed during the application execution, the total number of resources N, will be changed to $N \pm K$ resources. Hence the new application to agents mapping can be $N \pm K : M$ or $N \pm K : M \pm L$, where L is the change in agents proposed by the controller. The strategy enabling the mapping $N \pm K : M$ is called Malleable Application with Rigid Agents (MA-RA) strategy and the mapping strategy $N \pm K : M \pm L$ is called Malleable Application with Malleable Agents (MA-MA) strategy.

In the below section, the resource granularity of *nodes* is used to explain the agent to application mapping for a malleable application. Agents are mapped based on the number of nodes for the *SHMR* strategy (see section 6.2.2), and for the *MR* strategy (see section 6.2.1), the agents are mapped based on the number of processes in the application. For the explanations in the below sections, a resource granularity of nodes are used for the sake of simplicity. Replacing the term nodes with processes also holds true for the below discussions.

7.2.1 MA-RA Strategy

In this strategy, iCheck has a rigid approach where $N \pm K$ nodes are mapped with the same number of agents as before (M agents). The existing agents are reconfigured to connect to the new node configuration of the

application. $N \pm K$ nodes are rearranged so that each agent is responsible for the checkpoint transfer of $(N \pm K)/M$ nodes. The precondition for this to work is that M should always be less than or equal to N and $N \pm K$. In other words, division by M should always give a number equal to or greater than one. This guarantees that an agent will be in charge of one or more nodes for checkpointing. Otherwise, in scenarios ($M > N \pm K$) where the agents count is greater than the number of nodes, $M - (N \pm K)$ agents will remain idle. Such a scenario is considered illegal in iCheck, and the controller always guarantees an initial mapping where N is fully divisible by M. As a result, during the initialization of iCheck for N application with M agents, the following two combinations of application-to-agent mapping are possible. The first is $M = N$, and the second is $M < N$. In both combinations, resource expansion and reductions have different effects on application and checkpointing.

7.2.1.1 Resource Expansion

Consider the scenario of resource expansion in a malleable application where K new nodes are added dynamically. In the case of $N = M$, the initial mapping $1:1$ is changed during the addition of the new nodes. The new mapping becomes $N+K:M$. Hence the ratio $1:1$ is changed to $N+K/M:1$ or $(1+K/M):1$. As a result, every agent needs to checkpoint an additional K/M number of nodes. If K is not divisible by M, then the $(N+K)/M$ nodes are assigned per agent and remaining $(N+K)\%M$ nodes are distributed among M agents in a round-robin fashion starting with agent ID 1.

For the second case of $N > M$, the initial ratio for application to agents will be $X:1$, where X is N/M. During an expansion, the M agents should be redistributed across $N+K$ nodes. So the new mapping will be $N+K:M$, and the ratio will be $(N+K)/M:1$. If $N+K$ is divisible by M, then each agent gets $(N+K)/M$ nodes to the checkpoint; otherwise, the agents get additional $(N+K)\,\%\,M$ nodes in round-robin manner similar to the first scenario.

There is an apparent performance penalty in such cases, which can be categorized into two types. In the first case, the application does not change its overall checkpoint size ($CPMEM$), and in the second case, the checkpoint size will also change ($CPMEM_{new}$). The performance penalty is less in the first case because M agents read the checkpoint of size $CPMEM$ in the case of N nodes and $N+K$ nodes. The overall bandwidth for checkpoint transfer remains the same ($CPMEM/M$). Hence, the performance penalty will not be due to the transfer rate since the same number of agents transfer the same amount of data before and after the resource expansion. As seen from the above checkpoint of $(N+K)\%M$ additional nodes should be retrieved by the agents. When a commit call happens, agents must use $(N+K)/M$ extra RDMA operations to transfer the checkpoint instead of N/M. The primary overhead will be associated with the iCheck agent configuration and is a one-time overhead. For the second strategy, the change in checkpoint size to $CPMEM_{new}$ will affect the performance since the iCheck agents need to transfer the checkpoint of size $CPMEM_{new}/M$ instead of $CPMEM/M$ where $CPMEM_{new} > CPMEM$ thereby potentially increasing the checkpoint transfer time. However, there will be a performance improvement in the case of $CPMEM_{new} < CPMEM$ since agents only need to transfer less data.

7.2.1.2 Resource Reduction

Consider the scenario of resource reduction in a malleable application where K nodes are removed dynamically. In the case of $N = M$, the initial mapping $1:1$ is changed during the removal of the new nodes. The new mapping becomes $N-K:M$. Hence the ratio $1:1$ is changed to $(N-K/M):1$ or $(1-K/M):1$. As a result, every agent needs to checkpoint K/M less number of nodes. Suppose K is not divisible by M. In that

case, the $(N-K)/M$ nodes are assigned per agent to be checkpointed in addition to $(N-K)\%M$ nodes are distributed among M agents in a round-robin fashion starting with agent 1.

For the second case of $N > M$, the initial ratio for application to agents will be $X:1$, where X is N/M. The M agents should be redistributed across $N-K$ nodes during a reduction. So the new mapping will be $N-K:M$, and the ratio will be $(N-K)/M:1$. If $N-K$ is divisible by M, then each agent gets $(N-K)/M$ nodes to the checkpoint; otherwise, the agents get additional $(N-K)\,\%\,M$ nodes in a round-robin manner similar to the first scenario.

There is no noticeable performance penalty in such cases, which can be categorized into two types based on the checkpoint sizes. In the first case, the application does not change its overall checkpoint size ($CPMEM$), and in the second case, the checkpoint size will also change ($CPMEM_{new}$). The performance might be improved in the first case because M agents read the checkpoint of size $CPMEM$ in the case of N nodes and $N-K$ nodes. The overall bandwidth for checkpoint transfer remains the same ($CPMEM/M$), and performance improvement comes from fewer RDMA operations. When a commit call happens, agents must use $(N-K)/M$ less RDMA operations to transfer the checkpoint instead of N/M. The potential overhead will only be associated with the iCheck agent configuration and is a one-time overhead. In the second case where $CPMEM_{new} > CPMEM$, the change in checkpoint size to $CPMEM_{new}$ will affect the performance since the iCheck agents need to transfer the checkpoint of size $CPMEM_{new}/M$ instead of $CPMEM/M$. However, there will be a performance improvement in the case of $CPMEM_{new} < CPMEM$ since agents need to transfer only less data.

7.2.2 MA-MA Strategy

In this strategy, the agents are malleable. For a malleable application with N nodes and M agents with a resource change of K nodes, the initial agent to application mapping $N:M$ will change to $N \pm K : M \pm L$ during the resource change, where L is the change in the agent count. This change in agents ensures that the aggregate bandwidth for checkpoint transfer remains the same (unlike in the MA-RA strategy). The initial agent to the application mapping ratio can be $1:1$ where $N = M$ and $(N/M):1$ where N is completely divisible by M. These two initial mappings can impact both resource expansion and reduction differently.

7.2.2.1 Resource Expansion

In the event of a resource expansion with K nodes for an application with N nodes and initial agents M, the new application to agent mapping will be $N+K : M+L$. In the case of $N = M$ agent distribution (the mapping ratio is $1:1$), the new mapping will be $N+K : M+K$ such that the ratio remains $1:1$. Here iCheck adds the same number of agents as the nodes K, thereby making L equals K for the application agent mapping.

In the case of $N > M$, the L needs to be calculated by the controller. iCheck ensures that N is fully divisible by M during the initialisation. As a result, the initial mapping for the ratio will be $(N/M):1$. The change in the node number to $N+K$ can cause similar scenarios explained in the above section. If $N+K$ is fully divisible by M, then the new number of agents will be $(N+K)/(N/M)$ (or $M+(MK/N)$) and the new mapping will be $(N+K):(M+(MK/N))$ which has the ratio of $(N+K)/(M+(MK/N)):1$ and can be simplified as $(N/M):1$. Since the new number of nodes K decided by the resource manager will give a new node configuration $N+K$, which might not be fully divisible by M. In such cases, the extra $(N+K)\%(N/M)$ nodes will be given to the agents in a round-robin manner, or iCheck can add extra L agents to M (where $L = N+K-M$) that makes the new ratio $1:1$ from $(N/M):1$.

As discussed in the above section, the checkpoint size can remain the same $CPMEM$ or change to the new size $CPMEM_{new}$. In the case of $1:1$ mapping with $CPMEM$ remaining the same for $N+K$ nodes, there will be a performance improvement since there are $M+K$ agents to transfer $CPMEM$ of data. Similarly, in the case of $(N/M):1$ mapping, $M+(MK/N)$ agents will transfer the checkpoint instead of M agents, thereby improving the checkpoint performance. If the checkpoint size increases $CPMEM_{new}$ for $N+K$ nodes with $1:1$ mapping, with the addition of $M+K$ agents, the aggregate checkpoint bandwidth previously available for the nodes will be available for the current distribution. In effect, the agents will provide a similar bandwidth as in the case of N nodes. The performance will improve compared to the MA-RA strategy with $1:1$ mapping. In addition, there will also be performance improvement for smaller $CPMEM_{new}$. In the case of $N/M:1$ mapping, iCheck ensures that node to agent ratio remains the same and gets a similar bandwidth as before with N nodes for the size $CPMEM_{new}$.

7.2.2.2 Resource Reduction

In the event of a resource reduction with K nodes for an application with N nodes and initial agents M, the new application to agent mapping will be $N-K:M-L$. In the case of $N=M$ agent distribution (the mapping ratio is $1:1$), the new mapping will be $N-K:M-K$ such that the ratio remains $1:1$. Here iCheck removes the same number of agents as the nodes K, thereby making L equals K for the application agent mapping.

In the case of $N>M$, the L needs to be calculated by the controller. iCheck ensures that N is fully divisible by M during the initialisation. As a result, the initial mapping for the ratio will be $(N/M):1$. The change in the node number to $N-K$ can cause similar scenarios explained in the above section. If $N-K$ is fully divisible by M, then the new number of agents will be $(N-K)/(N/M)$ (or $M-(MK/N)$) and the new mapping will be $(N-K):(M-(MK/N))$ which has the ratio of $(N-K)/(M-(MK/N)):1$ and can be simplified as $(N/M):1$. Since the number of nodes K to remove decided by the resource manager will give a new node configuration $N-K$, which might be partially divisible by M. In such cases, the extra $(N-K)\%(N/M)$ nodes will be given to the agents in a round-robin manner, or iCheck can remove extra agents and makes the new ratio $1:1$ from $(N/M):1$.

The performance will remain the same if the checkpoint size $CPMEM$ remains the same for the new node configuration. This can be attributed to the strategy of adjusting the agent number to maintain the previous application-to-agents ratio. However, the increase in checkpoint size to $CPMEM_{new}$ can lead to a drop in checkpoint performance since the agent number is also reduced with the number of nodes. Nevertheless, the performance will still be better than the MA-RA strategy. For a smaller $CPMEM_{new}$ size, the checkpointing performance of the application will be improved.

7.2.3 Agent Count Selection Algorithm

For the MA-MA strategy described above, iCheck must calculate a new agent count $M \pm L$ for every resource change in a malleable application. It is calculated using the agent count selection algorithm proposed in Listing 7. The agent count selection algorithm works as follows. When iCheck calls `MPI_Probe_adapt()` after a resource change, the information, like the new number of nodes and the current checkpoint size, is passed to the controller. The controller analyses the strategy employed by iCheck. If it is an *MR* strategy, the controller will map application processes to agents, while for an *SHMR* strategy, the controller will use application nodes for agent mapping. For the agent count selection, the controller sets the initial value of current resources as processes or nodes based on the strategy used.

Algorithm 7. The agent count selection algorithm

```
1  Function get_agent_count:
2      read the agent change strategy
3      if strategy is MA − RA then
4          return current_num_agents
5      if strategy is MA − MA then
6          if current_num_agents == current_num_resources then
7              set num_agents = new_num_resources
8          else
9              set num_agents = (current_num_agents/current_num_resources) ×
                 new_num_resources
10         end
11         return num_agents
12 return
13 Function dynamic_agent_selection:
14     Obtain app specific information app_id, number of nodes num_nodes, number of processes
         num_process, application type is_active application characteristic is_malleable; Get the
         strategy mr or shmr
15     if strategy is mr then
16         set current_num_resources = old_num_processes
17         set new_num_resources = num_processes
18     else
19         set current_num_resources = old_num_nodes
20         set new_num_resources = num_nodes
21     end
22     Get the total agents in managers total_agents
23     if app_id exists and is_active == false and is_malleable_app == true then
24         set current_num_agents = num_agents
25         call get_agent_count() to calculate new num_agents
26     Update total_agents
27     return num_agents
28 return
```

After that, the controller will call the *get_agent_count*() function with new resource information to get the agent count. In the *get_agent_count*() function, the controller will return the current number of agents for the MA-RA strategy (Lines 3 - 4). If the strategy is MA-MA, the controller will compare the previous agent count and the resources (process count for *MR* strategy and node count for *SHMR* strategy) count (Lines 6 - 10). The controller will set the new agent count as the new node count if the previous agent and node counts are the same to ensure a 1 : 1 ratio. Suppose the agent count and the previous number of nodes are not equal. In that case, the controller calculates the new agent count that satisfies the application's current node-to-agent ratio N/M : 1 (Line 9).

7.2.4 Pseudocode for Malleable Applications

The pseudocode of a simple iCheck-supported malleable MPI application is available in Listing 7.1. The malleability is provided by the iMPI library explained in Section 2.2.2.1. The pseudocode described in Listing 2.2 at Section 2.2.2.2 is extended with the iCheck malleability API call to demonstrate the functioning of iCheck in the context of a malleable MPI (`iMPI`). The control flow of an `iMPI` application is described in detail in Section 2.5.

As explained in Section 2.2.2.1, `MPI_Init_adapt()` (line 4 of Listing 7.1) will notify the malleability capability of the application with the invasive resource manager and returns the type of the `iMPI` process. A process can be initial (started during the application launch) or joining (launched during the expansion operation). Initial processes continue the application execution (skipping lines 11-19) and regularly call `MPI_Probe_adapt()` (line 22) to check for any resource change. However, joining processes immediately triggers the collective `MPI_Comm_adapt_begin()` (line 13) function and waits for initial processes (preexisting processes) to join. Meanwhile, the `MPI_Probe_adapt()` (in line 22) call informs initial processes about the resource change triggered by the malleable resource manager. Initial processes then call the collective `MPI_Comm_adapt_begin()` function (line 25).Once all the application processes (initial and joining) completes `MPI_Comm_adapt_begin()` routine (in lines 13 and 25), they can begin redistribution of data, share application-specific data and additional control information among them. After that, they execute `MPI_Comm_adapt_commit()` (in lines 15 and 27) method to finalise the resource adaptation and together continue the execution of the application. Regarding the checkpointing, `icheck_probe_agents()` (Line 17 and 29) is called immediately after the resource change by all processes. This probe call triggers the agent count selection algorithm (See Algorithm 7) in the controller. The controller will return new agent information to the application. Three possible outcomes, as mentioned in Section 6.4.2, can ensue here.

- iCheck gets a new agent number. The new agents list may reuse the existing agents or be completely new agents in different iCheck nodes.
- iCheck gets no change in agents. The agent number and mapping remain the same.
- iCheck gets agent change. The agent number remains the same, but the agent mapping (iCheck nodes) will be new.

The programmer is responsible for not calling the `icheck_restart()` for the newly joined processes (Lines 8 -10). As seen in line 8, the application developer can use the `type` obtained from the `MPI_Init_adapt()` calls to differentiate between new and joining processes.

```
#include<icheck.h>
int main() {
 /* Initialisation part of the application */
 MPI_Init_adapt(..., type) /* iMPI is intialised */
 float data[SIZE];
 icheck_init(..., type);
 icheck_add("data",...);
 if(checkpoint_available && type != joining){
  icheck_restart();
 }
 if (type == joining) {
   /* Start of adaptation window in joining processes */
   MPI_Comm_adapt_begin(...);
   /* Get metadata from preexisting processes, perform data redistribution */
   MPI_Comm_adapt_commit();
   /* Informs the iCheck controller about the resource change */
   icheck_probe_agents();
 }
 /* iterate over compute intensive phase of the application */
 while (true){
  /* iMPI processes must call probe periodically */
  MPI_Probe_adapt(resource_change,...);
  if (resource_change) {
    /* Start of adaptation window in preexisting processes */
    MPI_Comm_adapt_begin(...);
    /* Pass metadata to joining processes, Perform Data distribution. */
    MPI_Comm_adapt_commit();
    /* Informs the iCheck controller about the resource change */
    icheck_probe_agents();
  }
  /* Compute part of the application */
  /* Read/Write data[]*/
  if(iteration%100) {
    /* Checkpoint transfer happens here with new agent configuration */
    icheck_commit();
  }
  /* Check for agent change*/
  if(iteration%1000) {
    /* Passes new timing information to controller */
    icheck_probe_agents(hints);
  }
  /* End of compute loop*/
}
  /* Finalisation part of the iCheck API */
  icheck_finalize(...);
  /* Finalisation part of the application */
  MPI_Finalize();
}

```

Listing 7.1: Psuedocode of a malleable application with iCheck [78].

7.3 iCheck and Invasive Resource Manager

In the previous subsection, the adaptive nature of iCheck concerning the application was discussed. The primary focus was on how the iCheck components provide checkpoint support to a malleable application. As seen in Section 6.4.1, iCheck also supports system-level dynamism through the ability to scale its resources horizontally, and this subsection explores how the system-level dynamism is performed in iCheck in detail. Nodes can be added and removed dynamically from the iCheck environment. It is possible due to the ability of the controller to support nodes on the fly. The real benefit of such an approach can only be used in HPC with a malleable resource manager. iCheck's ability to scale nodes is complemented by a resource manager that can provide the nodes based on the checkpointing system requirement.

The invasive resource manager (`iRM`) introduced in Section 2.2.1 has been used to provide nodes to iCheck. iCheck and `iRM` needed to be extensively modified to enable adaptive checkpointing. Towards that extent, the following changes were made:

- Extended the iCheck controller to support iCheck-aware `iRS`.
- Created a new iCheck aware resource management plugin in `iRM`.

The following subsections cover in detail how these changes support system-level dynamism in iCheck.

7.3.1 Messages and Policy in the Controller

The default iCheck controller is extended heavily to support the iCheck-aware `iRS`. The extensions were done on two levels. In the first level, a communication backend is established with `iRM` to send the messages. In the second level, the policies were developed in the controller that dictates the necessity for resource scaling.

7.3.1.1 Communication Extensions

The iCheck controller reads the port information of `iRM` from the configuration file and establishes a connection during the launch. The controller sends the following different messages to `iRM` during the course of its execution.

- *Status Message:* Status messages are sent to `iRM` periodically in (`HB_IRM_INTERVAL`) heartbeat message interval provided in the configuration file (See Listing 6.2). Ideally, this interval should be less than the scheduler tick interval such that `iRM` knows the status of the iCheck before making decisions regarding the running applications. The status message contains information about iCheck nodes (For example, iCheck node status and available memory). A scenario where a status message might be relevant is `iRM` taking back the unused iCheck node to schedule a new application from the waiting queue.
- *Query Message:* The controller uses a query message to know about the current idle nodes inside the resource manager. Based on the response from `iRM`, the iCheck controller can make policies regarding checkpointing. For example, if iCheck nodes have insufficient memory to support a new application, the controller can request a new node based on the query message's response.
- *Resource Request Message:* The controller can proactively request `iRM` for new iCheck nodes using this message. The urgency of the resource requirement can be specified in this message to `iRM`.

- *Response Message:* This message is sent by the controller as a response to a message from `iRM`. `iRM` can request a resource change, and the controller provides the status with this response message. For example, `iRM` can retrieve an iCheck node from the controller. The controller gets adequate time to perform the agent migration and other bookkeeping activities. After that, the controller can send the response message to `iRM` that the resources can be taken back.

The controller receives the following different messages from `iRM` during its execution.

- *Query Message:* `iRM` sends this message to get information from the iCheck controller. There are different types of query messages `iRM` can send to the controller. The status message with `info` parameter can be used to elicit a response from the controller with the latest iCheck system metrics. A status message with `resource_change` parameter is used to get a response from the controller regarding the state of the previously triggered resource change.
- *Resource Change Message:* This message informs the controller about the resource change. `iRM` can send this message as a response for the *Resource Request Message* message from the controller. Additionally, `iRM` will initiate this message if it needs to take back the resources from iCheck proactively. In the event of a resource addition, this message contains information about the iCheck node. In case of a resource reduction, it contains the number of nodes iCheck should return to `iRM`.
- *Response Message:* This message is sent by `iRM` as a response to the *Query Message* from the controller. In this message, `iRM` will provide the count of current idle nodes in the system.
- *App info Message:* This message is sent by `iRM` proactively to inform the controller about the potential resource change in an application. Based on this information, the controller can pre perform operations to improve the overall checkpoint performance. For example, iCheck can prepare new agent distribution.

The communication backend is developed in the controller to support these various messages. The iCheck controller gets the information about `iRM` communication ports from the iCheck configuration file (As seen in Section 6.3.6). `iRM` will get the iCheck controller information from the initial status message.

7.3.1.2 Policy for Resource Extensions

The controller's actions can be categorized as proactive or reactive based on the current status of the iCheck system. Proactive actions are taken to request more resources if there is not sufficient memory in iCheck. The reactive actions are taken based on the instructions from `iRM`.

- *Proactive:* As described in Section 6.3, the controller has monitoring infrastructure that has a global view of the system. When the controller sees that an iCheck node is filling fast with checkpoints, the controller can request additional iCheck nodes. The controller sends a *Resource Request Message* to `iRM`. The controller can also set the urgency level in the message to `HIGH` to denote that the extra iCheck node is critical for the working of the iCheck system. The controller can also send a *Resource Request Message* in non-critical scenarios to ensure that the iCheck agents can provide better checkpoint services with more resources (For example, redistributing agents across multiple iCheck nodes to improve the overall bandwidth of the currently running applications). In such cases, the controller will initially send a *Query Message*. If the response contains idle node information, it will send a *Resource Request Message* message with the urgency level `LOW` to get new iCheck nodes.
- *Reactive:* Reactive approach is activated due to the *Resource Change Message* from `iRM`. `iRM` has the highest priority, and the controller is obliged to implement the instructions from the resource manager.

Algorithm 8. The controller horizontal scaling proactive algorithm

```
1  Function icheck_scaler_proactive:
2      for avail_memory,total_memory in each icheck_node do
3          get node id in node_id
4          if total_memory − avail_memory > LOWER_THRESHOLD then
5              add node_id to flagged node flagged_node list
6          else
7              combined_avail_memory = combined_avail_memory + avail_memory
8          end
9      end
10     for avail_memory in each flagged_node do
11         if total_memory − avail_memory > UPPER_THRESHOLD and MemoryDepletionRate >
             threshold then
12             if combined_avail_memory < total_memory then
13                 send ResourceRequestMessage with HIGHPriority
14             else
15                 send ResourceRequestMessage with LOWPriority
16             end
17             return
18         combined_avail_memory = combined_avail_memory − total_memory
19     end
20     clear flagged_node
21     for bandwidth, cp_frequency in each icheck_node do
22         get node id in node_id
23         if bandwidth < BW_THRESHOLD and cp_frequency > CP_THRESHOLD then
24             add node_id to flagged node flagged_node list
25     end
26     if pending_query_request == true and flagged_node contains nodes then
27         read the irm_idle_node_count flag from the controller
28         if irm_idle_node_count >= 1 then
29             update pending_query_request value to false
30             send ResourceRequestMessage with LOW Priority
31             return
32     if flagged_node contains nodes then
33         send a QueryMessage
34         update pending_query_request value to true
35         return
36 return
```

If `iRM` sends the *Resource Change Message* in response to the *Resource Request Message* in the controller, the controller can immediately claim the resources by launching the manager in the provided node and sending back a *Response Message*. In the event of a *Resource Change Message* triggered by `iRM`, the controller frees up the agent from the iCheck nodes and sends back the information of iCheck nodes that `iRM` can reclaim. *Response Message* from the controller has the information of the released iCheck nodes.

Algorithm 9. The controller horizontal scaling reactive algorithm

```
1  Function node_removal:
2      for icheck_node in each removal_candidates do
3          for application in each icheck_node do
4              get agent_info of application
5              get manager_info of application
6              notify agents in agent_info to write checkpoint into PFS
7              notify managers from manager_info to kill all agents in agent_info
8              recalculate new agent distribution for application
9              call agent_placement function for application
10         end
11         inform manager in icheck_node to destroy iCheck related data
12         instruct manager in icheck_node to self destruct
13         mark icheck_node as DOWN
14     end
15     return success
16 return
17 Function launch_manager:
18     for node in each new_candidates do
19         launch manager in node node
20         mark node as UP and available for checkpointing
21     end
22     return success
23 return
24 Function icheck_scaler_reactive:
25     read the message type msg_type and msg from iRM
26     if msg_type is NEW_NODES then
27         read the node_hostname, node_ip, node_properties
28         add the node information to the new_candidates data structure
29         call the launch_manager function with new_candidates information
30     else if msg_type is REMOVE_NODES then
31         read the number of nodes to remove node_remove_count from the message
32         Sort icheck_nodes with available memory in candidate_pool in increasing order
33         Sort candidates with similar available memory based on agent_count
34         Update candidate_pool
35         while counter < node_remove_count do
36             flag node_id of the icheck_node to removal_candidates
37             increment the counter counter
38         end
39         call the node_removal function with removal_candidates
40     else if msg_type is APP_INFO then
41         call the adapt_prepare_module
42     send ResponseMessage to the controller
43     return
44 return
```

The controller has algorithms that dictate the policy for proactive and reactive actions. The algorithm takes input from the monitoring infrastructure and considers the metrics defined in Section 6.3 for making management decisions. The key metrics the controller uses are available memory (*avail_mem*) and memory depletion rate (*MemoryDepletionRate*) for proactive actions, and the metrics available memory and agent counts (*agent_count*) are used for the reactive actions of the controller. Based on these actions resource scaling module in iCheck can be categorized as a proactive scaler or a reactive scaler. The algorithm for both scalers is shown in the Listing 8 and 9 and is explained below:

- *Proactive Scaler:* The proactive scaler algorithm is shown in Listing 8. The proactive scaler in iCheck is called periodically every *SCALER_INTERVAL* seconds by the controller. The scaler reads the performance metrics from the monitoring framework. In the first loop, it iterates through the metrics of every iCheck node and checks for memory usage characteristics (Line 2). The scaler flags all the iCheck nodes where the current memory usage is above a threshold (`LOWER_THRESHOLD`) (For example, 80%) of the total available memory (Lines 4-5). The scaler also calculates the overall available memory among the unflagged nodes (Line 7). Suppose the scaler sees sufficient memory available (*combined_avail_memory*) among all iCheck nodes to transfer the flagged nodes' checkpoint contents. In that case, the scaler will not send an urgent *Resource Request Message* with `HIGH` priority (Line 15).

  If any of the iCheck nodes have used above an `UPPER_THRESHOLD` (For example, 90%) of the memory and the *MemoryDepletionRate* rate is high (Line 11), and the combined available iCheck memory *combined_avail_memory* is not sufficient to maintain the checkpoint operations, it requests a *Resource Request Message* with `HIGH` priority (Lines 10 - 18). Scaler has a conservative estimation of *avail_memory* of a flagged node. Scaler uses $total\_memory$ instead of $total\_memory - avail\_memory$ of the flagged iCheck node to compare with *combined_avail_memory* from all of the iCheck nodes (Line 18). That means the scaler can send a `HIGH` priority message even though there might be sufficient memory *combined_avail_memory* among other nodes to store the checkpoints. The scaler calculates the total available memory and flags the nodes with the available memory reaching the specified threshold (Line 18). Scaler performs the comparison and requests an extra node if the `UPPER_THRESHOLD` is reached that cannot be satisfied by *combined_avail_memory* (Lines 12 - 13).

  The scaler can also ask for the *Resource Request Message* with `LOW` priority if the aggregate bandwidth available for an iCheck node is very low. For example, if the $cp\_frequency$ is very high for all the iCheck nodes, the scaler can request extra iCheck nodes. For the low-priority resource change, the scaler looks for the metrics *bandwidth* and $cp\_frequency$ (Lines 21 - 35). The scaler will flag iCheck nodes if the bandwidth exceeds the threshold `BW_THRESHOLD` and the checkpoint frequency exceeds the threshold `CP_THRESHOLD` (Lines 23 - 24). These thresholds are calculated based on empirical analysis and differ based on the system. Line 33 shows that the controller sends a *Query Message* and checks for the response in the next scaler pass (Line 26) to check the idle nodes. Since it is not a critical resource requirement, the scaler will wait till the next pass to issue the *Resource Request Message* for `LOW` priority messages (Lines 26 - 31). Meanwhile, the controller decodes the messages from `iRM`, and if the message is a reply to the *Query Message*, the controller then sets the flag *irm_idle_node_available* and adds the value in the message to the flag *irm_idle_node_count* (Line 27). The scaler uses this flag later, as seen in Algorithm 9.

- *Reactive Scaler:* The iCheck `iRM` communication module calls the reactive scaler after receiving a *Resource Change Message* message from `iRM`. The reactive scaler decodes the message and identifies the type of the message. If the *Resource Change Message* informs about the new nodes, the reactive scaler calls the launch function inside the controller to start the manager instance. In the case of resource removal, the scaler needs to perform the following set of operations. First, it must get the

number of iCheck nodes that must be returned to iRM. Secondly, the reactive scaler must flag the appropriate iCheck nodes for removal. After marking the iCheck nodes, the reactive scaler calls the node removal function in the controller before exiting. Lastly, when the nodes are removed, the controller informs iRM about the status of the resource change operation.

The reactive scaler algorithm is shown in Listing 9. The scaler has different control flows based on the type of message received. For the *NEW_NODES* in *Resource Change Message*, the scaler will read hostname, IP address, and node characteristics (number of cores, total memory) (Lines 26 - 27). Add the node information into a *new_candidates* data structure (Line 28). After getting all new node information, the *launch_manager* function is called (Line 29). In the *launch_manager* function (Lines 17 - 22), the controller launches managers in every node in *new_candidates* data structure, adds the manager information into the controller database, and updates the manager's status as *UP* and is available for agent placement (Line 20).

For the *REMOVE_NODES* message from iRM (Line 30), the control flow is complicated. Firstly, the reactive scaler will read the number of nodes *node_remove_count* that need to be removed (Line 31). Now the scaler must decide which iCheck nodes (*removal_candidates*) to be removed. iCheck creates a *removal_candidates* list based on the heuristic derived from empirical analysis. Firstly, iCheck creates a sorted list of currently available iCheck nodes based on the available memory (Line 32). The larger the available memory, the less the checkpointing action taking place in the node. Secondly, iCheck creates another sorted list of currently available iCheck nodes based on the agent count. The smaller the agents, the faster the agent removal process. The scaler will pick *node_remove_count* nodes from both lists and perform the intersection. The remaining candidates will be chosen from the list of available memory. This approach ensures that the iCheck removes the manager nodes with fewer agents and maximum available memory to avoid the overhead in agent redistribution and writing data into the parallel file system. These candidates are flagged for removal and are added to the *removal_candidates* list (Lines 35 - 37).

The *node_removal* function is called with *removal_candidates* list as input (Lines 1 - 15). The controller will iterate through each node in the list and obtain information about the applications using these nodes for checkpointing (Lines 3 - 9). For each application, the controller will peruse the agent information and inform the agent to write its most recently saved checkpoint into the parallel file system. Once the agents write the data into the parallel file system, the controller will ask the manager to kill the agents associated with the application (Line 7). The controller will then calculate new agent distribution for the application. Once all the applications in a node are informed about the agent change, the controller instructs the manager to remove iCheck-related data and kill the process (Lines 11 - 13). The controller then marks the manager as unavailable. After killing all of the manager processes, the scaler informs iRM that the nodes can be removed.

For the *APP_INFO* message from iRM, the controller will call the adaptation preparation module *adapt_prepare_module* (Lines 40 - 42). This module is responsible for calculating new agent distribution and data redistribution for an application that uses the data distribution service provided by iCheck.

Both scalers use the common API provided in the communication module to send messages to iRM. The following section explores the changes made in the Slurm scheduler plugin to support malleability in iCheck.

7.3.2 iCheck Aware Scheduler

An iCheck-aware job scheduling plugin in iRM was created to enable system-level malleability in iCheck. The new scheduling plugin supports the following interactions with iCheck:

- iRM can provide additional iCheck nodes as per the resource request from iCheck and availability of compute resources in iRM. For example, when iCheck runs out of memory in an iCheck node, the controller can request additional iCheck nodes (compute node) from iRM.
- iRM can reclaim nodes from iCheck. For example, to cater to the resource demands of a high priority job or to satisfy power requirements.
- iRM can communicate application-specific information to the controller. For example, it can notify the controller about an impending resource change of an application so that agents can make preparations for data distribution.

Towards this, a communication module and a scheduler plugin were added to iRM and are explained in detail below.

7.3.2.1 Communication Extensions

The communication module was added to send and receive control information from the iCheck controller. The communication module provides an API for iRM to send the *Query Message*, *Resource Change Message*, *Response Message*, and *App Info Message* to iCheck. The scheduler plugin sends these messages using the API *sendiCheckInfo(Message Type, Parameter 1, Parameter 2,...)*. The communication module provides a backend to receive and process the *Status Message*, *Query Message*, *Resource Request Message*, and *Response Message* from the controller. A dedicated communication thread waits to receive messages from iCheck. Once a message is received, the module will decode the message and, based on the message type, update the data in iCheck-specific data structures inside the scheduler plugin. For example, *Response Message* from the controller about releasing an iCheck node is added to the iCheck data structure. During the next scheduler pass, the scheduler will iterate through the stored iCheck data and utilize the received information to make resource management decisions. The different type of messages used by iRM and the iCheck controller is explained in detail in Section 7.3.1.1.

7.3.2.2 Scheduler Plugin

The iCheck aware scheduler plugin is an extension of ABS and iRS that aids iCheck and schedules malleable applications using two different scheduling strategies. In the first strategy, the priority is given to the iCheck system and is defined as *iCheck_Aware_Schedule* strategy. However, in the second scheduling strategy *iCheck_Aware_Schedule_Favor_App*, priority is given to running malleable applications.

Algorithm 10. The iCheck aware scheduling function with priority for iCheck system.

```
1  Function get_ic_node_info:
2      if Message from iCheck == ResourceRequestMessage and Priority == HIGH then
3          read ic_num_nodes requested
4          while ic_num_nodes > 0 and idle nodes are available do
5              get idle_node_name, idle_node_info
6              call function sendiCheckInfo(ResourceChangeMessage, idle_node_info, ...)
7              decrement ic_num_nodes and mark idle_node_name as busy
8          end
9          if ic_num_nodes > 0 then
10             set flag get_ic_node to HIGH
11         end
12     end
13     else if Message from iCheck == ResourceRequestMessage and Priority == LOW then
14         set flag get_ic_node to LOW
15 return
16 Function iCheck_Aware_Schedule:
17     Get workload and resource information
18     if no malleable job is adapting and number of running malleable jobs > 0 then
19         call the function get_ic_node_info
20         if strategy == favor_iCheck and get_ic_node == HIGH then
21             for each running malleable job j in the job queue do
22                 calculate the MTCT metric
23             end
24             Pick ic_num_nodes jobs with lowest MTCT metric and not in final phase end in
                candidates
25             for each running malleable job j in candidates and ic_num_nodes ! = 0 do
26                 Read the constraints of job j.
27                 Calculate new resource distribution as num_nodes − ic_num_nodes
28                 redistribute the job j.
29                 get the new idle_node_name, idle_node_info
30                 call function sendiCheckInfo(ResourceChangeMessage, idle_node_info, ...)
31                 mark idle_node_name as busy
32                 decrement ic_num_nodes
33             end
34         end
35         if get_ic_node == LOW and idle nodes available then
36             while ic_num_nodes > 0 and idle nodes are available do
37                 get idle_node_name, idle_node_info
38                 call function sendiCheckInfo(ResourceChangeMessage, idle_node_info, ...)
39                 mark idle_node_name as busy
40                 decrement ic_num_nodes
41             end
42         end
43     end
44 return
```

- *iCheck_Aware_Schedule*: In this strategy, the priority is given to the iCheck system where the *Resource Request Message* requests from the controller are handled immediately and catered to during every scheduler pass. The scheduler reconfigures the running application to free up resources to give nodes to iCheck. The potential candidate is selected based on the application performance, and resources are removed from the worst-performing application. The `MTCT` metric defined in Section 4.1 is utilized for analyzing application performance. The advantage of this strategy is based on the assumption that the time taken to write into PFS has a considerable impact on an application's performance which can be reduced using RDMA-based checkpointing by delivering additional nodes to iCheck when necessary.

  This strategy's algorithm can be seen in Listing 10 and works as follows. Initially, the scheduler will obtain workload information and analyze the current system state (Line 17). When malleable applications are running and no current adaptations are happening, the scheduler will check for any messages from iCheck in iCheck-related data structures (Line 19). Whenever a new message arrives, the communication module processes it and updates the information in iCheck-related data structures so the scheduler can utilize it. Since iCheck has higher priority in this strategy, the scheduler will immediately analyze the messages, and if there is a resource request from the controller and the request priority is `high`, the scheduler will read the number of nodes requested by iCheck (Line 3). The scheduler will check for available idle nodes immediately (Line 5). In case of availability, the scheduler will send this information to iCheck using the $sendiCheckInfo(ResourceChangeMessage, idle\_node\_info, ...)$ message (Line 6). The scheduler then marks the idle nodes as busy (Line 7). If enough idle nodes are absent, the scheduler will calculate the nodes needed (*ic_num_nodes*) and set the flag *get_ic_node* to *HIGH* (Line 10). For a resource request with `LOW` priority, the scheduler marks the nodes needed and sets the flag *get_ic_node* to *LOW* (Lines 13 -14).

  Then, the scheduler will iterate through the malleable applications and pick the candidates from which to take nodes to satisfy the iCheck's requirement of new *ic_num_nodes* nodes. These candidates are selected based on the `MTCT` metric (Line 22) and the time remaining in the job execution (Line 24). For example, if a worst-performing malleable job is ending, there is no benefit in taking resources away from that job. The runtime system takes the *ic_num_nodes* resources from malleable applications by reconfiguring the running applications (Lines 25 - 32). Then the newly obtained node information is passed to iCheck using the communication API (Line 30). The nodes are marked as busy by the scheduler. The scheduler does not rearrange the running malleable applications for `LOW` priority resource requests. Instead, if idle nodes are available at the scheduler pass's end, the requested nodes are provided to iCheck (Lines 35 - 40).

- *iCheck_Aware_Schedule_Favor_App*: In this strategy, the priority is given to the running malleable applications where the *Resource Request Message* requests from the controller are handled at the end of a scheduler pass and catered only if there are free resources. The scheduler reconfigures the running application and asks iCheck to free up resources for running applications or cater to the applications in the job queue. The potential candidate is picked by iCheck selected based on the available memory and agents, as discussed in Section 7.3.1.2. This strategy always prioritizes running malleable applications, and the only guarantee from `iRM` is that iCheck will always have at least one active node.

  This strategy can be seen in Algorithm 11 and works as follows. Initially, the scheduler will obtain workload information and analyze the current system state. When malleable applications are running and no current adaptations are happening, the scheduler will check for any messages from iCheck in iCheck-related data structures (Line 23). As explained in Algorithm 10, iCheck-related data structures will be filled with information for the scheduler to utilize during the beginning of the scheduler pass.

Since applications have a higher priority in this strategy, the scheduler will only set the *get_ic_node* flag with *HIGH* for the resource request from the iCheck controller with priority `HIGH` instead of assigning new nodes immediately (Lines 3 - 6 in Algorithm 11). Additionally, the scheduler will discard the lower-priority resource requests in the $get\_ic\_node\_info$ function.

After that, the runtime system can perform the runtime reconfiguration of malleable applications based on the performance-aware strategy (in Section 4.2.3) or power-aware strategy(in Section 5.2) (Line 26). Before calculating the new resource configuration, the scheduler will call the *get_ic_node* function (Lines 8 - 17) to get the iCheck node count that can be freed. As seen in the $get\_ic\_node()$, the potential candidate count is calculated based on the available memory in the iCheck nodes (Lines 12 - 15). For this, `iRM` uses the information about the iCheck system obtained from the *StatusMessage* (See Section 7.3.1.1). The potential candidate count from iCheck is also analyzed for calculating the new resource distribution (Line 29). Additionally, the scheduler parse through the waiting job queue and matches a job with a node count similar to the potential icheck node count. The exact match of node count is unnecessary since `iRM` supports job moldability.

Additionally, the scheduler will check for available idle nodes at the end of the scheduler pass (Line 34). If the *get_ic_node* flag with *HIGH* priority is set and idle nodes are available, the scheduler will send the idle node information to iCheck using the $sendiCheckInfo(...)$ call (Line 36). The scheduler then marks the idle nodes as busy (Line 37). If enough idle nodes are absent, the scheduler will only provide the available ones and discard the remaining request (Line 38). This means only the available idle nodes are sent to iCheck and `iRM` will not keep track of the requests in the later scheduler pass.

There are benefits associated with malleable applications and jobs waiting associated with this strategy. Favoring malleable applications provides additional nodes to reward higher `MTCT` applications. `iRM` can also use the $ic\_candidate\_count$ to launch new applications, thereby improving the average job waiting time.

7.3.3 Controller-iRM-Application Interaction

A bird's eye view of interactions happening between components of iCheck, `iRM` and an application is depicted in Figure 7.1. The application contacts `iRM` with performance metrics using `iMPI` and Slurm daemons. In addition, the application also contacts iCheck for availing checkpointing services. Meanwhile, `iRM` contacts iCheck with resource information (messages, redistribution requests) and `iRM` contacts application with resource redistribution information. Similarly, iCheck contacts `iRM` for resource requests and applications for passing checkpointing information.

As seen in Figure 7.1, communication between two components is independent of the other. iCheck does not need to account for the interactions between `iRM` and the application. Similarly, `iRM` can remain agnostic to the communication between the application and iCheck. Each interaction is independent among these components. iCheck and `iRM` interact with each other regarding the malleability part. Even though the interactions between iCheck and `iRM` are transparent to the third component application, the result of communication will be visible across different components. For example, the application's resources get redistributed upon a successful interaction between `iRM` and iCheck.

Algorithm 11. The iCheck aware scheduling function with priority for malleable application.

```
Function get_ic_node_info:
    Read the iCheck related information
    if Message from iCheck == ResourceRequestMessage and Priority == HIGH then
        read ic_num_nodes requested
        set flag get_ic_node to HIGH
    end
return
Function get_ic_node:
    set ic_candidate_count as 0
    if get_ic_node != HIGH then
        Read the iCheck node related information in ic_node_list
        for each iCheck node k in ic_node_list do
            add to ic_candidate with nodes above 90% available memory
            increment ic_candidate_count
        end
    end
    return ic_candidate_count
return
Function iCheck_Aware_Schedule_Favor_App:
    Get workload and resource information
    Read the iCheck related information
    if no malleable job is adapting and number of running malleable jobs > 0 then
        call the function get_ic_node_info
        if strategy == favor_app then
            call function get_ic_node and get potential ic_candidate_count
            Find new resource distribution based on perf_aware or power_aware strategies
            perform new resource redistribution
            for each waiting job j in priority order do
                Compute job requirements of j and compare with remaining ic_candidate_count
            end
            Call sendiCheckInfo(ResourceChangeMessage, ic_candidate_count)
            Pick new job from the waiting Queue when resources are available
        end
        if get_ic_node == HIGH and idle nodes available then
            get idle_node_name, idle_node_info
            call function sendiCheckInfo(ResourceChangeMessage, idle_node_info, ...)
            mark idle_node_name as busy
            reset get_ic_node
        end
    end
return
```

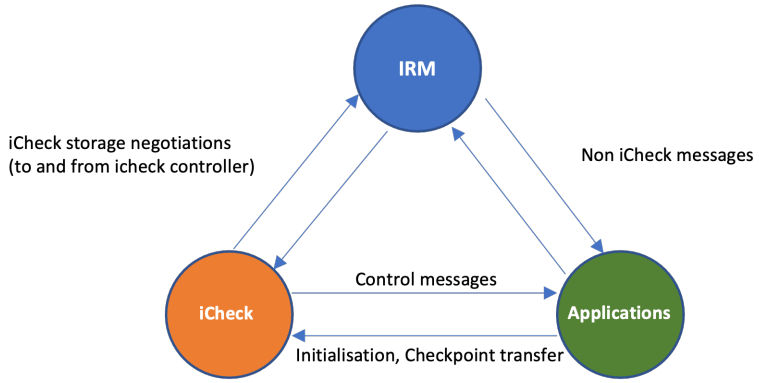


Figure 7.1: High level interaction overview of iCheck, iRM, and application from an iCheck Perspective.

7.4 Data Distribution in iCheck

The previous sections provided information about the application-level and system-level dynamism for checkpointing in iCheck using malleable applications. In addition to providing checkpointing services, a checkpointing system can also be regarded as a data distribution service for malleable applications. iCheck demonstrates this by providing data redistribution services to applications developed using `iMPI`. Below subsections cover different aspects of the data distribution framework provided by iCheck.

7.4.1 API for Data Redistribution

A proper API is needed for the application to benefit from the data distribution service. Towards that, iCheck proposes a set of special API calls based on the characteristics of the available data objects for defining and utilizing the dynamic data distribution feature in iCheck. A data object can be a variable, constant, data structure, structure object, or graph in iCheck. When an application is restarted or reconfigured with a different number of processes, the data object in an application can be ❶ local to a process (For example, the size of a communicator in MPI application), ❷ replicated among the processes (For example, the loop iterator variable), or ❸ distributed among the processes (For example, a 2D array distributed among N processes). These data objects should be restored or redistributed without violating the correctness of the application, and special APIs are needed to define data objects and label their characteristics. In a nutshell, the data objects in a distributed memory application can be categorized as *replicated* or *distributed* based on the characteristics of the data.

7.4.1.1 Replicated Data

Replicated data objects are local to each process, and their properties will be the same among all the participating processes. However, the data values may be different. For example, a loop control variable i of type integer is replicated in all the processes but with different values. Meanwhile, distributed data objects are scattered across all processes. For example, an array A of size M can be distributed among N processes with different sizes (For example, when M is not completely divisible by N) in every process. Combining all the processes' elements will constitute the complete data object A. In contrast, each instance is a complete data object for a replicated variable.

Replicated objects can be *volatile* or *constant*. The values of variables can be different in different processes for volatile objects. For example, a variable for storing the value of an MPI rank is volatile. However, values remain the same for constant objects and can hence be copied to the new processes without any modification during resource changes. `icheck_add_adapt()` is the new API provided by iCheck to simplify the checkpointing in dynamic processes. It is an extension of `icheck_add` with special features.

- `icheck_add_adapt("LABEL",..., attribute)` – The attribute parameter describe the data objects characteristics. The attributes provided by iCheck are `CONSTANT, SIZE, RANK, DEFAULT, VOLATILE`.

`CONSTANT` can be used if the value is the same among all the processes and can be copied to new processes whenever there is a restart with a different number of processes. However, a constant value in a standard checkpointing system can be volatile in a malleable checkpointing system. For example, the process size is used to divide the grid for efficient load balancing in a typical scientific application scenario. It is trivial for a standard checkpointing system to restore these variables since it is expected to remain the same during the entire execution time for a rigid application. However, in case of restart with a new number of processes or resource change during the application execution, they should be recalculated if these variables cannot be restored as it is. In such a scenario, attribute `VOLATILE` can be used.

It is trivial to identify some of the most commonly used volatile variables in a malleable application. A simple example is the number of processes in a distributed memory application. If an application was started with process count N (saved in a variable called *num_procs*) and is saved in the checkpoint using `icheck_add`. If the application is restarted/reconfigured with a new process count M, then it must be desirable that the value of the variable *num_procs* becomes M. In such a scenario, the attribute provided by iCheck can be used to label the variable as volatile. `SIZE, RANK, DEFAULT, VOLATILE, RECALC` are some of the attributes in iCheck.

- `icheck_add_adapt("label",..., num_procs, SIZE)` – If called with `SIZE` attribute, iCheck will restore the variable with the new application size.
- `icheck_add_adapt("label",..., rank, RANK)` – If called with `RANK` attribute, iCheck will restore the variable with the current rank.
- `icheck_add_adapt("label",..., rank, DEFAULT, default_value)` – If the relation between the process and data object is not a simple mapping, `DEFAULT` attribute along with $default\_value$ values can be provided to use it during the restore operation.
- `icheck_add_adapt("label",..., rank, DEFAULT, default_value, JOINING)` – Here, the parameter $default\_value$ value will only be used for the newly joining processes. All the existing processes use the most recently stored values.

A data object can be dependent on other data objects in complex scenarios. For example, a variable *array_length* will be associated with a particular array. The attribute `VOLATILE` can be used in such a scenario. During the reconfiguration, iCheck needs to recalculate the new value for this data object. Since these values are application dependent, the application should provide the new mapping needed to calculate the value. In that case, iCheck can ask the user to provide a *Recalculate Function*. As a result, whenever a particular data object is restored, this function will be called immediately before restoring the value from the checkpoint.

For example,

- `icheck_add_adapt("label",...,variable,VOLATILE,recalc_func,args recalc_func)` – During the resource reconfiguration/restart the $recalc\_func(args)$ will be called with the provided arguments.

7.4.1.2 Distributed Data

Data distribution plays a vital role in distributed memory applications. Data is often distributed according to the topology (virtual or physical) and application type. Restoring such distributed data is a trivial process for the existing checkpointing system. Data associated with the process can be restored, and the non-malleable application can be restarted successfully. However, restarting with a different number of processes is complicated with distributed data. Tackling such an issue is critical for a resource-aware checkpointing system. Consider a simple $2D$ array where the application distributes the data along a grid with dimension *NxN*, where *N* is the number of processes. If *N_new* is the new number of processes during the reconfiguration/restart of the application, the saved *NxN* data should be gathered and redistributed across the new *N_new* processes. This adds an extra layer of complexity in the checkpointing system. Distributed data in an application can either be *structured* or *unstructured*. The below paragraphs provide an overview of the different characteristics of distributed data.

Structured data: The data is distributed based on some predefined mapping for the structured data. For example, suppose the data is distributed as `CYLIC` or `BLOCK` across the processes. In that case, the restart with a new number of processes can be handled with less complexity. iCheck should only follow the same data distribution scheme and spread the data among all the processes accordingly. In other words, it should use the same data partitioning scheme. Hence, iCheck must know the partitioning scheme for a successful application restart. Towards that, iCheck provides two special functions `icheck_add_processor_mapping` and `icheck_add_adapt_global`. The former can be used by application developers to provide partitioning schemes needed for a successful restart. The latter can be used to pass the partitioning schemes created as part of `icheck_add_processor_mapping` to the iCheck system.

Multiple partitioning schemes can be predefined by programmers for the potential resource configurations that might be encountered during an application restart/reconfiguration, and iCheck will pick the suitable scheme during the restart based on the new resource configuration.

- `icheck_add_processor_mapping(num_procs,dim,...,partitioning_array)` – In this case, if the application is restarted with a process number *num_procs*, then the data should be distributed as $dim[0] * dim[1]$. This information is added to an iCheck structure called *partitioning_array*. Partitioning array is a two dimensional array specific to iCheck where it stores information regarding the number of processes, dimensions of the topology in which the data should be distributed. This array is populated automatically by iCheck based on the values given in the `icheck_add_processor_mapping()` call. Multiple processor mapping can be added similarly to the same *partitioning_array* by repeatedly calling `icheck_add_processor_mapping()` with different parameters. This partitioning array will be passed as an argument to the function `icheck_add_adapt_global`.
- `icheck_add_adapt_global(label,array,...,PREDEFINED,partitioning_array)` – This call will add *array* along with metadata mentioned in `partitioning_array` to the iCheck system. During restart, the provided partitioning scheme will be used to redistribute the data. `PREDEFINED` attributes specify that the partitioning array contains the processor mapping for redistribution.

The following example can be used to specify multiple partitioning schemes for a data object in an application that can be redistributed with process sizes 8 and 16 for a grid with 1600 elements across dimensions $2X4$ (where $dim[0]$ is 2 and $dim[1]$ is 4 for the first call) and $4X4$ (where $dim[0]$ is 4 and $dim[1]$ is 4 for the second call):

- `icheck_add_processor_mapping(8,dim,1600,pmap_array)`
- `icheck_add_processor_mapping(16,dim,1600,pmap_array)`

After adding the partitioning scheme to *pmap_array*, use it in the special add API Call:

- `icheck_add_adapt_global("array",local_array,type,size,PREDEFINED, pmap_ array)`

To simplify the data redistribution for cyclic and block distribution in 1D arrays, keywords `CYCLIC` and `BLOCK` provided by iCheck can be availed.

- `icheck_add_adapt_global("array", array, size, CYCLIC, width)` – `width` specifies the distribution size. For a `width` of 1 with `CYCLIC` distribution, the data is distributed across participating processes cyclically with a size of 1.
- `icheck_add_adapt_global("array", array, size, BLOCK, width)` – For a width of `width` using `BLOCK` distribution, the data is distributed across participating processes in a block with a block size of `width`.

Unstructured data: If the data distribution cannot be defined using simple partitioning schemes, the application must redistribute the data after a restart. For example, in a master worker model application, iCheck can provide the complete data of the data object (e.g., an unstructured distributed array) checkpointed in the system to a specified process. `MANUAL` attribute can be used inside `icheck_add_adapt_global()` to inform iCheck about the unstructured data, which is explained below.

- `icheck_add_adapt_global(label,array, size, MANUAL, ALL, restart_function, args, src)` – The iCheck library will transfer the complete data into an *array* and pass it to the `restart_function` in the process number specified using the parameter *src*. All of the participating processes in the application will call `restart_function`, and only the process with rank *src* has the checkpointed data inside *array*. Using that, the application can manually redistribute the data using `restart_function`.

Using the above special variants of `icheck_add` routines, the metadata regarding the checkpointed data can be passed to the iCheck system. During the restart, iCheck will utilise this metadata to restore the correct values into the provided data structures when the function `icheck_restore` is called. Even though this is sufficient for redistributing the data during resource adaptation in malleable applications, another API call `icheck_redistribute` is provided to avoid confusion. This API call is analogous to restore and can be used after the `MPI_Comm_adapt_commit` function call.

- `icheck_redistribute("label",data,...,size)` – In this function call, iCheck will use the metadata associated with *label* to redistribute and restore the checkpointed data into the data structure *data*.

7.4.2 Pseudocode for Data Distribution

Listing 7.2 presents the pseudocode of a malleable application using `iMPI` that employs the special APIs from iCheck for data distribution. The control flow of the `iMPI` application in the context of iCheck is described in Listing 7.1. Here, the explanation is only provided in the context of data distribution. The newly introduced API calls can be seen in lines 7, 17 and 29 of Listing 7.2. `icheck_add_adapt` (Line 7) adds the metadata for data object *data* into iCheck using the label *dis_data*. To finalise a resource change in `iMPI` application, the joining and preexisting processes call the `MPI_Comm_adapt_commit` function (Lines 16 and 28). Immediately following the resource change, the new data redistribution API `icheck_redistribute` is called (Line 17 and Line 29). It is a collective operation, and all processes should call this operation to perform the data distribution. During this call,

- the iCheck library calls `icheck_probe_agents` internally to obtain the new agent reconfiguration based on the agent selection strategy explained in Algorithm 7.
- the iCheck library will transfer the data from the agents into the preexisting processes. After complete data is available inside the library, iCheck redistributes it among the new set of processes.

Nevertheless, the process of agent change and data redistribution remains transparent to the application. After this call, the *data* contains the checkpointed data associated with each process, and the application can resume the execution. Nevertheless, if no resource redistribution happens and application is restarted with a different number of processes, the `icheck_restore` in Line 10 will redistribute the data based on the provided metadata. At present, iCheck only supports 1D arrays and simple data redistribution techniques like block (Line 7) or cyclic during a resource change in an application.

```
1  #include<icheck.h>
2  int main() {
3   /* Initialisation part of the application */
4   MPI_Init_adapt(..., type) /* iMPI is intialised */
5   float data[SIZE];
6   icheck_init(..., type);
7   icheck_add_adapt("dis_data",data,...,width, BLOCK);
8   if(checkpoint_available && no_adapt){
9    icheck_restart();
10   icheck_restore("dis_data",data,..,new_size);
11  }
12  if (type == joining) {
13    /* Start of adaptation window in joining processes */
14    MPI_Comm_adapt_begin(...);
15    /* Get metadata from preexisting processes, perform data redistribution */
16    MPI_Comm_adapt_commit();
17    icheck_redistribute("dis_data",data,...,new_size);
18  }
19  /* Iterate over compute intensive phase of the application */
20
21  while (true){
22    /* iMPI processes must call probe periodically */
23    MPI_Probe_adapt(resource_change,...);
24    if (resource_change) {
25      /* Start of adaptation window in preexisting processes */
26      MPI_Comm_adapt_begin(...);
27      /* Pass metadata to joining processes, Perform Data distribution. */
28      MPI_Comm_adapt_commit();
29      icheck_redistribute("dis_data",data,...,new_size);
30    }
31    /* Compute part of the application */
32    /* Read/Write data[]*/
33    if(iteration%100)
34      icheck_commit();
35    /* Check for agent change*/
36    if(iteration%1000)
37      icheck_probe_agents();
38    }
39    icheck_finalize(...);
40    /* Finalisation part of the application */
41    MPI_Finalize();
42  }
43
```

Listing 7.2: Psuedocode of a malleable application using iCheck data distribution API [77].

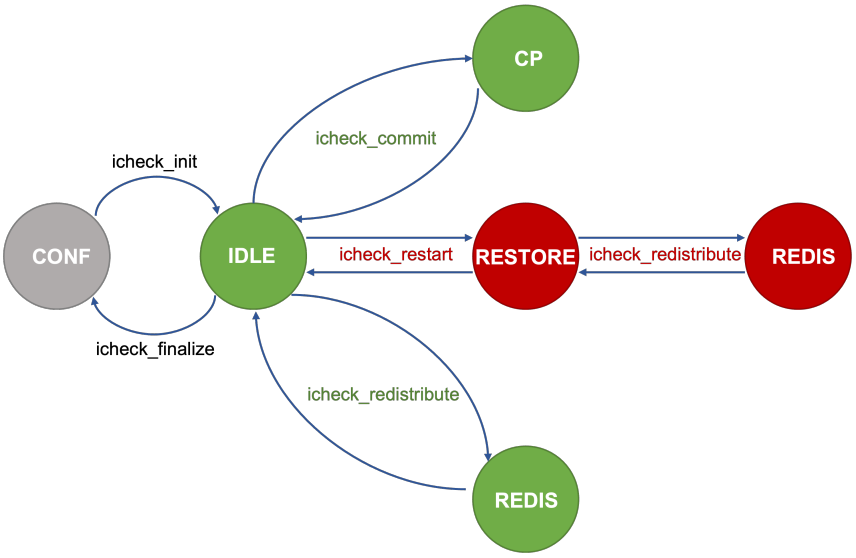


Figure 7.2: State transition diagram of iCheck library (High level overview).

7.4.3 State Diagram

The state transition diagram of iCheck library can be seen in Figure 7.2. If no checkpointing operation is performed, the iCheck library will be in an `IDLE` state. When an application starts, the library will contact the iCheck core and initialize the checkpointing services for the application, transitioning into a `CONF` state from `IDLE`. Whenever an application writes checkpoints, iCheck will go from `IDLE` to `CP` state (or store checkpoint state) and transfer the checkpoints to a remote node in the iCheck system. After the checkpoint operation, iCheck will return to the `IDLE` state. During a resource change, iCheck will go into data redistribution `REDIS` state and performs the redistribution based on the application requirements (It is necessary that the application should checkpoint its data immediately before the resource adaptation to ensure that the most recent value of the data will be redistributed after a resource change). After that, the state of the iCheck library will change from `REDIS` to `IDLE`. During an application restart, iCheck must re-enter the `CONF` state, later transitioning into the checkpoint `RESTORE` state (restore checkpoint operation happens). Based on the provided mapping, iCheck library redistributes the data (go into `REDIS` state) or transfers the checkpoint data to individual process. Following that, iCheck will be in the `IDLE` state. The application will continue execution whenever iCheck library is in `IDLE` state. During the application end, iCheck will go into the `CONF` state for finalization.

7.4.4 Resource Malleability Using Checkpointing

By using the data distribution API from iCheck, it can be demonstrated that the checkpointing system can bring Malleability to Non-Malleable Applications. Any resource manager paired with an adaptive checkpointing system that supports data redistribution can be considered a malleable infrastructure. For example, the application can specify the data distribution scheme for the restart with a new number of resources. The checkpointing system can use this information to inform the resource manager about the flexibility of the application to be restarted with a different number of resources. Using this information, the resource manager can kill the application and relaunch the application with different resources. This gives flexibility to the resource manager to reorganise the application to improve the system metrics.

8

Evaluation Setup

This chapter gives an overview of the evaluation setup used for validating and verifying the contributions mentioned in Chapters 4 - 7. This chapter is divided into two sections. The first section describes the system setup used to evaluate the contributions. It also explains how it was made possible to run a complete Slurm resource manager inside a Slurm job. The second section discusses the applications used to assess the contributions of this work and the setup used to evaluate the work. The applications used for adaptive batch scheduling (Chapter 4), power corridor management (Chapter (5)), and iCheck (Chapters 6 and 7) are described in the order in which it is introduced in this work.

8.1 System Setup

The software and plugins developed as part of this work were tested on the supercomputing infrastructure (SuperMUC) hosted at Leibniz-Rechenzentrum (LRZ), Germany. During the proposed work, two different models of SuperMUC were hosted at LRZ. The first model, SuperMUC Phase 2 [211], was replaced by an advanced model named SuperMUC Next Generation (SuperMUC-NG [56]). The evaluations were performed on SuperMUC Phase 2 and SuperMUC-NG based on the chronology of the work.

8.1.1 SuperMUC Phase 2

The SuperMUC Phase 2 was a powerful PetaScale System composed of six islands of Lenovo's NeXtScale nx360M5 WCT systems [212], with each island hosting 512 nodes. Each node in this system was equipped with two Intel Haswell Xeon E5-2697v3 processors (28 cores per processor) [213], equipped with 64 GB of RAM. The nodes were interconnected using an Infiniband FDR14 network, featuring a non-blocking intra-island topology and a pruned 4 : 1 inter-island tree topology. Altogether, the system had 86,016 cores and 194 TB of RAM, delivering a performance of 2.81 PetaFlop/s. Storage was supported by a parallel filesystem based on IBM's GPFS [214], providing 15 PB of space. The system operated on SUSE Linux Enterprise Edition 11 and used IBM's LoadLeveler [52] for job scheduling and batch processing.

8.1.2 SuperMUC NG

SuperMUC-NG [215] is a PetaScale System featuring 6,480 compute nodes distributed across eight islands, amounting to 311,040 compute cores, all based on the Intel Skylake-SP architecture [216]. This setup enables a peak performance of 26.9 PetaFlop/s. Each compute node in the system includes two sockets, each equipped with an Intel Xeon Platinum 8174 processor [217], totalling 24 cores per processor and 96 GB of RAM per node. The network connectivity among compute nodes is facilitated by a high-bandwidth Intel OmniPath [207] Interconnect with a fat-tree topology, ensuring efficient data transfers. The core frequency for each processor is maintained at a nominal 3.10 GHz. For workload management, SuperMUC-NG employs Slurm [50]. Additionally, features like Hyper-Threading and Turbo Boost are deactivated to maintain consistent performance metrics across the system.

8.1.3 Running RJMS inside RJMS

To evaluate the performance and power-aware batch scheduling strategies proposed in this work, a virtual cluster is created inside the job allocation of the SuperMUC system with the modified SLURM as RJMS on the compute nodes (virtual cluster). Adaptive job management system is bootstrapped on the virtual cluster and the jobs submitted inside the virtual cluster are managed by the adaptive job management system. However, it is not a trivial use case and is not easy to achieve.

As mentioned in Chapter 7.4.4, supercomputing centres employ RJMS software for efficiently managing the resources and job submissions. Users will provide their job requirements (For example, number of nodes, job duration) and the application information (For example, application path, input arguments) to run. When resources are available, and the job requirements can be met, the RJMS system allocates the resources for the job and launches the application specified in the job script. Even though this is a standard operating procedure in every supercomputing centre, it is unsuitable for evaluating resource managers (instead of applications). There are no infrastructure deployment tools (For example, Ansible [218], Terraform [219], Kubernetes [220]) available in these centres to deploy and test the resource management infrastructure with ease for normal users. Nevertheless, these centres only provide user-level access, so users cannot install these tools. Hence, it is not trivial, and the resource managers are tested using simulators [221].

However, [102] provides techniques to test the infrastructure by deploying Slurm (`iRM` specified in Chapter 2) inside a load leveler. This technique can be employed to test the infrastructure on SuperMUC Phase 2 since `iRM` is an extension of Slurm, and SuperMUC Phase 2 uses the load leveler as RJMS. However, applying this technique to SuperMUC-NG is impossible since both the SuperMUC-NG and the proposed work (`ABS`) use Slurm as RJMS. As a result, evaluation becomes challenging because the initial job script should be submitted to the default Slurm in SuperMUC-NG. When the job allocations are granted, the private version of the slurm should be booted up using the job script. The unavailability of infrastructure deployment tools (For example, Ansible [218], Terraform [219], Kubernetes [220]) means the standard bash scripts should be used to boot up the virtual Slurm cluster. Once the test Slurm is up and running, a new job script should be used to submit the job to the test Slurm. It is particularly challenging because the binary names (For example, `sbatch, srun`) are the same for both Slurms, and both Slurms use the same set of environmental variables (For example, `SLURM_JOB_ID, SLURM_TASKS_PER_NODE`).

Additionally, Slurm uses an authentication service called MUNGE (MUNGE Uid 'N' Gid Emporium) [222] for authenticating the nodes. It authenticates the UID and GID of processes (remote or local) within a group of hosts having common users and groups. A security realm is formed using these hosts and is characterized by a common cryptographic key. Hence, a new MUNGE should be explicitly used for the test Slurm.

Furthermore, the new dedicated MUNGE must be patched to support the test Slurm. Similarly, Slurm configuration and defaults must be modified to avoid conflict with the MUNGE used in the default Slurm. In addition, all of these changes should be made in the context of a user space (no super-user privileges).

Nevertheless, the proposed work developed a strategy to run one slurm cluster inside another slurm within the constraints of a job script. Towards that, a set of scripts was developed to automate the process and guidelines for installing and deploying Slurm and MUNGE were identified. It is a cost-effective way of testing resource managers on production-level systems without affecting their working and functionality. As a result, dedicating a separate smaller cluster to test the resource managers is unnecessary. After the end of the job execution, default Slurm can reclaim the resources as it does for any other jobs.

8.2 Application Setup

Eight different applications were used to evaluate the performance and functional aspects of the proposed work. Each contribution used a subset of these eight applications to analyse the system's performance. The applications used for analysing the adaptive batch scheduling and power corridor management systems were not developed as part of this work. However, the applications used for the checkpointing analysis were extensively modified (or were entirely developed from scratch) as part of this work to utilise iCheck. More details about the applications can be seen in the following subsections.

8.2.1 Adaptive Batch Scheduling

The proposed scheduling strategy was evaluated on SuperMUC-NG [215]. The performance and effectiveness of the strategy were analysed using the modified Effective System Performance (ESP) benchmark [223]. This benchmark is used to evaluate the performance of RJMS software and is an effective tool for comparative analysis of various scheduling strategies [115, 224]. This technique consists of 230 jobs of 14 different types, each running the same synthetic application with a unique execution time. Table 9.1 shows the job types, instance counts, fraction of total system size, and target runtimes. In the evaluation scheme employed in this work, the synthetic application is replaced with a malleable tsunami simulation [225] application. This simulation was modelled using the 2-D shallow water wave equation and uses `iMPI` for parallelism. The Tsunami simulation is a representative case of a real-world scientific application developed using the $sam(oa)^2$ framework for dynamically adaptive meshes. At every simulation time step, the framework ensures that grid refinement and load are balancing taking place.

8.2.2 Power Corridor Management

SuperMUC Phase 2 [211] was leveraged to evaluate the proposed power corridor management (pcm) infrastructure and was executed as a standard Load Leveler job. The tests for forecasting and upper power corridor enforcement on pcm infrastructure was evaluated on 32 nodes (896 processes) while the remaining analysis were performed on 16 nodes (448 processes). Three `iMPI` applications were used to evaluate the infrastructure. They are

1. Heat Simulation: This application simulates the heat transfer in a metal plate. The material in this code is divided into a grid of cells, and heat is applied to the borders of the 2D plane. Through the Laplace equation [226], the transfer of heat from one grid point to another over time is determined. To solve this equation for heat transfer, the Jacobi iterative [227] method is utilised. This method

```
cp_size = 100000
cp_type = float
cp_interval = 10 # in seconds
cp_count = 1000
cp_mpiio = true
enable_icheck = false
fault_inject = 10 # in seconds
probe_interval = 5 # in seconds
is_invasive = true
```

Listing 8.1: The configuration file for synthetic benchmark application.

involves continuously refining the solution until a stable solution is obtained or a fixed number of steps have been completed.

2. LU decomposition: LU decomposition is a widely used technique for solving linear systems and finding inverse matrices [228]. It is deemed a more efficient approach for solving linear systems with the repeated left-hand side, such as when solving the equation $Ax = b$ with different values of b for the same A. In this technique, matrix A is decomposed into two separate matrices, L and U, where L is a lower triangular matrix, and U is an upper triangular matrix. This decomposition allows for the efficient solution of linear systems, as it reduces the problem's computational complexity.
3. Pi calculation: This distributed application calculates the value of Pi using the Leibniz formula [229]. The Leibniz formula is a popular method for calculating the value of Pi using an infinite series. While the formula is straightforward, it requires a large number of iterations to produce a low-precision value of Pi.

The power-aware scheduler was evaluated on SuperMUC-NG [215] using the applications Heat Simulation and Pi calculation.

8.2.3 iCheck – Performance and Resource Adaptivity Analysis

The analysis of the iCheck system is performed on the SuperMUC-NG [56]. It is tested using the below four standard MPI applications:

1. ls1 mardyn [230]: The ls1 mardyn is a highly scalable molecular dynamics simulation code optimised for massively parallel execution on high-performance computing systems and allows application of pair potentials to length and time scales. In addition to dynamic load balancing schemes, it supports multicenter rigid potential models based on Lennard-Jones sites, point charges, and higher-order polarities.
2. LULESH [231]: Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics, or LULESH, is a proxy application representing a hydrocode. The application finds the solution to the hydrodynamics equations by the approximation technique. The spatial problem domain is partitioned into a collection of volumetric elements using the concept of unstructured hex mesh.
3. Synthetic Benchmark: It is a synthetic application that can checkpoint the data of a prescribed size (checkpoint size) in the provided intervals (checkpoint interval). The values for size and interval can be provided before the application launch via the configuration file (Listing 8.1). In addition, faults can be injected into the application by specifying the duration after which the application should be killed in the configuration file.

4. Heat Simulation: A non iMPI version of the heat simulation code introduced in the above section is used (See section 8.2.2).

iCheck support is added to ls1 mardyn, LULESH, and heat simulation codes. The synthetic benchmark application with iCheck and MPI-IO support was also created as part of the evaluation process. The ls1 mardyn has an in-house MPI-IO-based application-level checkpointing implementation and is used as a basis for comparing with iCheck. Furthermore, extensive iCheck library overhead analysis, dynamic agent evaluations, and comparison with MPI-IO in the ls1 mardyn and synthetic benchmark application were also performed. LULESH and the heat simulation application were used along with ls1 mardyn and synthetic application to demonstrate the adaptive nature of iCheck.

```
1  #include<icheck.h>
2  int main() {
3    MPI_Init(NULL, NULL);
4    void *data;
5    /* Read checkpoint size, intervals etc from config file */
6    read_config();
7    icheck_init(argv[1], "testApp", MPI_COMM_WORLD, status);
8    /* Declare datastructure with size and type from the configuration file */
9    data = (config_file_type*)malloc(config_file_size);
10   icheck_add("Array", data, config_file_size, config_file_type);
11   if(is_checkpoint_available) {
12     icheck_restart();
13     icheck_restore("Array",data,config_file_size);
14   }
15   for(i = 0; i<config_file_count; i++) {
16     /* Modify the array data[]*/
17     /* Perform checkpointing on the provided intervals */
18     if(time_elapsed_last_cp > config_file_interval) {
19       icheck_commit();
20       /* Perform MPI-IO based checkpointing if enabled in config file */
21       if (config_mpiio_true)
22          checkpoint_mpiio();
23       }
24     /* Kill the application after the specified time period */
25     if(time_elapsed_app_begin > config_file_fault)
26       exit();
27     /* Check for agent change in specified intervals */
28     if(config_file_probe_interval)
29       icheck_probe_agents(hints);
30   }
31   icheck_finalize(IC_PERSIST);
32   MPI_Finalize();
33 }
```

Listing 8.2: Pseudocode of a synthetic benchmark using iCheck.

8.2.3.1 Synthetic Application

The pseudocode of the synthetic benchmark can be seen in Listing 8.2. The application will read the configuration parameters (See line 6 in Listing 8.2) from the synthetic benchmark configuration file. Listing 8.1 shows the sample configuration file. The checkpoint configuration file can provide information like

checkpoint type, size, and frequency (See Lines 1-3 in Listing 8.1). In addition, information like whether to inject fault or enable MPI-IO-based checkpointing for timing analysis can also be provided (See Lines 5-6 in Listing 8.1). The benchmark supports more options, which are not detailed here due to the brevity.

Based on the type and size provided in the configuration file, the synthetic benchmark allocates memory. It creates the data structure for storing the checkpoint (See Lines 9 - 10 in Listing 8.2). The checkpoint transfer (See lines 18 - 19) will be based on the `cp_interval` specified in the configuration file. The fault injection (Lines 25 - 26) and check for agent change (Lines 28 - 29) will also be based on the values provided in the configuration file. Furthermore, the configuration file can turn the MPI-IO-based checkpointing (Lines 32 -22) on or off. In addition, iCheck can be turned off or on using the variable `enable_icheck` mentioned in the configuration file. It is not shown in the pseudocode to avoid the complexity of the control flow.

8.2.3.2 Synthetic Malleable Application

The invasive nature of the iCheck system is assessed with a synthetic iMPI application. It is an extension of the synthetic benchmark described in the above section (Section 8.2.3.1). The configuration parameters remain the same for both MPI and iMPI synthetic benchmarks with an additional variable called `is_invasive` (See Line 9 in Listing 8.1). The variable `is_invasive` is exclusive for the iMPI synthetic application, and if `true`, the application follows the control flow for the `iMPI` application; otherwise, it runs as the synthetic benchmark application defined in section 8.2.3.1.

9

Results

This chapter analyses and evaluates the contributions made in this work. The contributions are analysed using the system setup and applications described in Chapter 8. The four following sections describe the results of the contributions in the order of the definition of their corresponding chapters (Chapters 4 - 7). The first section describes the results associated with adaptive batch scheduling, the second section describes the results of power corridor management contribution, and the third and fourth sections provide the results regarding the contributions associated with iCheck.

9.1 Adaptive Batch Scheduling

As described in Section 8.2.1, the Effective System Performance (ESP) [223] benchmark was adapted and extended to evaluate the performance of the proposed adaptive batch scheduling system. The modified ESP consists of 230 jobs derived from 14 job types running the same Tsunami simulation[1] [225] (iMPI) application with a fixed unique execution time per job. Table 9.1 showcases the workload characteristics of the modified ESP Benchmark suite. It displays information regarding type of Job, fraction of total system size, job instance counts, and target runtimes. Furthermore, constraints on the node count for dynamic resource reconfiguration decisions on each job type (see Section 2.2.1.2) are added to the ESP benchmark to evaluate the system. The varied target runtimes for each job type (see Table 9.1) are achieved by modifying the resolution of grid and total simulation time. For the performance evaluation on SuperMUC-NG, 34 compute nodes were utilised to adhere to the ESP benchmark requirements.

The performance of the proposed performance aware scheduling strategy (described in 4.1) is compared with static backfilling [41] and Favour Previously Started Malleable Applications (FPSMA) first [117] strategies (See section 4.2.2.1). Backfill scheduling [41] is a technique that allows lower-priority jobs to start if their expected start time does not delay any higher-priority jobs. The expected start time of pending jobs is determined based on the expected completion time of running jobs in backfill scheduling. As seen in Section (4.2.2.1), based on the start times (increasing/decreasing order), jobs are targeted for expansion/reduction in the FPSMA strategy. Four metrics are used to quantify the performance,

[1] https://github.com/mohellen/eSamoa

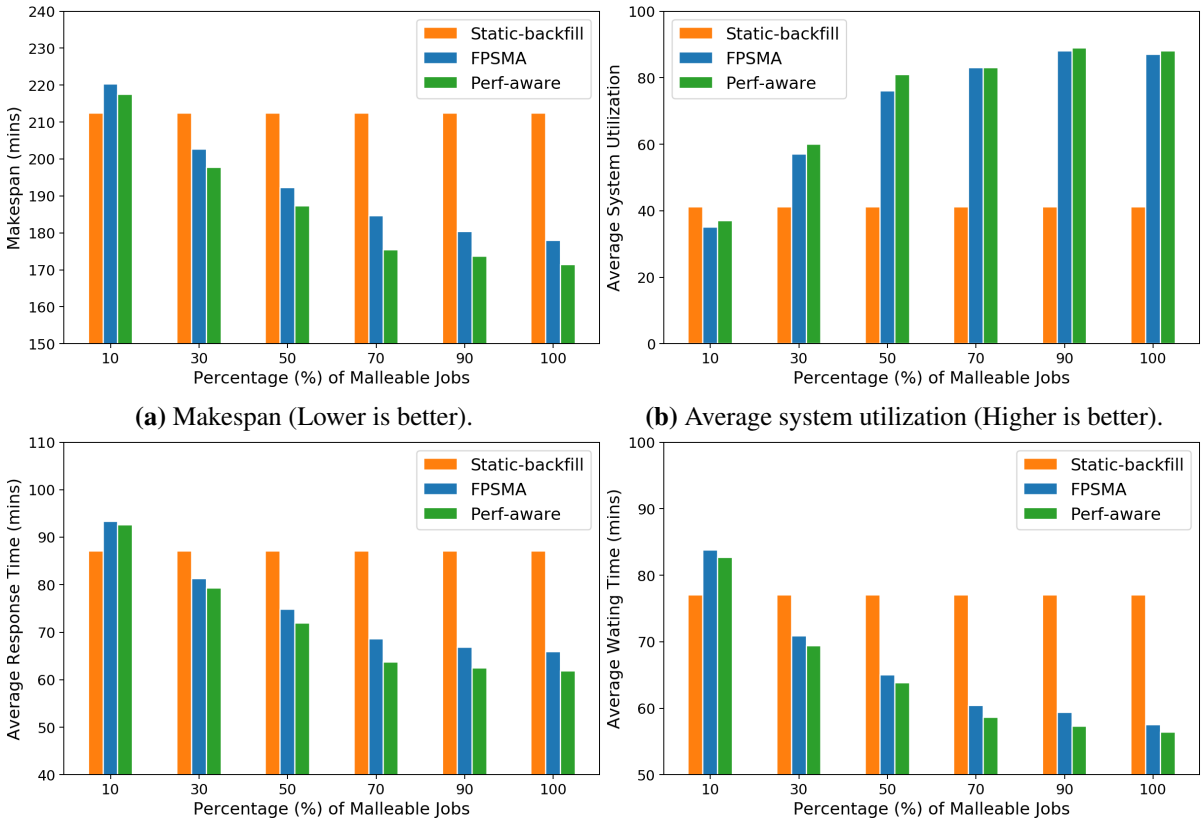
Table 9.1: Workload characteristics for the modified ESP Benchmark [223] used in ABS analysis [55].

Job Type	Fraction of System Size	Count	Static Execution Time [secs]	Constraints
A	0.03125	75	267	-
B	0.06250	9	322	pof2
C	0.50000	3	534	-
D	0.25000	3	616	even
E	0.50000	3	315	-
F	0.06250	9	1846	pof2
G	0.12500	6	1334	even
H	0.15625	6	1067	odd
I	0.03125	24	1432	-
J	0.06250	24	725	pof2
K	0.09375	15	487	-
L	0.12500	36	366	even
M	0.25000	15	187	-
Z	1	2	100	-

1. Makespan: It is the duration between the completion of the last job and the arrival time of the first one.
2. Average system utilisation: It is a metric representing the proportion of the total system utilised during the execution of entire workload.
3. Average waiting time: It is the interval between the start time and submission time of all jobs and is averaged across all of them.
4. Average response time: It is computed as the sum of the waiting time and runtime for all jobs and is averaged across all of them.

The analysis results for the three strategies with varying mixes of rigid and malleable jobs are illustrated in Figure 9.1. This figure shows the outcomes for the four metrics discussed earlier across the implemented strategies. The sequence in which jobs are submitted to `ABS` in the ESP benchmark is determined using a pseudo-random generator, with a consistent inter-arrival time of 30 seconds between jobs. This submission sequence is kept constant across all strategies and scenarios to ensure comparability. For malleable job selection in both the FPSMA and performance-aware strategies, a pseudo-random generator with a fixed seed is employed, ensuring consistency in job selection across these strategies. The number of nodes allocated to each malleable job is specified using the `--nodes` parameter, aligning with the system size. The minimum and maximum number of nodes for each job adhere to the lowest and highest values within the node constraints provided. For malleable jobs without specific constraints, the node range is set from one to 32. The metric outcomes for the static backfilling strategy are obtained by running Slurm's standard `backfill` scheduling plugin, which operates with default parameters and does not engage in any job expansion or reduction operations.

As depicted in Figures 9.1a - 9.1d, the performance-aware scheduling strategy demonstrates improved efficiency with 19.3%, 29.0%, and 26.8% reductions in makespan, average response, and waiting times, respectively, compared to the static backfilling strategy in scenarios with 100% malleable jobs. Compared to the malleable FPSMA strategy, the metric values obtained for makespan, average response and waiting time are 4.0%, 6.1%, and 2.0% lower for the performance-aware scheduling strategy. Regardless, the performance-aware strategy outshines the backfilling method in every scenario except when dealing with 10% malleable jobs. In this particular case, both the FPSMA and performance-aware strategies exhibit higher metric values,



(a) Makespan (Lower is better).

(b) Average system utilization (Higher is better).

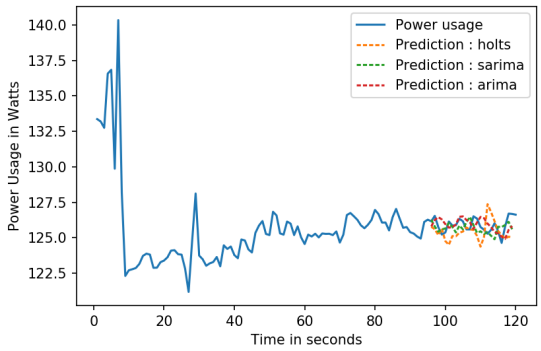
(c) Average response time (Lower is better).

(d) Average waiting time (Lower is better).

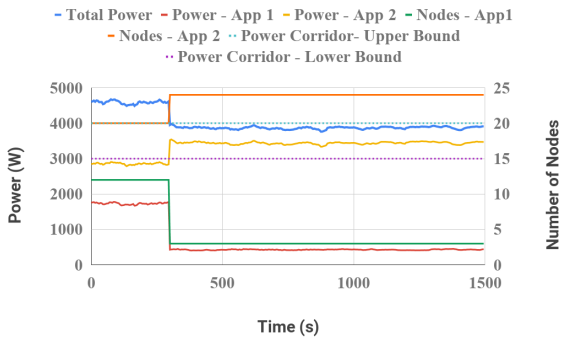
Figure 9.1: Evaluation of the adaptive batch scheduling system [55]

which can be attributed to two key factors: ❶ The specific node constraints for malleable jobs as defined in the ESP benchmark influence the scheduling outcomes. ❷ The malleable scheduling strategies, including both FPSMA and performance-aware, adhere to a First Come, First Served (FCFS) policy, unlike the static backfilling strategy, which is not bound by FCFS. Consequently, static backfilling can optimize job execution by choosing jobs that are ready to run immediately, thus efficiently utilizing available resources without disrupting the queued higher-priority jobs' resource reservations.

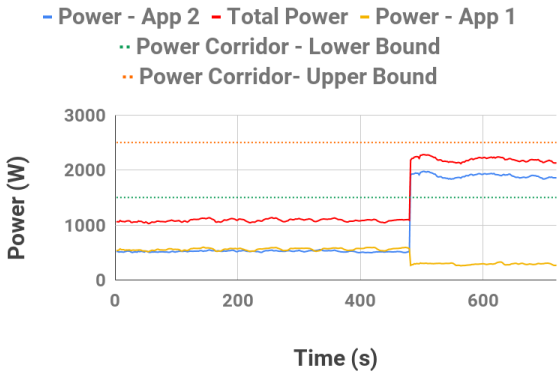
Figure 9.1b showcases a comparison of average system utilisation across the three scheduling strategies. The data illustrates that the malleable scheduling strategies, both FPSMA and performance-aware, generally surpass the backfilling approach in efficiency across all scenarios, with the exception of the one featuring 10% malleable jobs. This trend mirrors the observations made for the other three metrics previously discussed. The figure also reveals that the FPSMA and performance-aware strategies achieve similar levels of average system utilisation across various scenarios. However, the performance-aware strategy achieves greater throughput because it makes more efficient decisions for job expansion/reduction based on the current `MTCT` ratio, unlike the FPSMA strategy, which bases its decisions on job start times (refer to section 4.2.2.1 for FPSMA and section 4.2.3 for performance-aware scheduling). Furthermore, it can also be described based on the characteristics of the tsunami simulation application used in the analysis, which employs AMR [108]. The communication and computational requirements of the application are changed as the simulation undergoes a periodic grid refinement process. Therefore, a scheduling strategy that employs a heuristic criterion that takes into account the application specific characteristics results in better performance.



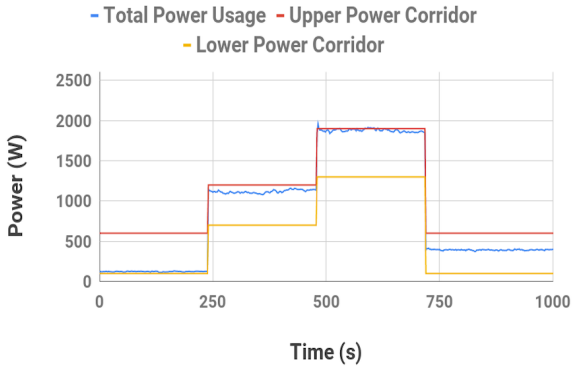
(a) Forecasting the power usage



(b) Upper power corridor enforcement



(c) Lower power corridor enforcement



(d) Dynamic power corridor enforcement

Figure 9.2: Power Corridor enforcement using the resource reconfiguration among the running malleable applications [59].

9.2 Power Corridor Management

The initial power corridor management framework was evaluated as a standard Load Leveler job on SuperMUC Phase 2 [215] and assessed using three `iMPI` applications: 2D Jacobi heat simulation, LU decomposition, and Pi calculation. On the other hand, the power-aware adaptive batch scheduler's evaluation was conducted on SuperMUC-NG using two `iMPI` applications: 2D Jacobi heat simulation and Pi calculation. For the initial assessment, the power corridor management infrastructure operated on 32 nodes, equivalent to 896 processes, to conduct tests on forecasting and enforcement of the upper power corridor. The tests for dynamic power corridor management were carried out on 16 nodes with a total of 448 processes. This evaluation illustrates that managing resources dynamically for running malleable applications can effectively enforce power corridors in systems subjected to changing power constraints. The initial evaluation results are detailed in Subsections 9.2.1 - 9.2.3. Meanwhile, the power-aware scheduler was evaluated on 16 nodes (768 processes) and is described in Subsection 9.2.4.

9.2.1 Forecasting

As detailed in Section 5.1.4, three different methods (ARIMA, SARIMAX, Holt-Winters) were employed to forecast the power usage prediction of applications in the power aware analysis. The forecasts generated by these three models are presented in Figure 9.2a. These models were trained in real-time, and `iRM` selected the most accurate one for its forecasting needs. The model's performance was assessed using the Mean Absolute Percentage Error (MAPE) [232], a standard metric for quantifying the average error magnitude produced by a model. MAPE is instrumental in understanding the typical deviation of the model's predictions. As demonstrated in Figure 9.2a, among the various models, the SARIMA model exhibited the highest accuracy, making it the preferred choice for `iRM` in its scheduling and decision-making processes. Even though the model does not accurately predict the exact power usage at a time, it is adequate for power corridor management infrastructure since the LP model (as seen in Section 5.1.1) deals with the power usage bounds. Further, as depicted in Figure 9.2a, the maximum and minimum of the actual time series for the same interval are identical to the upper and lower power usage during the predicted interval. The power-aware resource management system utilises the predicted maximum and minimum power values defined in Section 5.1.4.

9.2.2 Upper and Lower Power Corridor Enforcement

Figure 9.2b and 9.2c demonstrate the effect of dynamic resource reconfiguration of applications on the system-level power consumption. For the analysis, two applications (App 1 and App 2) were started with different initial node configurations. App 1 was launched with 12 nodes, and App 2 was launched with two nodes. The power corridor limit was set as 3000 for the lower limit and 4000 for the upper limit. As seen from Figure 9.2b, the upper power bound violation occurred with the applications' start. As a result, the runtime system decides to perform the resource reconfiguration of the running applications. During the first scheduler pass, the node allocation of App 1 was reduced to 3 nodes and the node allocation of App 2 was increased to 24 nodes. This change reduced the system level power usage and brought the system back in the power corridor, as seen in the graph at 300 seconds. The change in individual power usage of the application is also visible in the figure. The power consumption of App 1 is decreased, and App 2 is increased as an effect of the resource redistribution. As seen from Figure 9.2b, there were no further resource reconfigurations during the execution of applications since the forecast module predicted the power usage in the acceptable boundary during each scheduler pass.

Figure 9.2c demonstrates the lower power corridor enforcement using the power corridor management infrastructure. The same applications, App 1 and App 2, were used for the evaluation. Both applications were launched with four nodes, and the initial power corridor was set in the interval of 1500 and 2500 watts. Like the above scenario, the initial power corridor was violated during the application launch. As a result, the power corridor management runtime performs resource reconfiguration by performing a resource reduction on App 1 (four nodes to two nodes) and a resource expansion on App 2 (four nodes to 16 nodes). This transformation is observed at 500 seconds in Figure 9.2c and led to the increase in total power usage. The change in individual power usage of the application is also visible in the figure. The power consumption of App 1 is decreased, and App 2 is increased as an effect of the resource redistribution. As a result, the system is back in the required provided power corridor. As seen from Figure 9.2c, further resource reconfigurations were not needed since the system remained in the power corridor during the course of the application execution.

9.2.3 Dynamic Power Corridor Enforcement

The flexibility of the system to adapt to the change in power limits of the power corridors is demonstrated in Figure 9.2d. It used the dynamic power corridor enforcement techniques described in Section 5.1. To showcase the dynamism, the initial power corridor was set in the range of 100 and 600 watts. Then, a single malleable application (Invasive Pi calculation application in Subsection ref) is launched with a single node allocation. The figure shows that the system already adheres to the power corridor limit, and no violation is recorded. However, after 240 seconds, the power corridor limit was changed (increased to 700 and 1200 watts) to evaluate the capability of the system to adapt to changes in power corridors. As seen in 250 seconds, the total power usage of the system was increased to adhere to the power usage limits by the power corridor enforcement infrastructure. For that, the node allocation of the application was increased to nine nodes. Again, the power corridors were modified (increased to 1300 and 1900 watts), and the total power usage was brought back to the power corridor after resource redistribution. At around 700 seconds, the power corridor limit was reduced to the initial budget of 100 and 600 watts. As seen in Figure 9.2d, the total power usage of the system is again back in the accepted budget. However, the interesting aspect is the change in the total power usage value of the system for the same power corridor limits (100 and 600) during the application's beginning and end. It can be attributed to the node allocation performed as part of the system reconfiguration in the last scheduler pass. For the last power corridor change, the power corridor infrastructure has given four nodes to the system instead of a single node, which was the initial node configuration.

9.2.4 Power-aware Scheduling Strategy

Two iMPI applications were utilised to evaluate the power-aware scheduling strategy proposed in this work. The first application (App 1) is the PI calculation, and the second (App 2) solves the 2D heat equation using the Jacobi iteration. The applications are submitted as a total of 20 jobs with equal distribution among both applications. The node configuration (`--nodes`) for the applications was in the range of one to four. The average power usage per node of App 1 is 170 watts, and App 2 is 250 watts. The 20 jobs are submitted to the power-aware `ABS` with a fixed inter-arrival time of two seconds. The minimum (`--min-node-invasic`) and maximum node counts (`--max-node-invasic`) for all submitted jobs were given one and 14, respectively.

Three strategies were considered to compare and evaluate the effectiveness of the proposed power-aware scheduler. In the first strategy (scenario 1), the static `backfill` scheduler in Slurm with no redistribution of resources is considered. In the second strategy (scenario 2), the power corridor management runtime demonstrated in the above sections (9.2.1 - 9.2.3) are considered. As seen in the above sections, only running malleable applications were considered to bring back the system in the power corridor. However, in strategy three (scenario 3), the power-aware scheduler is considered where the applications in the job queue are also considered during the resource reconfiguration phase (See 4). In scenario 2, no jobs are held in the queue and launched if the requested resources are found, unlike scenario 3.

Nevertheless, the job submission order is identical in the above three scenarios. However, for scenarios 2 and 3, all jobs are considered malleable. Additionally, the power corridor values are changed dynamically in fixed intervals to evaluate the adaptive capability of the system. The initial upper and lower bound is set as 1700 and 2500 Watts, respectively. Later, the lower and upper power boundaries are brought down to 1000 and 1700 watts. After that, it is increased to 2500 and 3500 watts for lower and upper corridors. This dynamic change in corridors can be observed in Figure 9.3. To determine the lowest possible value for the system's lower power bound, the power consumption of idle nodes is calculated. On average, an idle compute node consumes 71 Watts. To establish the maximum value for the upper power bound, it's

Table 9.2: Comparison of metric values for different scheduling scenarios [55].

Scenario	Power corridor violations	Makespan (mins)
1	6	13.75
2	2	14.25
3	0	14.00

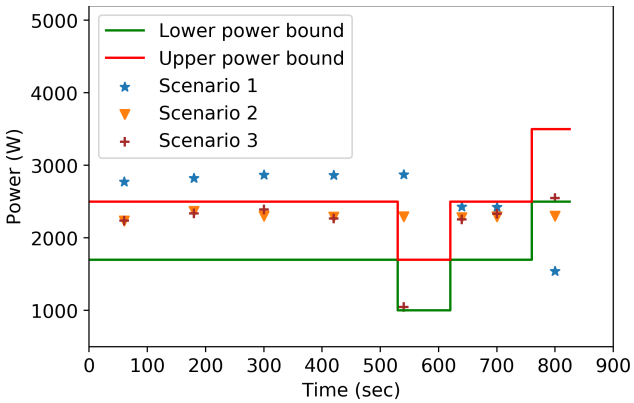


Figure 9.3: Average system power usage and power corridor violations for different scheduling scenarios [55].

assumed that the heat simulation is running across all 14 nodes. The heat simulation (App 2) was the most power-hungry application (250 watts per node) in the evaluation.

The results of the experiments using the three scenarios mentioned earlier are depicted in Table 9.2 and figure 9.3. Table 9.2 summarises the power corridor violations that happened in the three scenarios, along with the makespan for the 20 jobs. The number of power corridor violations denotes whether the system could return to the desired power corridor using dynamic resource management. As expected, the static backfill scheduler strategy wasn't able to recover from the power corridor violations due to the lack of flexibility to change the resources. However, scenarios 2 and 3, with their malleability support, were able to finish the execution of jobs with fewer power corridor violations than the backfill strategy. As seen in the makespan column of the table, scenario 1 has a better makespan because no resource reconfiguration was performed to adjust for the power corridor violations. This gave the scheduler the flexibility to schedule the jobs from the queue as efficiently as possible. The comparison between the power corridor violations in scenarios 2 and 3 can be better understood from Figure 9.3.

For the initial power corridor limits, the applications in scenarios 2 and 3 adhered to the power corridor through the resource reconfigurations performed by the power corridor management infrastructure. As described above, when the power corridor was decreased dynamically to 1000 and 1700 watts, only scenario 3 was able to maintain the power corridor. This was possible due to the flexibility to pick the desired job from the queue that satisfies the power requirements (or by employing the moldability to enforce the power requirements). As a result, the power-aware scheduler in scenario 3 generated a new viable resource configuration using the LP model by considering the running and waiting jobs. However, the LP model could not find a feasible resource configuration that brings the system back into the power corridor only using the running applications. This can be observed at the 550 seconds in figure 9.3.

Similarly, scenario 3 was able to adjust to the change in power corridor (the lower limit was increased to watts 2500 and the upper limit was increased to 3500 watts), triggered around 800 seconds (See figure 9.3). A feasible resource configuration was not obtained for scenario 2 to bring the system back into the power corridor. In the power-aware scheduler implementation, the LP model takes the job in the waiting queue as

an argument in contrast to scenario 2. In the analysis, the total number of jobs picked from the queue and given to the LP model as input was either one or two. Nevertheless, the feasible solution was found with a single waiting job in all the experiments performed as part of the evaluation. The average overhead for obtaining resource configuration from the LP model was 20 milliseconds for the proposed implementation. The proposed LP model also supports more number of waiting jobs as input, which can lead to higher overhead.

9.3 iCheck – Performance Analysis

The experimental evaluation of the iCheck system was performed on 16 compute nodes in SuperMUC-NG. The iCheck system was launched on four compute nodes, giving the applications up to twelve nodes. The node count associated with each application changed based on the experiment types. As seen in Section 8.2.3, four applications were used to analyse the performance and various design aspects of iCheck. To evaluate iCheck, the ls1 mardyn application was initially deployed on a scale ranging from one node (48 processes) to twelve nodes (576 processes). The initial step in the evaluation involved analysing iCheck's performance in comparison to the internal MPI-IO based checkpointing mechanism within ls1 mardyn, as detailed in Subsection 9.3.1.Then, the performance variations in blocking and non-blocking implementation of iCheck, as well as push and pull techniques employed inside agents, are analysed (Subsection 9.3.2 and Subsection 9.3.3) using ls1 mardyn. For these data transfer tests, the ls1 mardyn application was run for 100 time steps with a checkpoint interval of 10 time steps. This was performed to stress test the system. Finally, the overhead associated with iCheck is analysed using ls1 mardyn (Subsection 9.3.4). For overhead analysis, checkpointing (using iCheck and MPI-IO) was performed in every 1000 timesteps of the ls1 mardyn run. The application was run for a total of 10,000 time steps, and the simulation system was initialised with 65536 molecules for all the tests performed. During every checkpoint operation, a checkpoint of size eight megabytes was created by the application. The checkpoint contained information regarding the molecules (For example, molecule id, position, momentum, and velocity). iCheck was evaluated against MPI-IO and the state-of-the-art Scalable Checkpoint/Restart (SCR) library using a synthetic application, as detailed in Subsection 9.3.5. The application was executed on a range of one to twelve nodes, producing a checkpoint of up to 2.3 GB for each checkpoint operation.

9.3.1 iCheck vs MPI-IO

In this subsection, the efficiency of iCheck's blocking checkpointing approach is analysed with the native MPI-IO-based checkpointing utilised in ls1 mardyn. Figure 9.4a presents the average time required for checkpoint transfers in iCheck and MPI-IO across various agent counts displayed on a logarithmic scale. The agent count varied from one to twelve. Throughout these tests, the application consistently ran on twelve nodes. Additionally, Figure 9.4b illustrates the average duration of checkpoint operations for both iCheck and MPI-IO when the application is executed on different node configurations—specifically, one, four, eight, and twelve nodes. In this part of the analysis, iCheck operates with a single agent to maintain consistency across the different node setups.

As seen in Figure 9.4a and Figure 9.4b, iCheck performs better than the in-house checkpointing system in ls1 mardyn for all the agent/node configurations used in the analysis. Figure 9.4a illustrates that in iCheck, the checkpoint transfer process accelerates as the number of agents increases, demonstrating the system's scalability and efficiency in handling parallel operations. The worst-performing combination of a single agent performs checkpoint operations 400 times faster than the MPI-IO based checkpointing. The best

performance results were obtained while using twelve iCheck agents. The figure shows that the twelve-agent configuration performed checkpointing 5,000 times faster than MPI-IO based checkpointing.

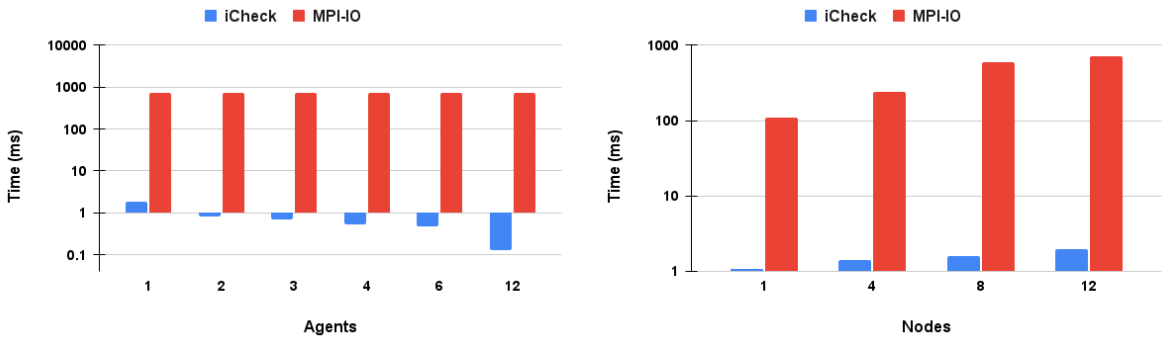
Figure 9.4b also depicts a similar trend. iCheck outperforms the in-house checkpointing system with a single agent for different node configurations. iCheck only took .8 to 1 milliseconds for transferring checkpoints, while the MPI-IO took around 100 to 700 milliseconds for transferring the same amount of checkpoints. As the figure shows, iCheck demonstrated a performance improvement of over 125 to 700 times. Additionally, it can be observed that when there are no agent changes, checkpoint transfer time in iCheck increases as the number of nodes of the application increases. It can be attributed to the fact that the same number of agents need to perform more checkpoint transfers as the number of nodes increases. The agent change in these experiments was performed statically (i.e., the application had the same number of agents from beginning to end). The agent count was manually set during the application launch using the configuration file for experimental purposes.

9.3.2 Blocking vs Non-Blocking Checkpointing

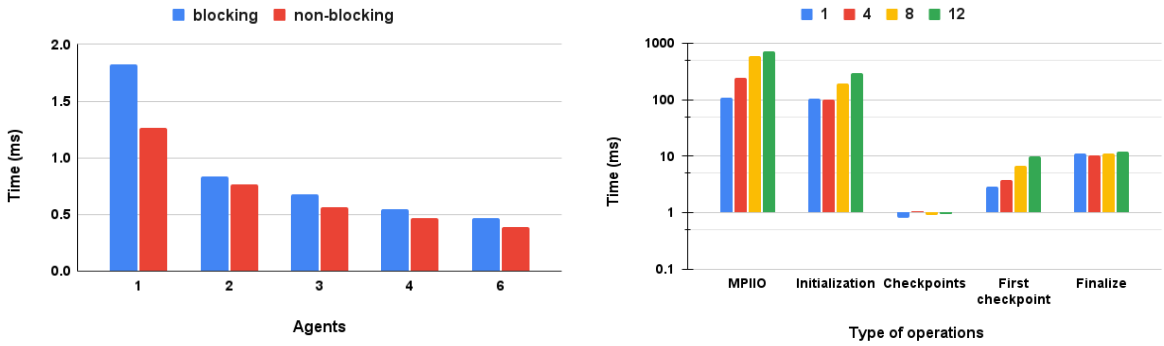
Figure 9.4c depicts the contrast in checkpoint performance of ls1mardyn while using the blocking and non-blocking checkpointing techniques provided by iCheck. It displays the time taken to write the checkpoint for different agent configurations in both checkpointing techniques. As seen in the figure, the non-blocking checkpointing technique of iCheck provides a better checkpoint transfer rate than the blocking version. This trend is visible in all the agent configurations evaluated. The non-blocking mode provides up to 50% improvement over the blocking mode. An interesting observation from the figure is that the non-blocking version of iCheck offers performance comparable to that of the blocking version but with the added efficiency as if an extra agent was involved in the operation. Another insight from the graph, similar to the previous analysis, is that the checkpoint transfer becomes faster as the agent number increases. This trend holds for both blocking and non-blocking techniques. Section 6.2.4 provides information about the implementation aspects of blocking and non-blocking techniques. The agent count change in this experiment was performed manually during the application launch using the configuration file.

9.3.3 Push vs Pull Strategies

Figure 9.5 compares the average time taken for checkpoint transfer using push and pull strategies in ls1 mardyn. Different agent configurations were also used in this analysis. The implementational aspects of push and pull techniques are described in detail in Section 6.2. It can be observed from the graph that the push strategy performs better than the pull strategy for all agent configurations. In ls1 mardyn, each process has a different number of molecules, which changes during the application execution. Hence, the agents must read the new checkpoint size before transferring the checkpoint, which induces an additional overhead in the pull strategy. With a single agent, the push strategy is 40% faster than the pull strategy, and this can be attributed to the fact that a single agent needs to read the checkpoint size before transferring the checkpoint from all the application nodes. However, as the number of agents increases, the advantage provided by the push strategy is barely visible. As the agent number increases, the checkpoint size information to be read from application nodes is reduced in the pull strategy (agents can read the checkpoint size information from application nodes in parallel). Hence, it can be concluded that the push strategy is ideal for single-agent applications. The agent count change in this experiment was performed manually during the application launch using the configuration file. Since iCheck supports non-MPI applications, this strategy can also be set as the default option for non-MPI applications.



(a) Comparing iCheck and MPI-IO with different agents **(b)** Comparing iCheck and MPI-IO with different nodes



(c) Comparing blocking vs non-blocking strategies in iCheck **(d)** Breakdown of iCheck induced overhead using a single agent

Figure 9.4: Checkpointing analysis on ls1 mardyn using iCheck and MPI-IO [77].

Table 9.3: Effect of checkpointing on execution time in ls1 mardyn [77].

Time	MPI-IO	iCheck
1775.5s	104.383s	0.567538s

9.3.4 iCheck Overhead Analysis

This subsection delves into the overhead that the iCheck library introduces to the ls1 mardyn application. As mentioned earlier, ls1 mardyn performs a checkpoint of application data at every 1000th time step and is executed for a total of 10,000 time steps. The induced overhead by iCheck is represented in Table 9.3 and Figure 9.4d. While Table 9.3 compares the overall impact of iCheck and MPI-IO on the ls1 mardyn application, Figure 9.4d breaks down the overhead for each specific API call made by iCheck in the ls1 mardyn context.

For a comprehensive analysis of the overall checkpointing impact on ls1 mardyn, the application was executed on 12 nodes, comparing the performance of iCheck and MPI-IO. This analysis was conducted using a single agent in the iCheck setup. As seen in the above sections, a single iCheck agent takes maximum time for checkpoint transfer and is used for comparison with the MPI-IO to ensure the fairness of the overhead analysis. It can be inferred from Table 9.3 that iCheck only takes .03% of the overall execution time of the application while MPI-IO takes around 5.6% of the overall application time. A major portion of the overhead associated with the iCheck system can be attributed to the overhead associated with the initialisation

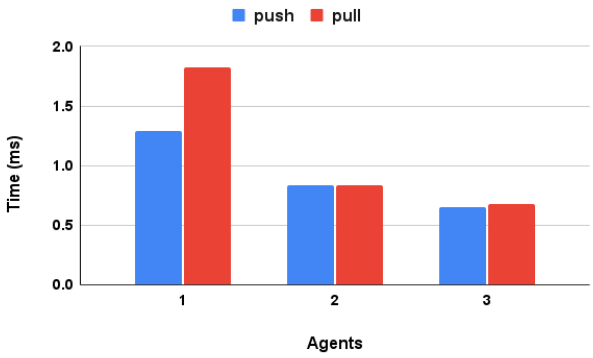


Figure 9.5: Comparing checkpoint performance in Push and Pull Techniques.

of the agents. This is clearly visualised in Figure 9.4d. Nevertheless, it can be seen that the performance of the iCheck library increases as the application progresses due to the RDMA-based checkpoint transfer happening inside the icheck commit operation. Using iCheck resulted in 6% faster execution of the application than the MPI-IO based checkpointing. Additionally, the .03% of overhead on the application can be further reduced by using more agents.

To assess the overhead introduced by iCheck during its initial setup, the ls1 mardyn application underwent testing on one, four, eight, and twelve nodes. Figure 9.4d, which is displayed on a logarithmic scale, scrutinizes the overhead for each iCheck operation in relation to the checkpoint transfer time observed with MPI-IO. For this comparison, iCheck was configured to use a single agent, which, according to Figure 9.4a, represents the least optimal performance scenario within iCheck, thereby providing a fair basis for comparison against MPI-IO. The trend depicted in the graph indicates that as the number of nodes increases, there is a corresponding increase in the time required for both checkpoint transfer and the initial setup processes. This is also true for MPI-IO and is observable in the figure. There is no initialisation and finalisation overhead associated with MPI-IO since it only uses a single API call to perform the checkpoint transfer operation. Hence, there is only one entry associated with MPI-IO in the graph. However, as defined in Section 6.1.3.3, iCheck has API calls for configuration and checkpoint transfer, and Figure 9.4d shows the overhead associated with these operations. One interesting observation is the transfer time associated with the first checkpoint transfer operation and the remaining operations. The first checkpoint transfer takes up to five to ten times more time than the remaining one. This is attributed to the memory registration overhead associated with libfabric. Nevertheless, this happens only once (or after agent change triggered by probe operation), and as the application progresses, the overhead associated with this becomes negligible.

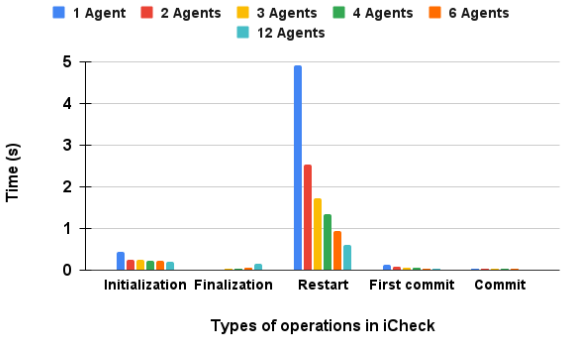
Another significant insight from the figure is that the overall time for initialisation and first checkpoint transfer in iCheck is comparable to that of a single checkpoint transfer using MPI-IO. As seen in the figure, iCheck outperforms MPI-IO from the second checkpoint transfer. However, it is noteworthy that deploying additional agents can significantly decrease the configuration time in iCheck.

9.3.5 Comparing iCheck, SCR and MPI-IO

Porting a large application, such as ls1 mardyn, to state-of-the-art Scalable Checkpoint/Restart [188] can be a challenging task. The quality of the porting process can significantly affect the application's performance. To ensure a fair comparison, a synthetic benchmark application is created and ported with SCR. Using a synthetic application, with its ability to configure checkpoint sizes, enables the comparison of various

checkpoint sizes, a feature not feasible with ls1 mardyn. Consequently, a performance analysis of iCheck compared to MPI-IO and SCR is conducted using this synthetic application. Furthermore, this setup allows for examining the overhead iCheck introduces during large data transfers. In this analysis, the application processes a 2.3 GB data checkpoint across 12 compute nodes.

(a) Comparing iCheck, MPI-IO and SCR on checkpointing large data size



(b) Overhead analysis of various iCheck operations

Figure 9.6: Checkpointing analysis on synthetic application using iCheck and MPIIO [77].

Figure 9.6a shows the average time taken for checkpoint transfers in iCheck, MPI-IO, and SCR. The figure reveals that iCheck, even with just a single agent, conducts checkpoint transfers 3.9 times faster than SCR and 7.6 times faster than MPI-IO. Additionally, iCheck's performance improves progressively as more agents are introduced to facilitate the checkpoint transfer, a trend that is consistent across all experiments discussed in the preceding sections. For the 12-agent configuration, iCheck exhibits a performance improvement of 57 times over SCR and 100 times over MPI-IO. The plotted checkpoint transfer times of iCheck, MPI-IO and SCR are averaged over five runs. The performance improvement with iCheck is attributed to the usage of RDMA operations to transfer the checkpoint into the in-memory of a remote compute node. Meanwhile, SCR and MPI-IO write the checkpoint data directly into the parallel file system.

Figure 9.6b shows the iCheck library overhead associated with iCheck operations for large checkpoint transfers in a synthetic application. It shows the time taken for iCheck initialisation, checkpoint data transfer, finalisation, and the time to restore the checkpointed data into the application after restart. Measurements were taken using iCheck with varying agent configurations (one to twelve). This analysis is similar to 9.3.4, and the overheads graph exhibit a similar trend (See Figure 9.4d). The figure shows that an increase in agents can reduce overhead induced by the library on the application. A particular point of interest in the figure is the time taken to perform the restart operation. Restart operation refers to application processes connecting to the agents, registering the RDMA memory, transferring the checkpointed data from the iCheck system back to the application and copying it into the relevant data structures. The depicted performance variation between the commit and restart operations is noteworthy since both operations employ remote memory access (RDMA read and write) to transfer the checkpoint. In the commit operation, iCheck performs an intra-node synchronisation among the processes before performing the checkpoint transfer to ensure the consistency of the checkpoint. However, in the restart operation, inter-node synchronisation is performed by the iCheck after receiving the data from the agents. This synchronisation is needed for providing data distribution services to malleable applications. Although the time taken for the actual data transfer is similar to the commit operation, this explicit synchronisation among all participating processes considerably increases the checkpoint restart time in iCheck. Nevertheless, unlike the commit operation, the restart operation is only performed once during the application's lifecycle. However, increasing the number of agents can significantly improve the performance of the restart operation. As seen in Figure 9.6b, the time taken

for restart improved up to ten times as the number of agents increased from one to twelve.

9.4 Resource Management in iCheck

This section demonstrates the resource management capability of iCheck. First, the iCheck system's adaptive nature is analysed concerning a non-malleable application (Subsection 9.4.1.1). Second, the ability of iCheck to support checkpointing in malleable applications is discussed (Subsection 9.4.2). Finally, a simple use case of data distribution API from iCheck is demonstrated (Subsection 9.4.3).

9.4.1 Adaptive Resource Management in iCheck

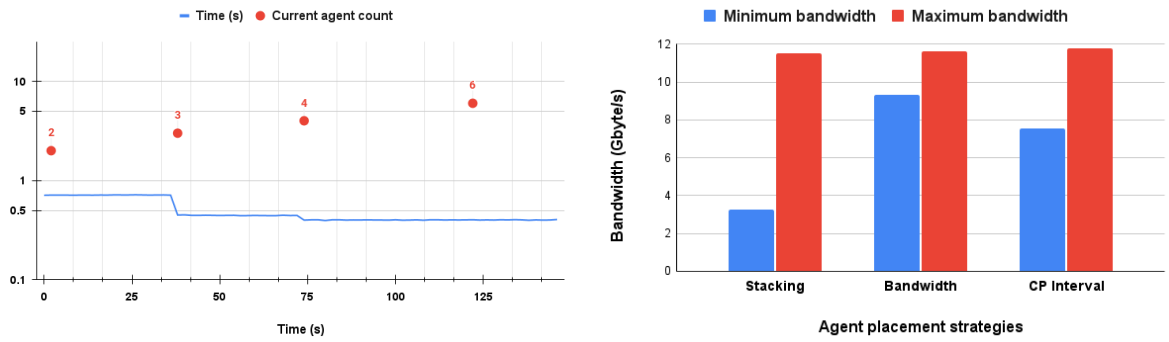
The heat simulation and LULESH were used to demonstrate the malleability aspect of iCheck. Heat simulation application was used to analyse the impact of dynamic agents on running applications (Subsection 9.4.1.1). It was also used along with LULESH to demonstrate the horizontal scaling characteristics of the iCheck system (Subsection 9.4.1.3). Both applications were run on two to twelve nodes for evaluation. The synthetic application was used to perform the effect of agent placement on running applications (Subsection 9.4.1.2). In addition, the synthetic application was also used to analyse the performance and overhead associated with the alternative checkpoint mechanism enabled when the iCheck library fails to connect to the iCheck core system (Subsection 9.4.1.4).

9.4.1.1 Impact of Dynamic Agents

It can be concluded from the results of the above analysis that the checkpointing performance of iCheck is proportional to the number of agents employed. Significant performance improvement occurs if the number of agents is changed from single to the number of nodes in an application. For example, in synthetic application analysis, a performance improvement of 50 times was observed, while in ls1 mardyn, a performance improvement of 400 times was observed when agents were changed from one to twelve (number of nodes).

The above experimental analyses restarted the applications to utilise the new agent configuration. However, in this section, the agent change is performed dynamically to analyse the performance of the iCheck library on an application. The `icheck_probe_agents()` call made by the application will trigger the dynamic agent reconfiguration based on the agent change decision made by the controller. As described earlier, a 2D heat simulation application that checkpoints the data of size 1.4 GB is used to perform the experiments. The application was run on twelve nodes, and agents were changed on the fly by the controller based on a predefined order.

Figure 9.7a demonstrates the impact of dynamic agents on the heat simulation application. Initially, the application was allocated two iCheck agents, and it took around 0.7 seconds to transfer the checkpoint from the application to the agents. After forty seconds, a dip in the checkpoint transfer time can be observed in the graph. This dip (or faster checkpoint transfer) can be attributed to the agent change (from two to three agents) triggered by the iCheck controller. At around 80 seconds, the data transfer rate was almost doubled compared to the initial rate. This performance improvement is also the consequence of the agent change (from three to four agents) performed by the controller. However, the controller again performed an agent change at around 120 seconds, and there was no improvement in performance because the network bandwidth was utilised effectively by the iCheck agents and the overhead associated with checkpoint transfer masks the benefits provided by the new agents.



(a) Impact of dynamic agent adaptation on a 2D heat simulation

(b) Effect of agent placement strategies on synthetic application

Figure 9.7: Effect of dynamic agents and agent placement strategies [77].

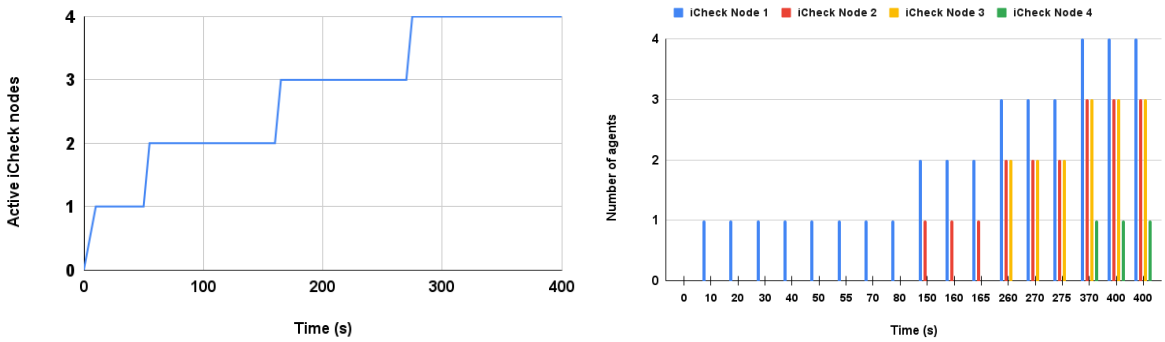
As described in Section 6.4.2, `icheck_probe_agents()` call inside the application contacts the controller to obtain the information about the new agent configuration (if available). The overhead associated with a successful probe call in the experiment was around 2.5 seconds. It can be translated as the time for performing four iCheck commit operations (as depicted in the figure). A successful probe call denotes that the new agent information is obtained, and the application has successfully connected to the new agents and is ready for checkpoint transfer. An unsuccessful probe (or no agent change) costs the time needed for two commit operations (or 0.002 seconds). This overhead is minuscule compared to the performance improvement provided by the new agents for every checkpoint transfer. As more and more commits are performed with the new agents, the overhead associated with the agent change is nullified. However, modelling and application-based heuristics are needed to determine the ideal agent change that can immediately give the maximum performance to the application instead of a step-by-step agent increase, as observed in the figure.

9.4.1.2 Effect of Agent Placement Strategies

The previous subsections primarily examined how different numbers of agents impact various applications. This subsection shifts focus to investigate the effects of positioning agents on iCheck nodes, utilising the different strategies outlined in Section 7.1.2. In this experimental analysis, a synthetic application (App A) is run on two nodes (96 cores) in an infinite loop. A single agent was allocated to the application for checkpoint transfer operation. After that, five synthetic applications were run on ten compute nodes (two nodes per application). Each of the applications was given a single iCheck agent for checkpoint transfer. As described in Algorithm 6, agents can be placed on the iCheck nodes based on different strategies. In the experiments, the agents corresponding to these five applications were placed on iCheck nodes based on the ensuing three strategies and their effect on App A's checkpoint transfer rate is studied. A total of four iCheck nodes were utilised in this analysis.

The three strategies used for placing the agents are:

1. Stacking: In this strategy, all the agents were allocated in the same iCheck node as App A. All agents are stacked together on the same iCheck node.
2. Bandwidth: In this strategy, the iCheck nodes with the least activity are identified, and the agents are placed there (This ensures higher bandwidth for data transfer).
3. CP interval: In this strategy, the iCheck nodes where the checkpoint interval is higher are identified, and the agents are placed there.



(a) Number of active iCheck nodes in the system

(b) Distribution of agents across iCheck nodes

Figure 9.8: Demonstrating the horizontal scaling in iCheck during a time interval [77].

It is evident from Figure 9.7b that these agent placement strategies significantly impact the application's performance. The fluctuation between the checkpoint transfer bandwidth for App A is higher in the stacking strategy. The figure shows that the checkpoint transfer rate fluctuated between 3GB/s to 11.5GB/s when all the agents were allocated on the single iCheck node. This is in contrast to the other two strategies considered for analysis. In these strategies, the agents are placed among two different iCheck nodes. As depicted in the figure, the bandwidth strategy has the least fluctuations because the agents are placed across iCheck nodes with less activity, and the agent can avail the total bandwidth to perform the checkpoint transfer. Hence, there is less inference with App A's checkpoint transfer process. As a result, the fluctuation was minimal in this strategy (App A obtained an aggregate bandwidth of 9.5GB/s to 11.5 GB/s).

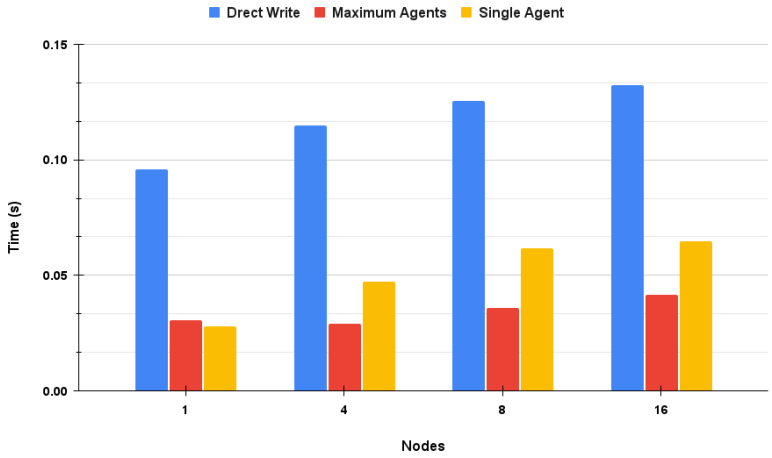
9.4.1.3 Adding Nodes to iCheck

iCheck is entirely malleable and can expand or reduce its iCheck nodes based on the availability of compute resources. This system-level malleability aspect of the iCheck system is demonstrated with the use of predefined resource expansion. That is, the nodes are given to iCheck dynamically during the course of the evaluation. Four iCheck nodes were used for this experiment. Initially, the iCheck system is started with a single node, and the remaining iCheck nodes are given individually to the iCheck system during predetermined intervals. After the nodes were given to iCheck, an iCheck-aware application was launched in a pre-defined manner to verify whether the iCheck controller had allocated the agents in the newly issued iCheck nodes. The event log information from the controller and nodes was analysed to confirm the addition of new nodes and agent launch in those nodes. The best agent placement strategy in the above section 9.4.1.2 was employed for this analysis.

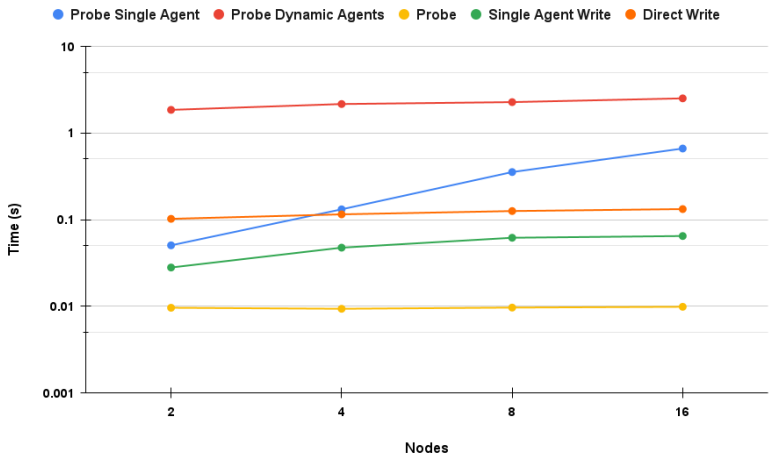
This analysis utilised ls1 mardyn, LULESH, the heat simulation and the synthetic application. Initially, the ls1 mardyn application was launched on two compute nodes, followed by LULESH on two nodes after a predetermined interval. Later, the heat simulation application is launched on four nodes, and the synthetic application is launched on the remaining four nodes at the end. The number of agents for each application was predetermined. The ls1 mardyn got one agent, LULESH got two agents, and heat distribution and synthetic applications got four agents each. The node number and agent number were chosen randomly.

The results of this experiment are depicted in Figure 9.8a and Figure 9.8b. Figure 9.8a plots the number of active nodes during the course of the application execution. The second figure provides information about the number of agents placed in an iCheck node during the period of the experiment. Four iCheck nodes are used for this analysis, and each iCheck node is labelled based on the order in which it is added to the

system. iCheck node 1 is the first node added to the system, and iCheck node 4 is the last node added to the system. The change in the iCheck nodes, as shown in Figure 9.8a, highlights the iCheck system's ability to dynamically adjust its resources (nodes) in real time. Furthermore, from Figure 9.8b, it is evident that iCheck can leverage these additional iCheck nodes to allocate agents for new incoming applications. Initially, the ls1 mardyn application was started with a single agent. The iCheck controller then allocated this agent to the sole available node at the time (refer to iCheck node 1 in Figure 9.8b). Subsequently, at approximately the 40-second mark, an additional node (iCheck node 2) was introduced to the system, as depicted in Figure 9.8a. After the successful node addition, when LULESH was connected, the iCheck controller distributed the two agents associated with it equally among the iCheck node one and two. It can be observed in Figure 9.8b (See 150th second). Similarly, after all of the iCheck nodes were added and all of the applications were launched, the agents were placed across all the iCheck nodes in a balanced manner (See 270 seconds and 400 seconds in Figure 9.8b) based on the employed bandwidth based agent placement strategy.



(a) Checkpoint performance without utilising iCheck core system



(b) Overhead analysis of iCheck probe functionality in the event of an agent change

Figure 9.9: Analysis of fallback mechanism for checkpoint transfer in iCheck

9.4.1.4 Fault Tolerance in iCheck

As mentioned in Section 6.5, upon detecting a failure to reach iCheck components, the iCheck library will start writing the checkpoints directly into the parallel file system instead of leveraging the RDMA-based checkpointing offered by agents. This incurs significant overhead since the lead rank inside each node will be writing the checkpoints to make a consistent format for storing the checkpoints. One of the advantages of such an approach is utilising the agents to read the checkpoints into the system in case of a restart. This makes it portable, but it comes with a significant overhead and is visualised in Figure 9.9a.

Figure 9.9a compares the time taken for the `icheck_commit` operation performed directly by the icheck-library on the compute node of the application (labelled as *Direct Write* in the plot) and RDMA-based checkpointing (depicted for both Maximum Agent and Single Agent configurations) where remote agents transfer the checkpoints from the compute nodes to the iCheck nodes. *Direct Write* in the plot refers to the checkpoint transfer performed directly by the library into PFS using POSIX [233] file write operation. For the RDMA-based checkpointing (i.e, normal iCheck operational mode), the time taken using the best agent configuration (labelled as *Maximum Agents* where agent count is equal to the number of nodes) and worst agent configuration (labelled as *Single Agent*) is plotted. The figure depicts the time for running the synthetic application on one, four, eight and sixteen nodes. As seen in the figure, there is a considerable performance difference between library-based and RDMA-based checkpointing. This difference increases as the number of nodes and checkpoint sizes increase. Checkpoint size is increased by $4x$, $8x$ and $16x$ for nodes four, eight, and sixteen compared to a single node (Single node application checkpointed a size of approximately half GB). However, the increase in transfer time was not linear. It was expected that since much of the time was taken for configuration (file open, file close, data preparation) and in addition, the file write does happen in parallel (with one process per node writes the checkpoint), there would be similar write time for applications with different node counts.

As explained in Section 6.5, the iCheck library will try contacting the controller based on the predefined frequency upon a failure. The controller is contacted via the probe operation to get information about new agents. As defined in section 6.4.2, the probe operation has the following three outcomes.

- no agent change (referred as **outcome 1**)
- agent count change (referred as **outcome 2**) and,
- agent shift (iCheck node change or new agent in the same iCheck node - referred as **outcome 3**).

These outcomes might happen while the iCheck library continuously probes for agents while checkpointing is performed on library-based mode. The overhead for such probes can accumulate and needs to be analysed carefully in association with failure recovery. This is visualised in Figure 9.9b. It depicts the time taken for the above three scenarios and compares it with the time taken for RDMA-based checkpointing (labelled as *Single Agent Write* in the plot) and Library Based checkpointing (labelled as *Direct Write* in the plot). Some interesting insights can be derived from this plot. Firstly, the time taken for a simple probe in **outcome 1** (labelled as *Probe* in the plot) is negligible in comparison with probe calls made in **outcome 2** (labelled as *Probe Dynamic Agents* in the plot) and **outcome 3** (labelled as *Probe Single Agent* in the plot). The time taken for **outcome 3** is increasing steadily since the application nodes need to connect to a single agent, and the overhead is increased as the number of nodes is increased. Hence, the maximum time is taken when the application runs on 16 nodes since all nodes need information about their agent. In addition, an interesting observation can be made about **outcome 2**; even though the time taken for the probe remains steady, it is much higher than the direct write mode and **outcome 3**. In this scenario, when an application probes the controller, the controller assigns a new set of agents (here, the number of agents is equal to the number of nodes). This new set of agents must be launched in the iCheck nodes before passing the agent data to the

application, resulting in an additional overhead compared to **outcome 3**, where a single agent only needs to be launched. By analysing this plot, it can be concluded that after an agent failure, an application going back to RDMA-based checkpointing is beneficial only if sufficient commit/checkpoint operations remain to gain performance improvement from the overhead incurred by associated probe operations.

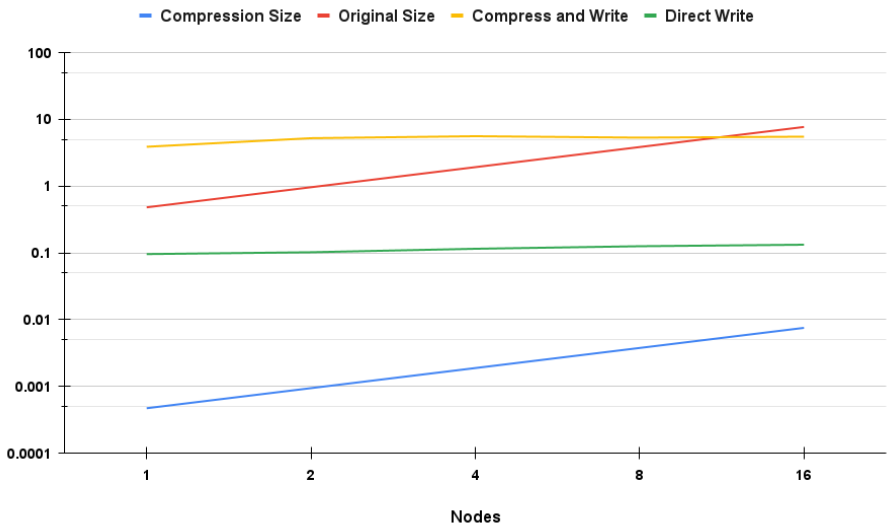


Figure 9.10: Data compression inside iCheck

9.4.1.5 Checkpoint Compression in iCheck

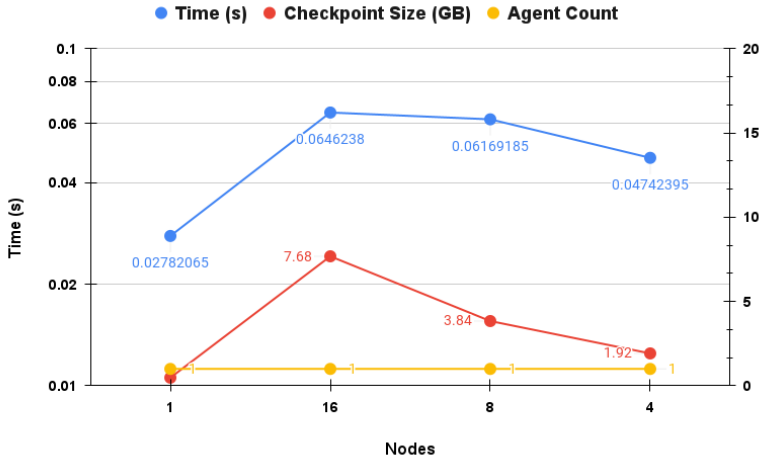
This subsection demonstrates the checkpoint compression feature of iCheck introduced in section 6.2.5. The compression can be activated either inside the agent (while writing to PFS) or in the application library (while writing the checkpoint directly due to the iCheck system failure). Figure 9.10 gives an overview of the effect of compression on checkpoint size and transfer time while the application is running on different numbers of nodes. The synthetic application was checkpointing one million floats per process and was run on one, four, eight and sixteen nodes for the evaluation. In the plot, the compression and original sizes are represented in Gigabytes, and the time for *Compress and Write* and *Direct Write* are defined in seconds. As observed in the figure, the compression inside iCheck significantly reduced the checkpoint size to the original size; however, the time taken to write directly into the PFS increased heavily since the checkpoint compression took a significant amount of time. The magnitude of compression overhead can easily be understood by analysing the time difference between *Direct Write* and *Compress and Write* in the plot. Hence, it can be derived that the compression is only relevant on the agent side while writing into the PFS since the agent can effectively utilise its idle time to compress the checkpoint. As a result, the overhead won't be reflected in the application. Hence, iCheck will only utilise the checkpoint compression on the agent side.

9.4.2 Malleable Application and Agents

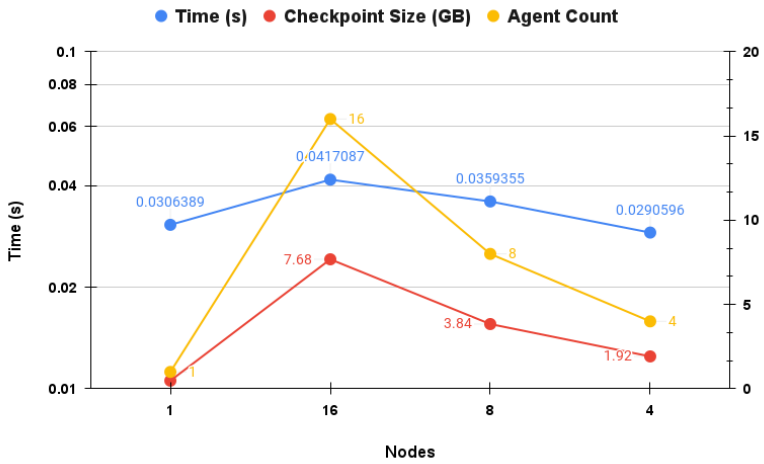
The experimental evaluation of the malleable characteristic of the iCheck system was performed on up to 16 compute nodes in SuperMUC-NG. A synthetic malleable benchmark application (Section 8.2.3.2) was

used to analyse the performance of the system. The application resources were expanded and reduced dynamically using iRM, and the adaptivity of iCheck with regard to the resource changes was analysed. The application was run on up to 16 nodes, and the iCheck system was run on two nodes. Additionally, the performance evaluation and overhead analysis of pipelining support in iCheck was also studied using the benchmark.

In this section, the performance evaluation associated with malleable applications is depicted. The effect of different iCheck strategies (MA-RA and MA-MA in Section 8.2.3.2) on malleable applications was analysed in detail. For this analysis, the synthetic application was initially run on a single node, and then iRM was leveraged to expand the resources to 16 nodes. Later, the number of resources was halved using iRM. In addition, the checkpoint size was also considered for the analyses. In the first set of experiments, the checkpoint size increased as the number of nodes increased and reduced as the number of nodes were reduced. In the second set of experiments, the checkpoint size remained constant for malleable applications. In the synthetic benchmark, each process was tasked to checkpoint 100,000 floats every five seconds.



(a) Impact of iCheck on a malleable MPI application with no agent change



(b) Impact of iCheck on a malleable MPI application with agent change

Figure 9.11: Effect of dynamic agents on a malleable MPI application with dynamic checkpoint size

9.4.2.1 Malleable Application with Increasing Checkpoint Size

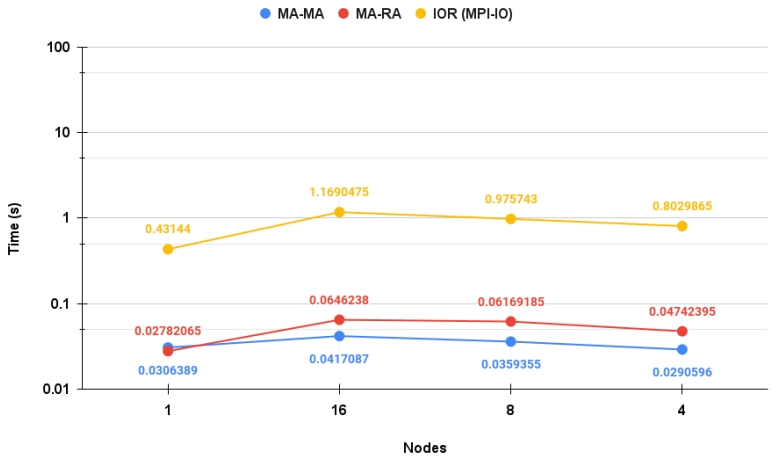
Using Malleable Application Rigid Agent (MA - RA) Strategy: Figure 9.6a shows that iCheck with a single agent performs considerably slower than multiple agents. Hence, a single agent was used for this evaluation. Figure 9.11a depicts the results of the MA-RA strategy on a malleable application. The time for transfer (in seconds) is mapped into the left vertical axis, and the checkpoint size (GB) and the agent count are mapped to the right vertical axis (This also holds for Figure 9.11b). The figure shows that the application was initially started with a single node and expanded to 16 nodes. During the following resource change, the number of nodes was brought down to eight and eventually to four. It can also seen from the figure that the number of agents remains the same throughout the application execution. The initial checkpoint size is 0.48 GB for the application. As the number of nodes increased, the checkpoint size was increased to 7.68 GB and then later reduced to 3.48 and 1.92 GB, respectively. Similarly, it can be observed that the checkpoint time was doubled as the number of nodes was increased. This can be attributed to the change in checkpoint size. However, it is noteworthy that there was not a considerable reduction in checkpoint time as the number of nodes was reduced to eight from sixteen. Even though the checkpoint size was halved, the performance of iCheck almost remained the same. Nevertheless, there was performance improvement as nodes were reduced again. The performance improvement was around 36%.

Using Malleable Application Malleable Agent (MA - MA) Strategy: In this strategy, the number of agents was chosen as the number of nodes in the application. It can be observed from Figure 9.11b that the initial number of agents is one. As the node count is expanded to 16, the number of agents is also increased to 16. Similarly, during the following resource reduction, the number of agents was brought down to eight and eventually four. The initial total checkpoint size is .48GB for the application, and as the number of nodes increased, the checkpoint size was increased to 7.68 GB and then later reduced to 3.48 and 1.92 GB, respectively, as the nodes were taken away. The figure shows that the checkpoint transfer time for the iCheck remains almost the same even as the checkpoint size was increased sixteen times. This is due to the change in the number of agents triggered by iCheck as it detects a change in resource in the malleable application. Similarly, when the number of nodes was reduced to eight, the agent count was changed to match the node count. Nevertheless, the checkpoint transfer time remained the same. The performance improvement can be attributed to the parallel RDMA-based checkpoint transfer performed by multiple agents in iCheck.

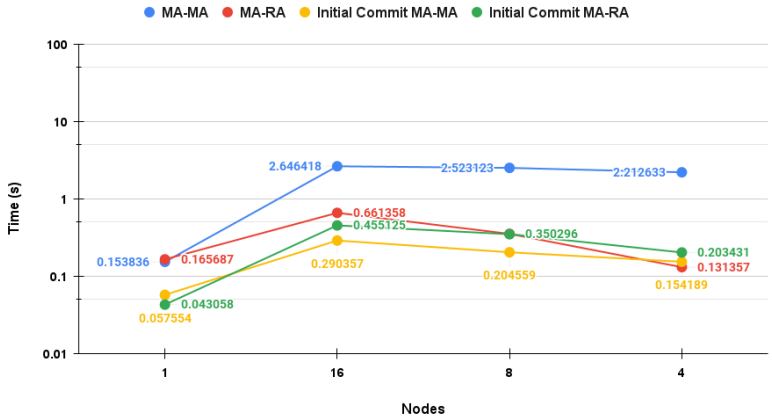
Overhead Analysis: Figure 9.12a shows the checkpoint transfer time in MA-MA and MA-RA strategies as well as provides a comparison with the IOR benchmark [234] for writing the same amount of data into the parallel file system. IOR is a popular benchmarking application for I/O performance analysis in HPC. In this analyis, IOR is run using MPI-IO under the hood and ensures that iCheck is compared with state-of-the-art I/O benchmarking application. Nevertheless, overhead analysis in iCheck also considers the checkpoint data preparation overhead which is not present in the IOR benchmark since it only measures the overhead for file operations (open and close) and data transfer. Figure 9.12b shows the overhead associated with agent change while analysing the MA-RA and MA-MA strategies. In addition, the time taken for the initial commit is also analysed. The difference between the initial commit and the other commits is the extra steps iCheck takes during the initial (or first) commit operation. During the initial commit operation, the iCheck library will perform the key exchange with iCheck agents to configure the remote memory and the data transfer. Hence, the initial commit takes more time than the proceeding commits. As observed in the figure, this holds for both MA-RA and MA-MA strategies.

Figure 9.12a demonstrates the performance variation of commit operation using both strategies. It is clear that the MA-MA strategy performs better than the MA-RA strategy as the number of nodes of the applications was changed. In all the resource change instances, the MA-MA strategy showed at least an improvement of 58% to 110%. This is due to the parallel checkpoint transfer provided by multiple agents in the

MA-MA strategy; meanwhile, a single agent has to perform the checkpoint transfer from the MA-RA strategy. In contrast to the IOR benchmark, both iCheck strategies outperform with a minimum improvement of over 90%. This performance improvement can be attributed to the RDMA-based checkpoint transfer in iCheck.



(a) Performance comparison with IOR Benchmark



(b) Overhead analysis for different strategies

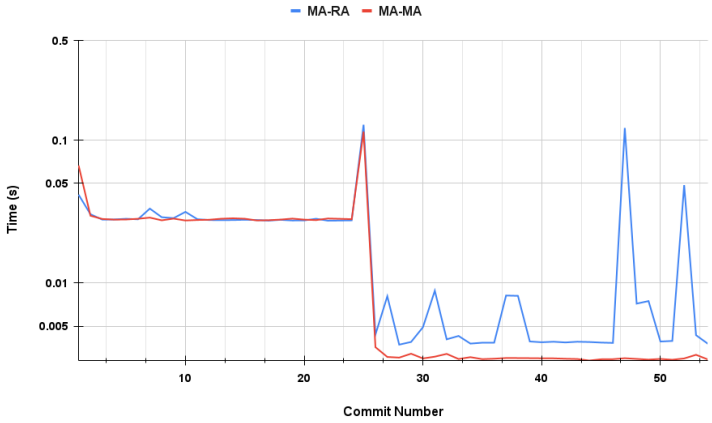
Figure 9.12: Overhead analysis of different strategies in comparison to IOR Benchmark (using MPI-IO)

Figure 9.12b showcases the resource change overhead for both the MA-RA and MA-MA strategies. As the graph shows, the resource change overhead is increased for MA-MA strategies as the agent number changes. This overhead arises due to the creation of new agents (or removal of existing agents) as well as connecting the application processes to the new agents. In MA-MA strategies, as the agent count increases, the overhead also increases. However, this is not the case for the MA-RA strategy, where the agent count is always one. As the number of nodes increases, the overhead increases and as the number of nodes decreases, the overhead decreases. The overhead is much lesser than the MA-MA strategy. This can be attributed to the need for connecting only a single agent to the application. As a result, the reconfiguration overhead is minimal in MA-RA strategy. However, the benefits of MA-MA strategies are associated with the agents' parallel checkpoint retrieval. As the agent number increases, more data can be retrieved in parallel, resulting

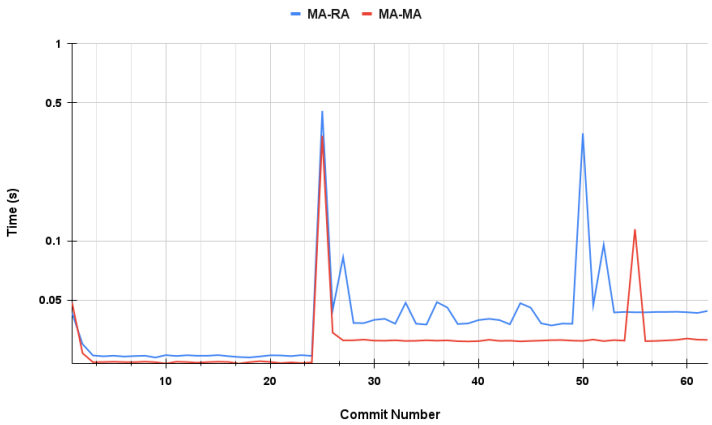
in performance improvement. This is visible in the time taken for checkpoint transfer in Figure 9.12a. This also holds true for the initial commit operation, where all application nodes can perform a key exchange with a single agent (this has to be done sequentially) in MA-RA strategy. Meanwhile, in the MA-MA strategy, there is a dedicated agent for each application node; hence, parallel key exchange is possible. Hence, there is a performance improvement for the initial commit in the MA-MA strategy, as seen in Figure 9.12b.

9.4.2.2 Malleable Application with Fixed Checkpoint Size

Section 9.4.2.1 showcased the overall performance impact of agent number on the checkpoint performance for a malleable application. Hence, the average time per commit operation is considered based on the number of commits that happened over the complete application execution. However, the following subsection dives deep into the time taken for individual commits during the application execution and resource change. It analyses the nuances which are not possible with the overall performance analysis. This reveals iCheck-specific insight into the RDMA-based checkpointing.



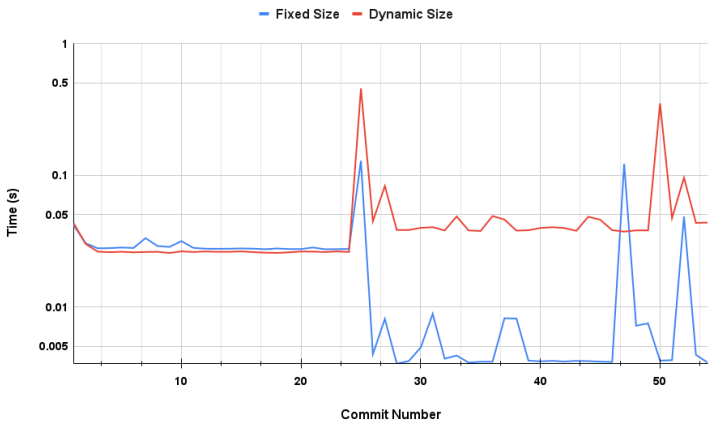
(a) Impact of iCheck on a malleable MPI application with fixed checkpoint size



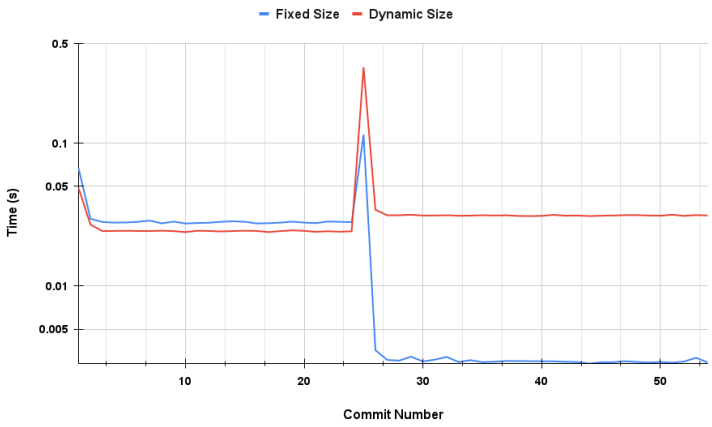
(b) Impact of iCheck on a malleable MPI application with dynamic checkpoint size

Figure 9.13: Effect of agents on a malleable MPI application with fixed and dynamic checkpoint size

Impact of checkpoint transfer on MA-RA and MA-MA strategies: Figure 9.13a compares the time taken for each commit on malleable application with no agent change (MA-RA) and malleable application with agent change (MA-MA) strategies for the following scenario. The initial checkpoint size (*size*) for a single node was chosen as 0.48 GB for easier comparison with the evaluation of the dynamic size technique with MA-RA and MA-MA strategies explained in the above subsection. Initially, each process gets a chunk of *size* ($size/(num\_nodes * num\_proc\_per\_node)$, where *num_nodes* can be 1, 4, 8 or 16, and *num_proc_per_node* is 48) for checkpointing. As the application expands to 16 nodes, the checkpoint size per process is further reduced since it is distributed among all the nodes ($size/(16 * num\_proc\_per\_node)$). The initial number of agents remained same (single agent) for both MA-RA and MA-MA strategies and as application is expanded to 16, the number of agents were also increased to 16 for the MA-MA strategy.



(a) Impact of iCheck using MA-RA strategy with fixed and variable checkpoint size



(b) Impact of iCheck using MA-MA strategy with with fixed and variable checkpoint size

Figure 9.14: Effect of agents on a malleable MPI application with fixed and dynamic checkpoint Size

It can be seen that the checkpoint time is similar for both MA-MA and MA-RA strategies till commit number 26. This is expected since the number of agents was the same for both strategies. However, a spike in the checkpointing time can be seen for the commit number 27 and afterwards the checkpoint time is reduced for both strategies. This spike can be attributed to the initial commit happening after a resource adaptation which

incurs additional overhead as explained in the above overhead analysis section 9.4.2.1. The checkpoint time reduction is attributed to the reduction in overall checkpoint size since, after the resource expansion, each process now only has $1/768^{th}$ of the total size to checkpoint, while previously, each process had $1/48^{th}$ of the total size (*size*) to checkpoint. This leads to faster checkpointing in both MA-RA and MA-MA strategies. Nevertheless, if we look into the time taken for individual commit, MA-MA has up to 20% performance improvement over MA-RA because of the additional agents in parallel retrieving the checkpoints.

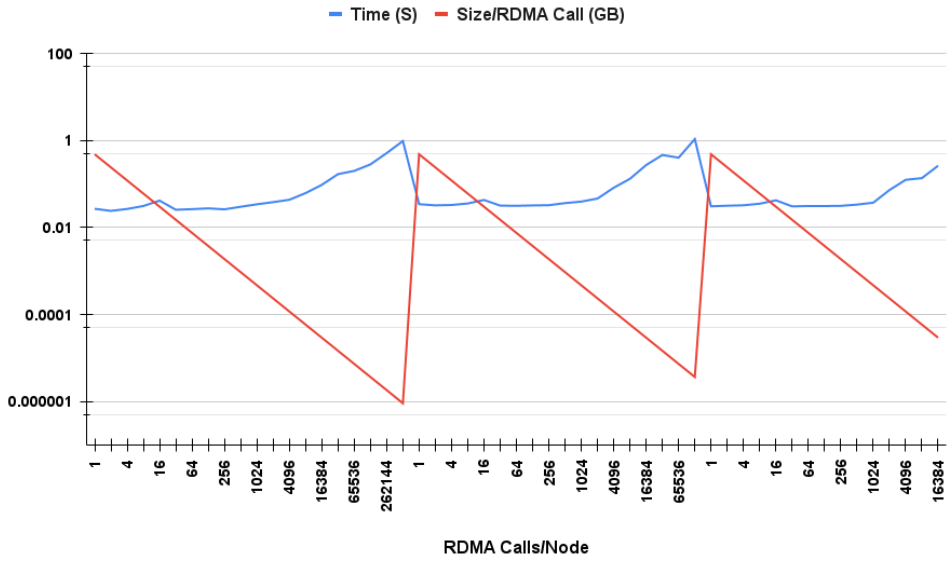
In addition, Figures 9.14a and 9.14b compare the time taken for both MA-RA and MA-MA strategies for fixed and dynamic size. It can be seen from Figure 9.14a that as the number of nodes are increased (after the 26th commit), the commit time was reduced for the fixed size evaluation and commit time was increased in dynamic size evaluation. This can be explained based on the change in checkpoint size. In the fixed size evaluation, as the application was expanded to 16 nodes, the total size to checkpoint per process was reduced by a factor of 768 while for the dynamic size evaluation it was increased by a factor of 768. For MA-MA strategy (see Figure 9.14b), the faster commit is associated with the fixed size of checkpoint through out the application execution even after the resource adaptation. Similar to Figure 9.14a, the dynamic size evaluation takes more time to checkpoint due to the increase in checkpoint size as the number of nodes were increased. It is also worth noting that, the fluctuations in commit time are not present in MA-MA strategy when compared with MA-RA strategy. The fluctuations in MA-RA strategy can be attributed to the effect of single agent retrieving the data from the nodes of the application sequentially.

Overhead analysis of the iCheck library is not provided for this section to avoid redundancy in discussion with Section 9.4.2.1 since the only difference is regarding the checkpoint size (data transfer size), whose impact is visible by plotting the time taken for the checkpoint transfer operation. The library-specific overhead associated with agent reconfiguration and commit call is detailed in Section 9.4.2.1.

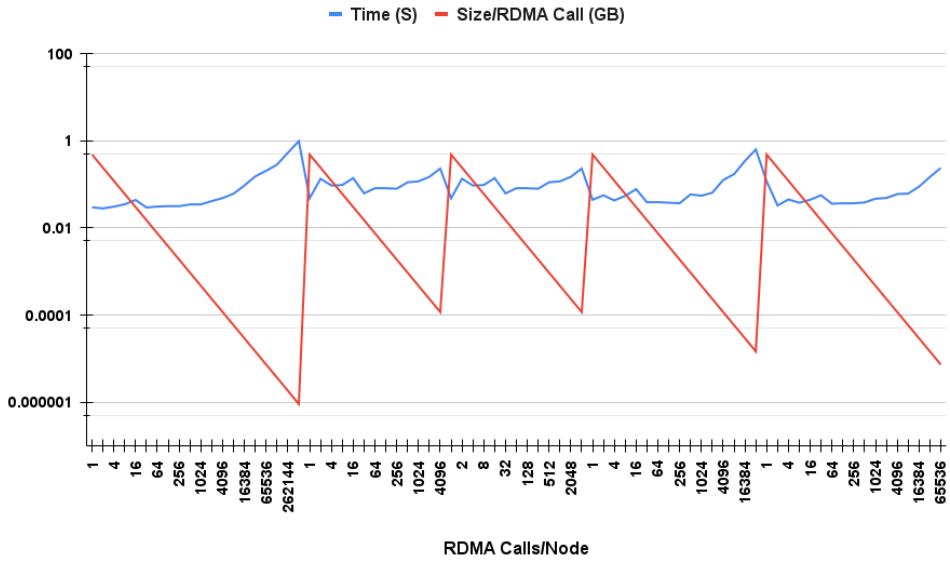
9.4.2.3 Impact of Pipelining

This section evaluates the performance of the agent-side pipelining feature described in Section 6.2.3.1. For this analysis, the synthetic application was initially run on a single node, and then iRM was leveraged to expand the resources to 16 nodes. Later, the number of resources was halved using iRM. The overall checkpoint size was increased from .48 GB to 7.68 GB after expansion and reduced from 7.68 GB to 3.84 GB during reduction. The pipeline performance is analysed for MA-MA and MA-RA strategies. Figures 9.15a and 9.15b depicts the time taken with regards to the checkpoint size and RDMA calls made for MA-MA and MA-RA strategies while Figure 9.16a compares in detail the performance variation in both strategies. Figure 9.16b compares the impact of pipelining on individual commits. In MA-MA strategy, the number of agents were the same as number of nodes while in MA-RA strategy the agent count was one.

Impact of Pipelining on MA-RA and MA-MA strategies: Figures 9.15a and 9.15b plots the relation between the checkpoint size (Size/RDMA call), the number of RDMA calls made and the time taken for the RDMA calls. Both the x and y axis are on log scale with base 10 and base 2 respectively for both the figures. The x axis depicts the number of RDMA calls per node while the y axis give insights into the time taken for checkpoint transfer as the pipeline size is varied. Size/RDMA call depicts the **pipeline size**. Each RDMA call retrieves the chunks of checkpoint with chunk size equal to the pipeline size. As seen in both figures, number of RDMA calls increase from 2^0 to 2^{18} initially, then from 2^0 to 2^{16} for MA-MA strategy and 2^0 to 2^{12} for MA-RA strategy after the resource change. This is due to the automatic change in pipeline size triggered after each commit in this analysis. In addition, the number of RDMA Calls/Node and Size/RDMA call (both are inversely propotional to each other) was also reset after every resource adaptation so that the nuances in checkpointing performance (behaviour of individual commit operation and resource change



(a) Performance of data transfer for various pipeline sizes for MA-MA Strategy



(b) Performance of data transfer for various pipeline sizes for MA-RA Strategy

Figure 9.15: Performance comparison of pipelining in MA-MA and MA-RA strategies

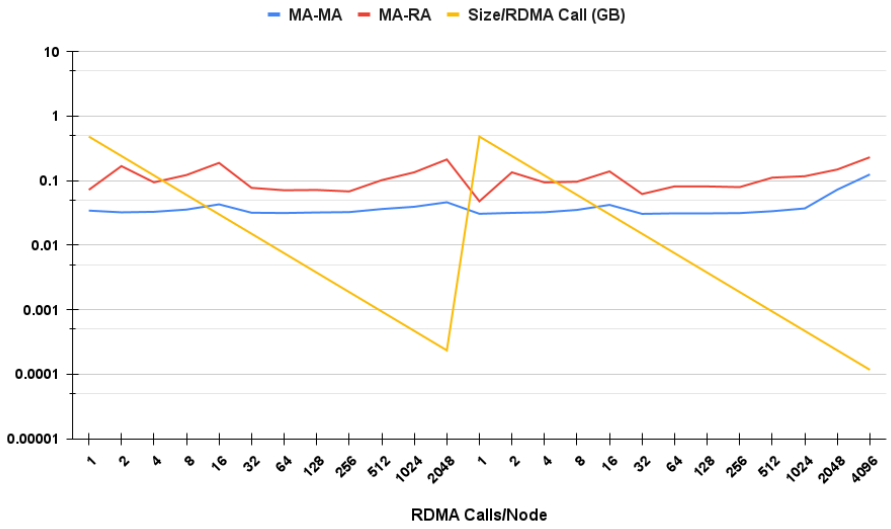
as the pipeline size varies) associated with pipelining can be compared with ease. This gives interesting insights into the checkpoint performance.

For example, in the MA-MA strategy in Figure 9.15a, the RDMA calls/Node performed increased to 2^{16} after an expansion while only upto 2^{12} for the MA-RA strategy in the same expansion with both application having the same checkpoint frequency. That means, with the MA-RA strategy with a single agent lags by 2^4 commits in comparison with the MA-MA strategy, implying that MA-MA strategy is faster which is consistent with the results in the above subsections. Initially, each node checkpoints a size of 0.48 GB, as the number of RDMA calls increases, the checkpoint size transferred per RDMA call decreases. For example, for 2^0 RDMA call, the checkpoint transferred per RDMA call will be the entire checkpoint which basically means there is no need for pipelining, while 2^1 RDMA calls per node, the transferred size will be 0.24 GB per call and the agent after copying the 0.24 GB, will write it into PFS and in the second RDMA call, reuses the previously allocated memory, hence pipelining is performed. Similarly for 2^{18} RDMA calls per node, each RDMA transfer checkpoints 1830 bytes. This can be visible in both the figures 9.15a and 9.15b. As seen in the figures, the size/RDMA call decreases as the number of RDMA call increases. However after a resource adaptation, the pipeline is reset for benchmarking and hence it restarts from complete checkpoint transfer per RDMA call. Resetting the counter means a resource adaptation has happened. So, it can be observed from 9.15a, two resource adaptations were performed, while Figure 9.15b shows that it had four resource adaptations. Another observable pattern in consistent with previous section is as agents are lesser than number of nodes, there is significant fluctuation in checkpoint transfer time even in case with pipelining. This is explained in below subsection.

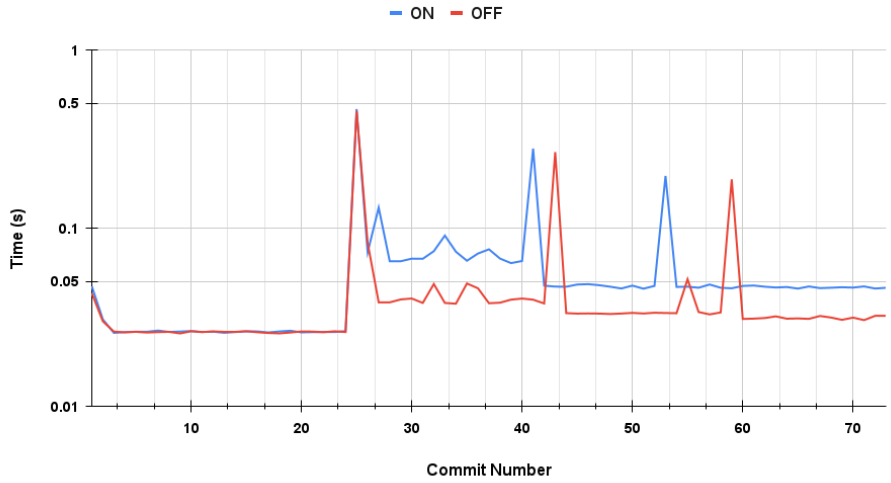
Comparing the Pipeline performance on MA-RA and MA-MA strategies: To simplify the comparison for MA-MA and MA-RA strategies, Figure 9.16a plots the time taken for checkpoint transfer with pipelining for sixteen nodes and a resource reduction to 8 nodes. The pipeline sizes from 0.48 GB (2^0 RDMA calls per node) to 0.23 MB (2^{12} RDMA calls per node) per node are analysed in the figure. As can be seen, the size per transfer is reduced as the number of RDMA Calls increases, and the time to transfer data follows a similar trend for both strategies. As expected, MA-MA performs better than MA-RA due to the parallelism in checkpoint transfer, and the performance difference between both strategies is consistent since the checkpoint transfer size per node remains the same for 16 and 8 nodes. Sixteen agents were used for 16 nodes, and the agent count was reduced to eight as the application was reduced to 8 nodes. Another observation, consistent with MA-MA and MA-RA (See the subsection 9.4.2.1) results, also persists here that the fluctuations are more predominant in MA-RA than MA-MA strategy due to the sequential checkpoint retrieval performed by a single agent. However, the intensity of the fluctuations is reduced due to the pipelining. This is because the stragglers can keep up with the faster processes due to the delay introduced by the pipelining technique.

Impact of Pipelining on individual commits: Figure 9.16b showcases the overhead of pipelining in comparison to no pipelining. In this analysis, the time taken for transferring checkpoint during each commit operation is studied while the application was expanded to 16 nodes from a single node and then reduced to 8 nodes, followed by another reduction to 4 nodes. MA-RA strategy with a single agent was employed in this analysis and each node contained 0.48 GB of checkpoint. This resource change can be seen around commit numbers 23, 43 and 53 with a spike in checkpoint transfer time. The pipeline buffer size is 0.48 GB, which means that the agent can store the complete checkpoint in its memory. This is intentionally done to analyse the overhead associated with pipelining. Since the checkpoint transfer size is the same with and without pipelining, additional time taken for pipeline mode gives insight into the overhead, which includes writing the data into a parallel file system after each checkpoint transfer. As expected, there is no performance difference for a single node with a single agent with and without pipelining since the agent can store 0.48 GB in its memory. However, the performance difference can be observed once the application is expanded to 16 nodes, where the agent needs to write the data into PFS after reading the checkpoint from

the first node and then transfer the data from the next nodes till all of the data is retrieved from all of the nodes. As a result, the agent incurs an overhead of 15 PFS writes with the pipelining technqiue (overhead for writing 0.48 GB to PFS during each write) when the application is running on 16 nodes. This behaviour is also consistent when the application was reduced to eight nodes and later to four nodes (As seen in commit numbers 40 and 53 in Figure 9.16b). Hence, there is a consistent performance difference for checkpoint transfer with and without pipelining.



(a) Performance of data transfer for various pipeline sizes after expansion and reduction

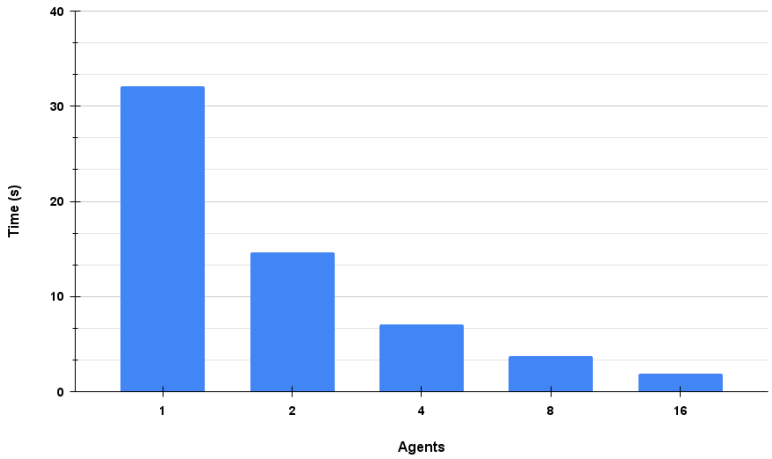


(b) Performance of data transfer for a single size with pipeline

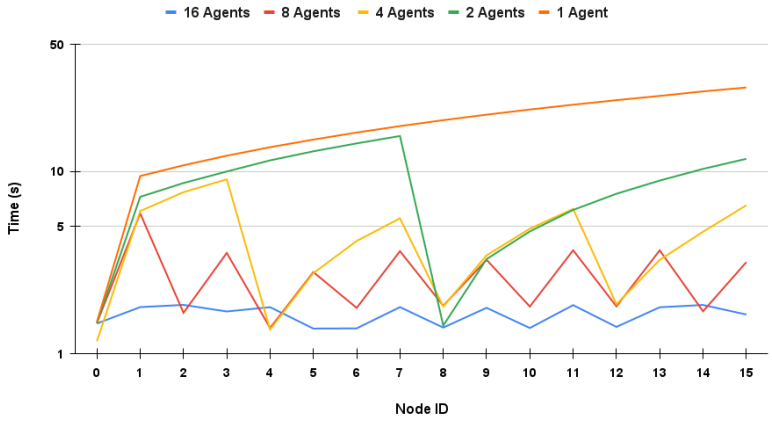
Figure 9.16: Performance comparison with and without pipelining

9.4.3 Data Distribution in iCheck

This section showcases the experimental evaluation of the simple `BLOCK` data distribution capability of the iCheck system. A synthetic malleable benchmark application (Section 8.2.3.2) was used to analyse the data redistribution feature. The application resources were expanded and reduced dynamically using iRM, and the data distribution functionality of iCheck with regard to the resource changes was analysed. The application was run on up to 16 nodes, and the iCheck system was run on two nodes. In the synthetic benchmark, the application checkpointed data of size 1.5 GB every five seconds. This section analyses the following aspects of the iCheck data redistribution feature. Firstly, the effect of different numbers of agents (MA-RA and MA-MA in Section 8.2.3.2) on data distribution was analysed in detail in this section. For this analysis, the synthetic application was initially run on a single node, and then iRM was leveraged to expand the resources to 16 nodes. The checkpoint size (or size of the global array) remained the same for resource expansion. Later, the number of resources was halved using iRM. Secondly, the redistribution time of iCheck is compared with that of a naive MPI-based data redistribution technique.



(a) Impact of agent count on data transfer time



(b) Impact of agent count on configuration (log scale)

Figure 9.17: Analysis of data redistribution in iCheck

Table 9.4: Lines of Code for data distribution on Malleable Application

Type	iCheck	MPI-Based
block	1	> 3

9.4.3.1 Performance Analysis

Figure 9.17a indicates the time taken for data redistribution for different agent configurations as the application is expanded from a single node to sixteen nodes. The application was expanded from a single node to 16 nodes, using the MA-RA and MA-MA strategies for the agent change. In the MA-MA strategies, the factors of the node size were used as the agent count, resulting in a distribution of two, four, eight and sixteen agents for the expansion to 16 nodes. As shown in the plot, the time taken for data redistribution is greatly reduced as the number of agents increases. This can be attributed to the data transfer parallelism made possible by the number of agents and is consistent with the observations made in Section 9.3. When there are sixteen agents, every agent can simultaneously transfer the data from corresponding nodes (a 1:1 mapping, which leads to less waiting time inside the application to transfer the checkpoints), giving maximum performance. On the other hand, with a single agent, the data transfer has to happen sequentially from each of the 16 nodes, which leads to a longer redistribution time (17 times slower than 16 agents), as observed in Figure 9.17a. Similarly, using agent sizes of two, four and eight results in performance improvements of $2x$, $4x$ and $8x$, respectively.

Analysing the time taken for each agent's data transfer and configuration offers exciting insights into the data redistribution process occurring within iCheck. Figure 9.17b shows the total time for libfabric configuration and data transfer with node-level granularity represented in a logarithmic scale when the application was expanded from a single node to sixteen nodes. When the number of agents is the same as the number of nodes (16 Agents), the data transfer time is steady across all nodes. However, some interesting insights can be derived by looking into the time taken for two agents in the figure. With a two-agent configuration, each agent is responsible for eight application nodes, with the first agent (say Agent 1) responsible for nodes zero to seven and the second agent (say Agent 2) for nodes eight to fifteen. This can be attributed to the variation in performance seen across Node IDs 0-7 and 8-15 in Figure 9.17b. As seen in the figure, there is a steady increase in transfer time across Node IDs 0-7 and Node IDs 8-15 since agents first send data to nodes 0 (Agent 1) and 8 (Agent 2) and then move on to nodes 1 (Agent 1) and 9 (Agent 2), which continues until the data is transferred to all nodes. As a result, the processes in the other nodes will wait for the data to be received, increasing the overall transfer time. For example, while Agent 1 transfers data to the 1st Node (Node ID 0), the processes in the remaining nodes (Node ID 1 - 7) will be waiting. Likewise, this pattern can also be observed in other agent configurations. For instance, for a single agent scenario, the transfer time is steadily increasing up to node 15, while for four agents, the time is increasing steadily from nodes zero to four, four to seven, eight to eleven, and twelve to fifteen since four agents are retrieving data in parallel. This leads to an interesting graph shape from which the agent count can be derived simply by looking into number of peaks in the corresponding line in the graph. The order of retrieval of checkpoint can also be derived by traversing the peaks and valleys in the graph.

9.4.3.2 iCheck vs MPI-IO

Figure 9.18 compares the time taken for data redistribution when the application was expanded from one to sixteen nodes, and then the application was later reduced to eight nodes from sixteen nodes. iCheck with the best agent configuration yields $7x$ faster data distribution during expansion and $9x$ faster data distribution

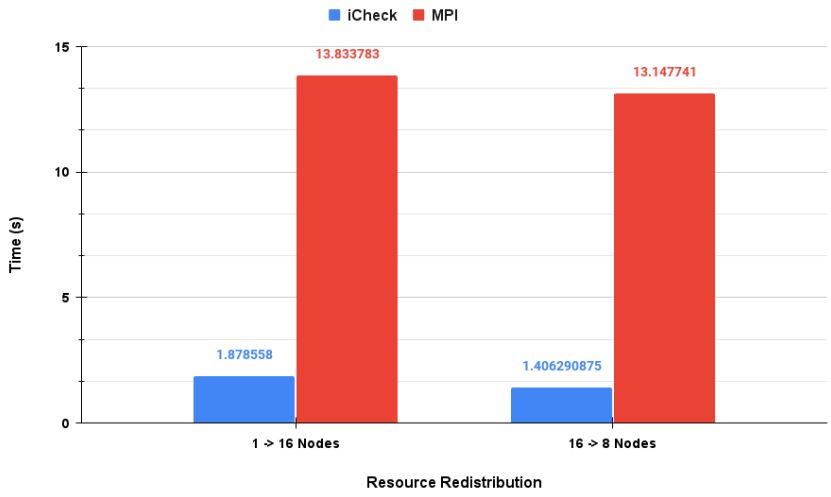


Figure 9.18: Data distribution overhead comparison

during reduction. This performance improvement is attributed to the RDMA-based data transfer from the iCheck system to the application. In the MPI-based data redistribution, a naive approach was implemented, wherein the expansion from a single node to sixteen nodes, the lead process first gathered the data before the expansion process, and later, after the expansion, the lead rank scatters the data into all the nodes using the collective MPI operations (`MPI_Gather` and `MPI_Scatter`). This is a naive implementation; hence, iCheck outperforms it due to the optimised RDMA-based approach. In addition, Table 9.4 shows the ease of use comparison for data distribution based on iCheck and MPI. In iCheck, `icheck_redistribute()` API needs to be called once by processes, while in MPI-based implementation, the application developer needs to add the code for gather and scatter (or Write and Read to PFS if MPI-IO is utilised), then based on the resource change type (expansion or reduction), the data distribution logic needs to be calculated (additional lines of code). For example, in the case of reduction (from 16 nodes to 8 nodes), the developer needs to calculate the processes in 8 nodes to which the data from the original 16 nodes to be transferred. This can complicate the application development process and can be avoided using the data redistribution capability offered by iCheck. In addition, the developer does not need to worry about sending the entry point of the newly joined application processes (See Section 2.2.2.2) since these variables can be restored directly using iCheck (Section 7.4.1), further reducing the number of lines of code.

10

Discussion

This chapter provides an overview of the key takeaways from the results section and how they align with the objective of the undertaken work. The discussion is presented in the order of the contributions of the work performed as part of this research work. The three following sections describe the results of the contributions in the order of the definition of their corresponding chapters (Chapters 4 - 7). Initially, a discussion associated with adaptive batch scheduling is provided, followed by a discussion about the results of power corridor management contribution, and finally, key takeaways from the adaptive checkpointing work are presented.

The primary motivation of this work is to demonstrate that dynamic resource management can be leveraged not only to improve system utilisation but also to utilise it for power management and fault tolerance. By analysing the experimental results provided in Section 9, it can be concluded that dynamic resource management is a potential candidate for tackling the major concerns in HPC like efficient system utilisation, power management and resilience to failure. An overview of the findings from this work and its relevance is discussed below:

10.1 Adaptive Batch Scheduling

The adaptive batch scheduler introduced in Chapter 4 performs runtime reconfiguration of resources of the running applications based on some predefined policy (For example, application performance is considered for decision-making) to improve the overall system utilisation. As a result, more jobs can be scheduled faster, leading to a better system makespan, which is demonstrated in Section 9.1. Although this assumption is obvious (changing resources of running applications to make space for the new applications and thereby improving the system utilisation), the nuances of the concepts were not discussed in the current literature. In addition, a working prototype on a real machine needed to be demonstrated to portray such nuances present in adaptive batch scheduling. From the performance metrics aspect, it can be concluded that using the performance-aware batch scheduler, an improvement of up to 30% can be obtained in overall system utilisation compared to a static backfill scheduler.

Nevertheless, the key takeaway is the percentage of malleable jobs considered for the evaluation. As seen in figure 9.1b, even with only half of the malleable jobs in the queue, the overall system utilisation can almost be doubled with a malleable resource management technique. The trend is similar also for makespan,

average response time and waiting time of the jobs. In addition, it is also visible from the makespan results (Figure 9.1a) that a performance-aware system that considers the application-specific characteristics (compute intensive/memory intensive) can trigger the resource change in such a way that it rewards the high-performing applications. As seen from the experimental results portrayed in figure 9.1, the performance-aware scheduler gives the best results for all metrics compared to the static backfill scheduler and FPSMA technique from the literature. It is due to the consideration of application characteristics that improved the overall metrics. This insight laid the foundation for the power-aware scheduling techniques developed as part of this work. The application's performance characteristics can also translate to the power usage characteristics. For example, a compute-intensive application can be a power-hungry application. Hence, the power-aware scheduler can be used to manipulate the system's power consumption by leveraging dynamic resource management concepts.

Another critical takeaway is that even with 100% malleable jobs, the system was not able to achieve above 85% of system utilisation. It is the indirect consequence of the constraints put forward by the application and is adhering to real-life application scenarios. For example, some applications require a specific number of resources (LULESH requires a cubic number of nodes for the execution). As a result, not all applications can be scaled as the resources become available; hence, there will always be idle nodes in the system. This notion is used as a building block for resource-aware checkpointing, where the dynamic resource manager can indirectly use the idle nodes to improve the application performance by giving more resources to the checkpointing system based on the checkpointing requirements.

10.2 Power Corridor Management

The proposed power-aware scheduler demonstrated that given there are malleable applications that have the power usage characteristics defined in Section 5.1.2, dynamic resource management, along with launching new applications, can ensure that a system can operate in the specified corridor. The key concept is taking into account the power usage characteristics of the application. The power-aware scheduler can reconfigure the applications in such a way that the overall power consumption of the system can either be increased or decreased. The fundamental limitation of this approach is the dependence on malleable applications. This approach requires the presence of sufficient malleable applications with varying power usage characteristics. If all the applications have similar power usage characteristics and the system is fully occupied, reconfiguring the resources will not bring the system back into the power corridor.

The experimental results from Figure 9.2 show that the runtime system can reconfigure the running applications to increase or decrease the power usage as per the demand. Additionally, the figure shows that the power-aware scheduler can launch new jobs as well as maintain the system-level power budget even when the power corridors are dynamically altered. Also, the makespan table (Table 9.2) demonstrates that the power-aware scheduler can get better makespan than the power-aware runtime system. It can be seen from figure 9.2.4 that leveraging the dynamism can be significant in maintaining the power budget. At the same time, the static backfill scheduler fails to adhere to the power budget. However, analysing the makespan of the power-aware and static scheduler demonstrates that the power-aware scheduler might have a slightly worse makespan than the static one. It is the cost of adhering to the system-level power corridor requirements. However, optimising for application performance might be counterproductive to achieve the system-level power budget. Nevertheless, it can be derived from the results that the system-level power budget can be maintained without heavily penalising the applications, like killing the applications as described in the literature.

10.3 iCheck - Invasive Checkpointing System

iCheck was designed to be adaptive and flexible to the changes in application characteristics. These design aspects can be used to:

- leverage the dynamism of resources (self-adaptivity) to improve the application's performance.
- satisfy the checkpointing needs of malleable applications.
- perform as a data distribution framework for malleable applications.

These three aspects were successfully demonstrated in the evaluation part of iCheck in Chapter 9.

The performance of iCheck compared to MPI-IO and SCR is demonstrated in Section 9.3.5. It is evident that iCheck performs faster checkpointing due to the use of RDMA technology and hence results in faster execution of the application. As depicted in table 9.3, utilisation of iCheck in the ls1 mardyn application resulted in 6% faster execution of the application compared to the in-house MPI-IO-based implementation. In addition, iCheck offers different checkpoint transfer techniques (See figures 9.3.2 and 9.5) that improve the overall checkpointing performance in an application.

iCheck also demonstrates that scaling its resources can significantly improve the checkpointing performance of the application (See Figures 9.4a and 9.6a). Furthermore, it can also be seen (from Figure 9.7a) that dynamically adding the agents improves the checkpointing time. It is observed that the overhead of dynamically adding the agents is negligible compared to the attained performance advantage. Figure 9.7a shows that there was a performance improvement of up to 80% while using dynamic agents. In addition, iCheck also demonstrated that different agent placement techniques that react to the nuances of the system can improve the checkpoint transfer in applications (Figure 9.7b). The malleability offered by iCheck enables combining these two functionalities (dynamic agent change and agent placement) and creates a faster and more efficient checkpoint transfer mechanism even for a non-malleable application. Further, Subsection 9.4.1.4 discusses the performance of the fallback checkpoint transfer mechanism employed by iCheck in case of a failure in the iCheck core system.

In addition, the horizontal scaling ability of iCheck depicted in Figure 9.8b presents opportunities to utilise the idle nodes of the system efficiently. Leveraging the horizontal scaling capabilities discussed in Section 9.4.1.3, iCheck introduces two significant advantages to checkpointing in high-performance computing environments. The first advantage is eliminating the necessity to allocate a fixed set of nodes exclusively for checkpointing purposes. Instead, a resource manager can dynamically adjust the number of nodes designated for checkpointing services, tailoring this allocation to the current resource availability within the system. Secondly, as HPC is moving towards the cloud [235], a dynamically scaling fault-tolerant system will benefit from a cost-based cloud model where it can expand or shrink its checkpointing nodes and agents based on the application demands and objectives.

Figure 9.11 demonstrates the ability of iCheck to support malleable applications and showcases the performance improvement brought by malleability-aware agent change techniques in iCheck. Figure 9.12 showcases the improvement of upto 110% among different iCheck strategies for malleable applications and an improvement upto 90% in comparison with IOR benchmark. In addition Figure 9.15 depicts the capability of pipelining in iCheck. Further, Section 9.4.3 paints the effect of data distribution using the iCheck provided APIs. The ease of using iCheck API is visible in the table 9.4, describing the necessity of fewer lines of code compared to the standard techniques for data distribution after a resource change. In addition, the overhead associated with data distribution is negligible compared to the benefits provided by the easiness of data redistribution.

11

Conclusion

Innovative solutions are needed to overcome supercomputing's numerous challenges, ranging from power management to fault tolerance and scalability. Dynamic resource management is one such solution that offers a flexible and intelligent approach. Using dynamic resource management, this work improved system utilisation, optimised power consumption and enhanced fault tolerance. As supercomputing continues to push the boundaries of scientific discovery and technological innovation, dynamic resource management enables supercomputers to utilise technological advancements efficiently while maintaining power efficiency and system reliability.

This work successfully showed that dynamic resource management can tackle diverse challenges in three different domains of supercomputing. The short summary of contributions and domains associated with each contribution is listed below:

- **Resource and Job Management System:** This work demonstrated that creating a performance-aware batch system can improve the system's overall performance (Section 9.1). The system utilisation was improved by 30%; the average job waiting time was improved by 25%; the makespan was improved by 20%. The improvement was demonstrated on SuperMUC-NG using a tsunami simulation application. In addition to a performance-aware batch system, different scheduling strategies were developed to showcase the usefulness of malleable job management. It is also demonstrated that significant improvement can be obtained for system metrics even with less than half of the applications in the queue are malleable.
- **Power Management:** In the area of power management, the contribution from this work is a power-aware runtime system and batch scheduler (Section 9.2). It showcased that redistribution of resources and power-aware job scheduling can keep the system in a specified power budget without using commonly used techniques like DVFS and power capping. It is also demonstrated that the system can successfully adapt to varying power budgets (dynamic power corridor management). The experiments were run on SuperMUC and SuperMUC-NG using heat simulation, Pi Calculation and LU decomposition applications.
- **Fault Tolerance:** This work brought novelty in four aspects of checkpointing in HPC. Firstly, a new RDMA-based application-level checkpointing system that can dynamically reconfigure its resources to improve the overall application performance is demonstrated (Section 9.3). The results were taken

on SuperMUC-NG using applications like ls1mardyn, lulesh, heat simulation and synthetic application. Secondly, the checkpointing system is extended to support malleable MPI applications that can dynamically change its resources. It is also demonstrated that iCheck can improve a malleable and standard MPI application's performance by reconfiguring the checkpointing resources to provide faster checkpointing services. A synthetic application is used to illustrate the impact of iCheck on malleable applications. In the third case, the iCheck-aware resource manager plugin is created to cater to the horizontal scaling capability of iCheck, which is beneficial whenever iCheck runs out of memory for the application. Lastly, the proof-of-concept checkpointing system as a data distribution framework is demonstrated successfully (Section 9.4). The data redistribution easiness using iCheck compared to the manual data distribution scheme was also portrayed.

These contributions demonstrate that dynamic resource management is suitable for approaching multiple supercomputing challenges. Further, it is also evident that the success of dynamic resource management depends upon the availability of malleable applications. A supercomputer's impact lies in its use case, specifically the applications that leverage high-performance hardware to create breakthrough inventions. Similarly, if there are not enough malleable applications to utilise the dynamism in HPC, the benefits of dynamic resource management cannot be fully reaped. Towards that, the research community's growing interest in malleability gives a promising outlook into the future of HPC. The contributions from this work can be leveraged to create a better malleable ecosystem for HPC.

11.1 Into the Future

The evolution of high-performance computing is invariably characterised by rising complexity and requirements, pushed by applications that become ever more ambitious. Three pivotal domains that can contribute immensely to addressing the next generation of high-performance computing systems performance, sustainability and reliability are dynamic resource management, power management and fault tolerance. As seen in this work, this approach can improve efficiency and performance. However, it also introduces several challenges and limitations that must be addressed. These challenges include ensuring that resources are allocated fairly and efficiently, managing the system's complexity, and ensuring the system is reliable and safe. As part of this work, some areas that require further investigation are identified and introduced below.

- **Resource and Job Management System:** This work's contributions to adaptive batch scheduling only consider the current application performance. The resource management system should ideally have astute knowledge about the effect of resource change on an application's performance. The MTCT metrics (see section 4.2.3) considered for triggering the resource change decision do not reveal information about the peculiar characteristics inside different phases of an application. For example, a resource change might turn a compute-bound application into a memory-bound one and might not benefit from the resource change. In this arena, there is much potential for improvements. Machine learning and AI-driven algorithms can be leveraged to model the application characteristics and make scheduling decisions based on this information. This could potentially improve the overall makespan of the system and increase the system utilisation.

  As seen in the above sections, the availability of malleable applications gives flexibility to the RJMS system to improve the overall system performance. However, writing malleable applications is a cumbersome process. Towards that, incentive-based approaches can be devised where the creators of malleable applications can be rewarded with more resources (CPU hours) or higher priority (during job submissions) if the creators provide malleable applications that the resource manager can manipulate. However, this is indeed challenging since the change of resources in the malleable applications

can impact the classical application performance analysis techniques and skew the benchmark results. For example, the application can switch between memory-bound and compute-bound based on the resource change. Performance analysis and optimisation will be tricky (For example, performing roofline analysis [236]), making it difficult to fine-tune the application. In that regard, another area to look into is the tool support for such change in resources. Most tools are designed with static MPI in mind, and considerable effort will be needed to port existing tools to support malleable applications. Lastly, the granularity of the resources in this work can be extended to cores. Currently, the resource manager takes the resource change decision based on the granularity of a node (consistent with the exclusivity of compute nodes in a typical supercomputing job submission scenario). As a result, a resource change is performed in the number of nodes and the MPI processes matching the number of cores are launched in the allocated node. A granularity of cores can be utilised to make more intelligent resource change decisions. In addition, the granularity of cores can also be utilised for the co-scheduling [237] scenarios.

- **Power Management:** This work's proposed power management strategy leverages the dynamic resource management techniques and guarantees that the system can maintain/change its power usage in/to the specific corridor if some predefined constraints are met (see section 5.1.2). Notably, the presence of malleable applications is crucial for materialising such a concept. However, the proposed approach is ideal only if sufficient malleable applications are present with ideal power profiles to bring the system back into the power corridor. The current state-of-the-art approaches (Power capping and DVFS) are alternatives in such scenarios. Suppose the applications cannot meet the criteria to change the power consumption using dynamic resource management. In that case, the power-aware scheduler can switch to the conventional techniques to maintain the power budget. However, the ideal approach is to merge the existing method of DVFS and power capping along with dynamic resource management to create a hybrid power management system that benefits from both conventional power management techniques and the flexibility offered by the dynamic resource management technique.

  Combining these techniques opens up opportunities for efficient power management. As a result, the power-aware resource manager's decision-making domain is expanded, and multiple techniques (power models) can be employed to tackle power management. For example, the power-aware resource manager can change the resources of a set of applications and manipulate the voltage and frequency to reach a power range that is not reachable only with dynamic resource management or DVFS. Furthermore, predictive models can be used to forecast the power usage of the applications with new resources. The information about the power usage characteristics of the application with regard to its scaling characteristics will be beneficial for decision-making. This enables the power-aware scheduler to make intelligent decisions regarding the resource change. For example, the resource manager can selectively perform resource change and frequency scaling on applications based on their power profiles. Another aspect is to utilise the dynamic power corridor management infrastructure and link it with information about the energy source. As a result, the power-aware scheduler can operate in high power mode (higher power corridors) whenever the energy is supplied from green energy sources (for example, the wind and the solar energy) and low power mode when the energy is supplied from conventional energy sources (For example, Coal) thereby creating a green power-aware batch scheduler.

  Another addition to this work can be about changing the granularity of the resources. In this work, the resources have the granularity of nodes, and all of the cores of a compute node are utilised whenever a resource expansion is triggered. This hinders the opportunities for fine-tuning the power usage inside the node by manipulating the processes inside a compute node. As a result, a change in the granularity of resources from compute nodes to compute cores can improve the power-aware scheduler's performance.

- **Fault tolerance:** This work's proposed checkpointing system targets classical and malleable HPC applications running on CPUs. In addition, iCheck design also supports both homogeneous and heterogeneous CPU systems. However, HPC is going beyond homogeneous CPU systems as more GPUs and accelerators are becoming a standard addition to top HPC systems [238]. Even though the iCheck core can run out-of-the-box MPI applications on the heterogeneous nodes of a cluster, one of the significant limitations of the iCheck system is the need for more support for accelerators. In principle, the iCheck system can checkpoint the data from the compute node of the application (even though the node is connected to GPUs) that also utilises the GPU for its computation. However, iCheck cannot read checkpoints directly from the GPUs; instead, it can be performed via RDMA/TCP from the primary memory of the compute node. Hence, iCheck must be further extended to support GPUDirect RDMA [239] to transfer data directly from GPUs to the iCheck core to provide faster checkpointing services to GPU-based applications.

  In the current implementation, iCheck does not leverage the benefits offered by machine learning techniques for checkpointing [240, 241]. iCheck decides on the resources for checkpointing and where to store the checkpoints based on the empirical data and the data provided via the configuration file. In this regard, improvement is possible by leveraging reinforcement learning [242] techniques to predict the number of resources to be allocated for an application and which strategy to use to place the checkpoints in the compute nodes. Based on the feedback, the checkpointing resources can be changed dynamically due to the adaptive design of iCheck. As a result, combining machine learning with the dynamism offered by iCheck could improve the checkpointing performance of the application. In the end, a fully automated self-adaptive checkpointing system can be created by applying machine learning techniques. Furthermore, the same learning techniques can be leveraged to decide when to transfer checkpoints to PFS. Currently, iCheck only supports PFS as the second level in its design and could also be extended to support NVMs and SSDs as second level and can be trained to use the storage hierarchy based on the system and application performance characteristics [185].

  Further, the data distribution API provided by iCheck only supports simple data distribution use cases (Block and Cyclic) and custom user-level functions. iCheck could be extended to support already available load balancing and data distribution libraries to cater for a wide array of HPC applications. Another potential for improvement is to utilise the predictive failure analysis to anticipate and isolate the hardware that can fail. This technique could be leveraged by iCheck and iCheck-aware schedulers to improve the application's resilience. Resource managers can proactively remove the compute nodes (that are predicted to fail) from the application, and iCheck can restore (redistribute) the data faster to improve the overall resiliency of the application.

As discussed, the proposed work can be improved by leveraging the machine learning techniques in the three domains. The resource manager can proactively avoid failures, perform resource redistribution based on the application's scalability, and efficiently manage the system's power usage. Towards that, a unified scheduler plugin must be devised. With the proposed work, three different scheduler plugins along with an application-level checkpointing system were developed to tackle the three aspects of HPC. These could be clubbed to create a one-for-all resource manager plugin aware of dynamism, power and failures in the system. Such a resource manager can further delve into the interdependencies between these three domains. The journey towards exascale computing and beyond is undoubtedly challenging, but it also presents many opportunities for innovative solutions. The HPC community should leverage dynamic resource management to achieve revolutionary breakthroughs in system utilisation, power efficiency and resilience. As the need for computational power continues to increase, and sustainability initiatives continue to gain momentum, the procedures and technologies developed by leveraging dynamic resource management in these areas will undoubtedly shape the next era of supercomputing.

List of Figures

1.1 Overview of the contributions from this work . . . 10

2.1 Invasive computing fundamental concepts [81] . . . 13
2.2 Research groups in Invasive Computing [81] . . . 14
2.3 Slurm Architecture . . . 15
2.4 Invasive Resource Manager [88] . . . 17
2.5 Control flow of a sample iMPI application [88] . . . 21
2.6 Control flow of a sample EPOP application [88] . . . 24

4.1 Adaptive Batch Scheduler Architecture [88] . . . 38

5.1 Adaptive Power-aware Batch Scheduler Architecture [88] . . . 45
5.2 Forecasting using different techniques . . . 49

6.1 iCheck system architecture [77] . . . 53
6.2 iCheck Agent . . . 53
6.3 iCheck Manager . . . 54
6.4 iCheck Controller . . . 54
6.5 iCheck architecture hierarchical view . . . 55
6.6 RDMA in iCheck . . . 59
6.7 Buffer Management in MR based checkpointing . . . 60
6.8 Buffer Management in SHMR based checkpointing . . . 61
6.9 iCheck Buffer Management in SHMR based checkpointing using the CP Manager and Coordinator . . . 62
6.10 Pluggable services in an iCheck agent . . . 69
6.11 Agent reconfiguration . . . 70

7.1 High level interaction overview of iCheck, iRM, and application from an iCheck Perspective. 96
7.2 State transition diagram of iCheck library (High level overview). . . . 101

9.1 Evaluation of the adaptive batch scheduling system [55] . . . 110
9.2 Power Corridor enforcement using the resource reconfiguration among the running malleable applications [59]. . . . 111
9.3 Average system power usage and power corridor violations for different scheduling scenarios [55]. . . . 114
9.4 Checkpointing analysis on ls1 mardyn using iCheck and MPI-IO [77]. . . . 117
9.5 Comparing checkpoint performance in Push and Pull Techniques. . . . 118
9.6 Checkpointing analysis on synthetic application using iCheck and MPIIO [77]. . . . 119
9.7 Effect of dynamic agents and agent placement strategies [77]. . . . 121
9.8 Demonstrating the horizontal scaling in iCheck during a time interval [77]. . . . 122

9.9 Analysis of fallback mechanism for checkpoint transfer in iCheck 123
9.10 Data compression inside iCheck 125
9.11 Effect of dynamic agents on a malleable MPI application with dynamic checkpoint size . . . 126
9.12 Overhead analysis of different strategies in comparison to IOR Benchmark (using MPI-IO) . 128
9.13 Effect of agents on a malleable MPI application with fixed and dynamic checkpoint size . . 129
9.14 Effect of agents on a malleable MPI application with fixed and dynamic checkpoint Size . . 130
9.15 Performance comparison of pipelining in MA-MA and MA-RA strategies 132
9.16 Performance comparison with and without pipelining 134
9.17 Analysis of data redistribution in iCheck 135
9.18 Data distribution overhead comparison 137

List of Tables

1.1 Name, architecture, number of cores, performance and power usage of the top five HPC systems in the Top500 list on November 2023 [15]. 3

9.1 Workload characteristics for the modified ESP Benchmark [223] used in ABS analysis [55]. 109
9.2 Comparison of metric values for different scheduling scenarios [55]. 114
9.3 Effect of checkpointing on execution time in ls1 mardyn [77]. 117
9.4 Lines of Code for data distribution on Malleable Application 136

List of Algorithms

1 The Adaptive Batch Scheduler (ABS) Iteration [55]. 36

2 Adapted Favor Previously Started Malleable Applications First (FPSMA) job scheduling strategy [117]. 40

3 The ABS Performance-aware Scheduling function [55]. 41

4 The ABS power-aware scheduling function [55]. 50

5 The agent count selection algorithm 75

6 The agent placement algorithm 76

7 The agent count selection algorithm 82

8 The controller horizontal scaling proactive algorithm 87

9 The controller horizontal scaling reactive algorithm 88

10 The iCheck aware scheduling function with priority for iCheck system. 92

11 The iCheck aware scheduling function with priority for malleable application. 95

List of Listings

2.1 Example batch script with additional parameters for `sbatch`. 16
2.2 Pseudocode of a sample iMPI application. 22
2.3 Pseudocode of a sample EPOP application [59]. 23

4.1 Simple batch script for adaptive batch scheduling [55]. 35

5.1 Example batch script with additional options for power-aware scheduling [55]. 48

6.1 Pseudocode of a simple iCheck enabled MPI application [77]. 57
6.2 iCheck dynamic configuration file. 67
6.3 iCheck static configuration file. 67

7.1 Psuedocode of a malleable application with iCheck [78]. 84
7.2 Psuedocode of a malleable application using iCheck data distribution API [77]. 100

8.1 The configuration file for synthetic benchmark application. 105
8.2 Pseudocode of a synthetic benchmark using iCheck. 106

Bibliography

[1] National Weather Service. *About Supercomputers*. URL: https://www.weather.gov/about/supercomputers.

[2] Zameer Shervani et al. "World's Fastest Supercomputer Picks COVID-19 Drug". In: *Advances in Infectious Diseases* 10 (Jan. 2020), pp. 211–225. DOI: 10.4236/aid.2020.103021.

[3] Gerard Dumancas. "Applications of Supercomputers in Sequence Analysis and Genome Annotation". In: Jan. 2015, pp. 149–175. ISBN: 9781466674622. DOI: 10.4018/978-1-4666-7461-5.ch006.

[4] International Nuclear Information System. *Supercomputer applications in nuclear research*. URL: https://inis.iaea.org/search/searchsinglerecord.aspx?recordsFor=SingleRecord&RN=23076643.

[5] V. Daniel Elvira et al. *The Future of High Energy Physics Software and Computing*. 2022. arXiv: 2210.05822 [hep-ex].

[6] C.S. Chang et al. "Simulations in the era of exascale computing". In: *Nature Reviews Materials* 8 (Mar. 2023). DOI: 10.1038/s41578-023-00540-6.

[7] US DOE. *Supercomputing and Exascale*. URL: https://www.energy.gov/supercomputing-and-exascale.

[8] top500. *Supercomputers – Prestige Objects or Crucial Tools for Science and Industry?* URL: https://www.top500.org/files/Supercomputers-Paper-London-Final.pdf.

[9] Juelich Forschungszentrum. *Supercomputer for Highly Complex Calculations and AI*. URL: https://www.fz-juelich.de/en/news/archive/feature-stories/supercomputer-for-highly-complex-calculations-and-ai.

[10] EuroHPC. *The European High Performance Computing Joint Undertaking (EuroHPC JU)*. URL: https://eurohpc-ju.europa.eu/supercomputers/our-supercomputers_en.

[11] J.V. Maizel. "Supercomputing in molecular biology: applications to sequence analysis". In: *IEEE Engineering in Medicine and Biology Magazine* 7.4 (1988), pp. 27–30. DOI: 10.1109/51.20377.

[12] National Research Council. *Getting Up to Speed: The Future of Supercomputing*. Ed. by Susan L. Graham, Marc Snir, and Cynthia A. Patterson. Washington, DC: The National Academies Press, 2005. ISBN: 978-0-309-09502-0. DOI: 10.17226/11148. URL: https://nap.nationalacademies.org/catalog/11148/getting-up-to-speed-the-future-of-supercomputing.

[13] ACM. *The decline of computers as a general purpose technology*. URL: https://dl.acm.org/doi/fullHtml/10.1145/3430936.

[14] European Investment Bank. *Financing the future of supercomputing: How to increase the investments in high performance computing in Europe*. URL: https://www.eib.org/en/publications/financing-the-future-of-supercomputing.

[15] top500. *top500 The List*. URL: https://www.top500.org/lists/top500/2023/06/.

[16] Microsoft. *What runs ChatGPT? Inside Microsoft's AI supercomputer*. URL: https://techcommunity.microsoft.com/t5/microsoft-mechanics-blog/what-runs-chatgpt-inside-microsoft-s-ai-supercomputer-featuring/ba-p/3830281.

[17] HPC wire. *US Plans 1.8Billion Spend on DOE Exascale Supercomputing*. URL: https://www.hpcwire.com/2018/04/11/us-plans-1-8-billion-spend-on-doe-exascale-supercomputing/.

[18] European Commission. *The EU enters the exascale era with the announcement of new supercomputing hosting sites*. URL: https://digital-strategy.ec.europa.eu/en/news/eu-enters-exascale-era-announcement-new-supercomputing-hosting-sites.

[19] HPC Wire. *Three Chinese Exascale Systems Detailed at SC21*. URL: https://www.hpcwire.com/2021/11/24/three-chinese-exascale-systems-detailed-at-sc21-two-operational-and-one-delayed/.

[20] DOE. *Launching a New Class of U.S. Supercomputing*. URL: https://www.energy.gov/science/articles/launching-new-class-us-supercomputing.

[21] US DOE. *The Opportunities and Challenges of Exascale Computing*. URL: https://science.osti.gov/-/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf.

[22] IEEE Spectrum. *FRONTIER SUPERCOMPUTER TO USHER IN EXASCALE COMPUTING*. URL: https://spectrum.ieee.org/exascale-supercomputing.

[23] Neha Gholkar, Frank Mueller, and Barry Rountree. "A Power-Aware Cost Model for HPC Procurement". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 1110–1113. DOI: 10.1109/IPDPSW.2016.35.

[24] Alexandros Nikolaos Ziogas et al. "A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3357156. URL: https://doi.org/10.1145/3295500.3357156.

[25] Daniel Dauwe et al. "An Analysis of Resilience Techniques for Exascale Computing Platforms". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2017, pp. 914–923. DOI: 10.1109/IPDPSW.2017.41.

[26] Hassan Khan, David Hounshell, and Erica Fuchs. "Science and research policy at the end of Moore's law". In: *Nature Electronics* 1 (Jan. 2018). DOI: 10.1038/s41928-017-0005-9.

[27] Dell Technologies. *SAVING THE FUTURE OF MOORE'S LAW*. URL: https://education.dell.com/content/dam/dell-emc/documents/en-us/2019KS_Yellin-Saving_The_Future_of_Moores_Law.pdf.

[28] R.R. Schaller. "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[29] Intel. *Moore's Law*. URL: https://www.intel.com/content/www/us/en/newsroom/resources/moores-law.html.

[30] Manish Parashar et al. "A Study of Software Development for High Performance Computing". In: (Mar. 1997). DOI: 10.1007/978-3-0348-8534-8_11.

[31] V Pallipadi and A Starikovskiy. "The ondemand governor: Past, present and future". In: *Proceedings of Linux Symposium* 2 (Jan. 2006), pp. 223–238.

[32] Milan Yadav. “A Brief Survey of Current Power Limiting Strategies”. In: (Mar. 2018).

[33] S. Kalaiselvi and Vaidy Rajaraman. “Survey of checkpointing algorithms for parallel and distributed computers”. In: *Sadhana* 25 (Apr. 2012), pp. 489–510. DOI: 10.1007/BF02703630.

[34] Dheya Mustafa. “A Survey of Performance Tuning Techniques and Tools for Parallel Applications”. In: *IEEE Access* 10 (2022), pp. 15036–15055. DOI: 10.1109/ACCESS.2022.3147846.

[35] Michael Gerndt et al. “Adaptive Resource Management for HPC Systems (Dagstuhl Seminar 21441)”. In: *Dagstuhl Reports* 11.10 (2022). Ed. by Michael Gerndt et al., pp. 1–19. ISSN: 2192-5283. DOI: 10.4230/DagRep.11.10.1. URL: https://drops.dagstuhl.de/opus/volltexte/2022/15925.

[36] Jose I. Aliaga et al. “A Survey on Malleability Solutions for High-Performance Distributed Computing”. In: *Applied Sciences* 12.10 (2022). ISSN: 2076-3417. DOI: 10.3390/app12105231. URL: https://www.mdpi.com/2076-3417/12/10/5231.

[37] Deep Projects. *Programming Environment for European Exascale Systems*. https://www.deep-projects.eu/.

[38] Mohak Chadha. “Adaptive Resource-Aware Batch Scheduling for HPC systems”. In: ().

[39] Joseph Schuchart et al. “The READEX formalism for automatic tuning for energy efficiency”. In: *Computing* 99.8 (2017), pp. 727–745. ISSN: 1436-5057. DOI: 10.1007/s00607-016-0532-7. URL: https://doi.org/10.1007/s00607-016-0532-7.

[40] SchedMD. *Multifactor Priority Plugin*. URL: https://slurm.schedmd.com/priority_multifactor.html.

[41] *Slurm Workload Manager*. URL: https://slurm.schedmd.com/sched_config.html.

[42] Xu Yang et al. “Integrating Dynamic Pricing of Electricity into Energy Aware Scheduling for HPC Systems”. In: SC ’13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503264. URL: https://doi.org/10.1145/2503210.2503264.

[43] Bianca Schroeder and Garth Gibson. “Understanding failures in petascale computers”. In: *Journal of Physics: Conference Series* 78 (Sept. 2007). DOI: 10.1088/1742-6596/78/1/012022.

[44] Michael A. Heroux. “Software Challenges for Extreme Scale Computing: Going From Petascale to Exascale Systems”. In: *The International Journal of High Performance Computing Applications* 23.4 (2009), pp. 437–439. DOI: 10.1177/1094342009347711. eprint: https://doi.org/10.1177/1094342009347711. URL: https://doi.org/10.1177/1094342009347711.

[45] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. “The MPI Message Passing Interface Standard”. In: *Programming Environments for Massively Parallel Distributed Systems*. Ed. by Karsten M. Decker and René M. Rehmann. Basel: Birkhäuser Basel, 1994, pp. 213–218. ISBN: 978-3-0348-8534-8.

[46] M. Gerndt I. Compres Urena. “Towards Elastic Resource Management”. In: *Proceedings of the 11th Parallel Tools Workshop, September 11-12, 2017*. to appear. 2017.

[47] Miao Chen, Fang Dong, and Junzhou Luo. “Dynamic Resource Management in a HPC and Cloud Hybrid Environment”. In: Dec. 2013, pp. 206–215. ISBN: 978-3-319-03858-2. DOI: 10.1007/978-3-319-03859-9_17.

[48] David Bailey, Robert Lucas, and Samuel Williams. *Performance Tuning of Scientific Applications*. Nov. 2010. ISBN: 9780429149900. DOI: 10.1201/b10509.

[49] Intel. *Our Future with Hierarchical Heterogeneous Computing*. URL: https://community.intel.com/t5/Blogs/Products-and-Solutions/HPC/Our-Future-with-Hierarchical-Heterogeneous-Computing/post/1495073.

[50] Andy B Yoo, Morris A Jette, and Mark Grondona. “Slurm: Simple linux utility for resource management”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[51] Dong H Ahn et al. “Flux: a next-generation resource management framework for large HPC centers”. In: *2014 43rd International Conference on Parallel Processing Workshops*. IEEE. 2014, pp. 9–17.

[52] S. Kannan et al. “Workload Management with LoadLeveler”. In: *IBM Redbooks* 2 (Jan. 2001).

[53] Adaptive Computing. *TORQUE Resource Manager*. URL: https://adaptivecomputing.com/cherry-services/torque-resource-manager/.

[54] Invasive Computing. *Welcome to the pages of the TCRC 89 "Invasive Computing" (InvasIC)*. URL: https://invasic.informatik.uni-erlangen.de/en/index.php.

[55] Mohak Chadha, Jophin John, and Michael Gerndt. “Extending SLURM for Dynamic Resource-Aware Adaptive Batch Scheduling”. In: *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2020, pp. 223–232. DOI: 10.1109/HiPC50609.2020.00036.

[56] *SuperMUC-NG*. URL: https://doku.lrz.de/display/PUBLIC/SuperMUC-NG.

[57] SchedMD. *Backfill Scheduling*. URL: https://slurm.schedmd.com/sched_config.html#backfill.

[58] Santiago Narvaez. “Power model for resource-elastic applications”. Master Thesis. Munich: TU Munich, 2018. URL: http://mediatum.ub.tum.de?id=1475095.

[59] Jophin John, Santiago Narváez, and Michael Gerndt. “Invasive computing for power corridor management”. In: *Parallel Computing: Technology Trends* 36 (2020), p. 386.

[60] Wesley Bland et al. “Post-failure recovery of MPI communication capability: Design and rationale”. In: *The International Journal of High Performance Computing Applications* 27.3 (2013), pp. 244–254. DOI: 10.1177/1094342013488238, eprint: https://doi.org/10.1177/1094342013488238. URL: https://doi.org/10.1177/1094342013488238.

[61] Jon Stearley et al. “rMPI : increasing fault resiliency in a message-passing environment.” In: (Apr. 2011). DOI: 10.2172/1012733.

[62] Gengbin Zheng, Lixia Shi, and L. V. Kale. “FTC-Charm++: An in-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI”. In: *Proceedings of the 2004 IEEE International Conference on Cluster Computing*. CLUSTER ’04. USA: IEEE Computer Society, 2004, 93–103. ISBN: 0780386949.

[63] Graham E. Fagg and Jack Dongarra. “FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World”. In: *Proceedings of the 7th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2000, 346–353. ISBN: 3540410104.

[64] Manoj Kumar, Abhishek Choudhary, and Vikas Kumar. “A Comparison between Different Checkpoint Schemes with Advantages and Disadvantages”. In: 2014.

[65] Andrew Lumsdaine and Joshua Hursey. “Coordinated checkpoint/restart process fault tolerance for mpi applications on hpc systems”. In: 2010.

[66] Kento Sato et al. “Design and modeling of a non-blocking checkpointing system”. In: Nov. 2012, pp. 1–10. ISBN: 978-1-4673-0805-2. DOI: 10.1109/SC.2012.46.

[67] *JSC*. URL: https://hbp-hpc-platform.fz-juelich.de/?page_id=1836.

[68] Travis Desell, Kaoutar Maghraoui, and Carlos Varela. "Malleable applications for scalable high performance computing". In: *Cluster Computing* 10 (Sept. 2007), pp. 323–337. DOI: 10.1007/s10586-007-0032-9.

[69] Manish Abhishek. "Dynamic Allocation of High Performance Computing Resources". In: *International Journal of Advanced Trends in Computer Science and Engineering* 9 (June 2020), pp. 3538–3543. DOI: 10.30534/ijatcse/2020/159932020.

[70] Cijo George and Sathish S. Vadhiyar. "ADFT: An Adaptive Framework for Fault Tolerance on Large Scale Systems using Application Malleability". In: *Procedia Computer Science* 9 (2012). Proceedings of the International Conference on Computational Science, ICCS 2012, pp. 166–175. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2012.04.018. URL: https://www.sciencedirect.com/science/article/pii/S1877050912001391.

[71] Kaoutar El Maghraoui, Boleslaw K. Szymanski, and Carlos Varela. "An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments". In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 258–271. ISBN: 978-3-540-34142-0.

[72] Jonas Posner. "Load Balancing, Fault Tolerance, and Resource Elasticity for Asynchronous Many-Task Systems". PhD thesis. Kassel, Universität Kassel, Fachbereich Elektrotechnik / Informatiik, Dec. 2021.

[73] Isaías A. Comprés Ureña and Michael Gerndt. "Towards Elastic Resource Management". In: *Tools for High Performance Computing 2017*. Ed. by Christoph Niethammer et al. Cham: Springer International Publishing, 2019, pp. 105–127. ISBN: 978-3-030-11987-4.

[74] Isaías Alberto Comprés Ureña. "Resource-Elasticity Support for Distributed Memory HPC Applications". Dissertation. Munich: TU Munich, 2017. URL: http://mediatum.ub.tum.de?id=1362721.

[75] Laxmikant V Kalé, Sameer Kumar, and Jayant DeSouza. "A malleable-job system for timeshared parallel machines". In: *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*. IEEE. 2002, pp. 230–230.

[76] admire. *Adaptive Multi-Tier Intelligent Data Manager for Exascale*. https://research.zdv.uni-mainz.de/research/admire/.

[77] Jophin John, Isaac David Núñez Araya, and Michael Gerndt. "iCheck: Leveraging RDMA and Malleability for Application-Level Checkpointing in HPC Systems". In: *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*. 2023, pp. 467–474. DOI: 10.1109/ICPADS56603.2022.00067.

[78] Jophin John and Michael Gerndt. *Designing an Adaptive Application-Level Checkpoint Management System for Malleable MPI Applications*. 2022. arXiv: 2211.04305 [cs.DC].

[79] DFG. *DFG Official website*. URL: https://www.dfg.de/en/research_funding/index.html.

[80] Jürgen Teich. "Invasive Algorithms and Architectures Invasive Algorithmen und Architekturen". In: *it-Information Technology* 50.5 (2008), pp. 300–310.

[81] Invasive Computing. *Projects of the TCRC 89 "Invasive Computing" (InvasIC)*. URL: https://invasic.informatik.uni-erlangen.de/en/projects_PhIII.php.

[82] FAU Erlangen. *FAU Erlangen Webpage*. URL: https://www.fau.eu/.

[83] Karlsruhe Institute of Technology. *Karlsruhe Institute of Technology Webpage*. URL: https://www.kit.edu/.

[84] Technical University of Munich. *Technical University of Munich Webpage*. URL: https://www.tum.de/en/.

[85] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[86] ODU Research Cloud Computing. *Slurm Job Scheduler*. URL: https://wiki.hpc.odu.edu/en/slurm.

[87] SchedMD. *Quick Start User Guide*. URL: https://slurm.schedmd.com/quickstart.html.

[88] Jophin John. "The Elastic Phase Oriented Programming Model for Elastic HPC Applications". In: 2018.

[89] HPC Wiki. *MPI*. URL: https://hpc-wiki.info/hpc/MPI.

[90] Intel. *Intel MPI Library*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html.

[91] IBM. *IBM Spectrum MPI*. URL: https://www.ibm.com/products/spectrum-mpi.

[92] MPI Forum. *MPI Sessions*. URL: https://www.mpi-forum.org/bofs/2022-05-mpi-bof.pdf.

[93] Rajesh K. Karmani et al. "Amdahl's Law". In: *Encyclopedia of Parallel Computing*. Springer US, 2011, pp. 53–60. DOI: 10.1007/978-0-387-09766-4_77. URL: https://doi.org/10.1007%2F978-0-387-09766-4_77.

[94] Chao Huang, Orion Lawlor, and Laxmikant V Kale. "Adaptive mpi". In: *International workshop on languages and compilers for parallel computing*. Springer. 2003, pp. 306–322.

[95] Chao Huang, Gengbin Zheng, and Laxmikant V Kalé. "Supporting adaptivity in MPI for dynamic parallel applications". In: *Rapport technique, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign* 14 (2007).

[96] Chao Huang et al. "Performance evaluation of adaptive MPI". In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2006, pp. 12–21.

[97] Laxmikant V Kale and Sanjeev Krishnan. "Charm++ A portable concurrent object oriented system based on C++". In: *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. 1993, pp. 91–108.

[98] Bilge Acun et al. "Parallel programming with migratable objects: Charm++ in practice". In: *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 647–658.

[99] Gengbin Zheng et al. "Hierarchical load balancing for charm++ applications on large supercomputers". In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE. 2010, pp. 436–444.

[100] Sayantan Chakravorty, Celso L Mendes, and Laxmikant V Kalé. "Proactive fault tolerance in MPI applications via task migration". In: *International Conference on High-Performance Computing*. Springer. 2006, pp. 485–496.

[101] Comprés Ureña and Isaías Alberto. "Resource-Elasticity Support for Distributed Memory HPC Applications". PhD thesis. Technische Universität München, 2017.

[102] Isaías Comprés et al. "Infrastructure and api extensions for elastic execution of mpi applications". In: *Proceedings of the 23rd European MPI Users' Group Meeting*. 2016, pp. 82–97.

[103] Gonzalo Martín et al. “Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration”. In: *Parallel Computing* 46 (2015), pp. 60–77.

[104] R. Sudarsan and C. J. Ribbens. “ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment”. In: *2007 International Conference on Parallel Processing (ICPP 2007)*. 2007, pp. 44–44. DOI: 10.1109/ICPP.2007.73.

[105] Yu-Kwong Kwok and Ishfaq Ahmad. “Static scheduling algorithms for allocating directed task graphs to multiprocessors”. In: *ACM Computing Surveys (CSUR)* 31.4 (1999), pp. 406–471.

[106] Kirk Pruhs. “Competitive online scheduling for server systems”. In: *ACM SIGMETRICS Performance Evaluation Review* 34.4 (2007), pp. 52–58.

[107] Dror G Feitelson and Larry Rudolph. “Toward convergence in job schedulers for parallel supercomputers”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1996, pp. 1–26.

[108] Tomasz Plewa, Timur Linde, and V Gregory Weirs. “Adaptive mesh refinement-theory and applications”. In: ().

[109] Abhishek Gupta et al. “Towards realizing the potential of malleable jobs”. In: *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE. 2014, pp. 1–10.

[110] Jan Hungershofer. “On the combined scheduling of malleable and rigid jobs”. In: *16th Symposium on Computer Architecture and High Performance Computing*. IEEE. 2004, pp. 206–213.

[111] T. E. Carroll and D. Grosu. “Incentive Compatible Online Scheduling of Malleable Parallel Jobs with Individual Deadlines”. In: *2010 39th International Conference on Parallel Processing*. 2010, pp. 516–524. DOI: 10.1109/ICPP.2010.60.

[112] Hongyang Sun, Yangjie Cao, and Wen-Jing Hsu. “Fair and efficient online adaptive scheduling for multiple sets of parallel applications”. In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE. 2011, pp. 64–71.

[113] Gladys Utrera et al. “A job scheduling approach for multi-core clusters based on virtual malleability”. In: *European Conference on Parallel Processing*. Springer. 2012, pp. 191–203.

[114] Suraj Prabhakaran et al. “A batch system with fair scheduling for evolving applications”. In: *2014 43rd International Conference on Parallel Processing*. IEEE. 2014, pp. 351–360.

[115] Suraj Prabhakaran et al. “A batch system with efficient adaptive scheduling for malleable and evolving applications”. In: *2015 IEEE international parallel and distributed processing symposium*. IEEE. 2015, pp. 429–438.

[116] Adrian T Wong et al. “ESP: A system utilization benchmark”. In: *SC’00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE. 2000, pp. 15–15.

[117] Ozan Sonmez et al. “Scheduling malleable applications in multicluster systems”. In: *2007 IEEE International Conference on Cluster Computing*. IEEE. 2007, pp. 372–381.

[118] H. H. Mohamed and D. H. J. Epema. “The Design and Implementation of the KOALA Co-allocating Grid Scheduler”. In: *Advances in Grid Computing - EGC 2005*. Springer Berlin Heidelberg, 2005, pp. 640–650. DOI: 10.1007/11508380_65. URL: https://doi.org/10.1007%2F11508380_65.

[119] Jack Dongarra. “HPC: Where we are today and a look into the future”. In: *Parallel Processing and Applied Mathematics, PPAM: Gdansk, Poland* (2022).

[120] Pawel Czarnul, Jerzy Proficz, Adam Krzywaniak, et al. “Energy-aware high-performance computing: survey of state-of-the-art tools, techniques, and environments”. In: *Scientific Programming* 2019 (2019).

[121] Matthias Maiterth et al. “Energy and Power Aware Job Scheduling and Resource Management: Global Survey — Initial Analysis”. In: May 2018, pp. 685–693.

[122] Bartłomiej Kocot, Paweł Czarnul, and Jerzy Proficz. “Energy-Aware Scheduling for High-Performance Computing Systems: A Survey”. In: *Energies* 16.2 (2023). ISSN: 1996-1073. DOI: 10.3390/en16020890. URL: https://www.mdpi.com/1996-1073/16/2/890.

[123] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. “A survey of design techniques for system-level dynamic power management”. In: *IEEE transactions on very large scale integration (VLSI) systems* 8.3 (2000), pp. 299–316.

[124] Monire Safari and Reihaneh Khorsand. “Energy-aware scheduling algorithm for time-constrained workflow tasks in DVFS-enabled cloud environment”. In: *Simulation Modelling Practice and Theory* 87 (2018), pp. 311–326.

[125] Kazuki Tsuzuku and Toshio Endo. “Power capping of cpu-gpu heterogeneous systems using power and performance models”. In: *2015 International Conference on Smart Cities and Green ICT Systems (SMARTGREENS)*. IEEE. 2015, pp. 1–8.

[126] Pavlos Petoumenos et al. “Power capping: What works, what does not”. In: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2015, pp. 525–534.

[127] Andrea Borghesi et al. “Power capping in high performance computing systems”. In: *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*. Springer. 2015, pp. 524–540.

[128] Adam Krzywaniak and Paweł Czarnul. “Performance/energy aware optimization of parallel applications on gpus under power capping”. In: *Parallel Processing and Applied Mathematics: 13th International Conference, PPAM 2019, Bialystok, Poland, September 8–11, 2019, Revised Selected Papers, Part II 13*. Springer. 2020, pp. 123–133.

[129] Howard David et al. “RAPL: Memory Power Estimation and Capping”. In: *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED ’10. Austin, Texas, USA: Association for Computing Machinery, 2010, 189–194. ISBN: 9781450301466. DOI: 10.1145/1840845.1840883. URL: https://doi.org/10.1145/1840845.1840883.

[130] NVIDIA. *System Management Interface SMI*. URL: https://developer.nvidia.com/nvidia-system-management-interface.

[131] H.-Y McCreary et al. “Energyscale for IBM POWER6 microprocessor-based systems”. In: *IBM Journal of Research and Development* 51 (Dec. 2007), pp. 775 –786. DOI: 10.1147/rd.516.0775.

[132] J. Sun, C. Huang, and J. Dong. “Research on Power-Aware Scheduling for High-Performance Computing System”. In: *2011 IEEE/ACM International Conference on Green Computing and Communications*. 2011, pp. 75–78.

[133] P. Dutot et al. “Towards Energy Budget Control in HPC”. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2017, pp. 381–390.

[134] D. Bodas et al. “Simple Power-Aware Scheduler to Limit Power Consumption by HPC System within a Budget”. In: *2014 Energy Efficient Supercomputing Workshop*. 2014, pp. 21–30.

[135] Y. Liu et al. “FastCap: An efficient and fair algorithm for power capping in many-core systems”. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016, pp. 57–68.

[136] M. Chadha et al. “A Statistical Approach to Power Estimation for x86 Processors”. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2017, pp. 1012–1019. DOI: 10.1109/IPDPSW.2017.98.

[137] M. Chadha and M. Gerndt. "Modelling DVFS and UFS for Region-Based Energy Aware Tuning of HPC Applications". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 805–814.

[138] Ying Li et al. "Energy-aware scheduling on heterogeneous multi-core systems with guaranteed probability". In: *Journal of Parallel and Distributed Computing* 103 (2017), pp. 64–76.

[139] Chunrong Yao et al. "EAIS: Energy-aware adaptive scheduling for CNN inference on high-performance GPUs". In: *Future Generation Computer Systems* 130 (2022), pp. 253–268.

[140] Marco D'Amico and Julita Corbalan Gonzalez. "Energy hardware and workload aware job scheduling towards interconnected HPC environments". In: *IEEE Transactions on Parallel and Distributed Systems* (2021).

[141] Xiaoyong Tang and Zhuojun Fu. "CPU–GPU Utilization Aware Energy-Efficient Scheduling Algorithm on Heterogeneous Computing Systems". In: *IEEE Access* 8 (2020), pp. 58948–58958. DOI: 10.1109/ACCESS.2020.2982956.

[142] Jian Chen and Lizy K John. "Energy-aware application scheduling on a heterogeneous multi-core system". In: *2008 IEEE International Symposium on Workload Characterization*. IEEE. 2008, pp. 5–13.

[143] Isabel Méndez-Díaz et al. "Energy-aware scheduling mandatory/optional tasks in multicore real-time systems". In: *International Transactions in Operational Research* 24.1-2 (2017), pp. 173–198.

[144] Ayham Kassab et al. "Assessing the use of genetic algorithms to schedule independent tasks under power constraints". In: *2018 International conference on high performance computing & simulation (HPCS)*. IEEE. 2018, pp. 252–259.

[145] Mateusz Guzek et al. "Multi-objective evolutionary algorithms for energy-aware scheduling on distributed computing systems". In: *Applied Soft Computing* 24 (2014), pp. 432–446.

[146] Pragati Agrawal and Shrisha Rao. "Energy-aware scheduling of distributed systems". In: *IEEE Transactions on Automation Science and Engineering* 11.4 (2014), pp. 1163–1175.

[147] Andrea Borghesi et al. "Scheduling-based power capping in high performance computing systems". In: *Sustainable Computing: Informatics and Systems* 19 (2018), pp. 1–13.

[148] Olli Mämmelä et al. "Energy-aware job scheduler for high-performance computing". In: *Computer Science-Research and Development* 27 (2012), pp. 265–275.

[149] Yu Li, Yi Liu, and Depei Qian. "A heuristic energy-aware scheduling algorithm for heterogeneous clusters". In: *2009 15th International Conference on Parallel and Distributed Systems*. IEEE. 2009, pp. 407–413.

[150] Muthucumaru Maheswaran et al. "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems". In: *Journal of parallel and distributed computing* 59.2 (1999), pp. 107–131.

[151] Tarun Biswas, Pratyay Kuila, and Anjan Kumar Ray. "A novel energy efficient scheduling for high performance computing systems". In: *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*. IEEE. 2018, pp. 1–6.

[152] Jing Mei and Kenli Li. "Energy-Aware Scheduling Algorithm with Duplication on Heterogeneous Computing Systems". In: *2012 ACM/IEEE 13th International Conference on Grid Computing*. 2012, pp. 122–129. DOI: 10.1109/Grid.2012.32.

[153] Julius Roeder et al. “Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems”. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021, pp. 501–510.

[154] Xinxin Mei et al. “Energy-aware task scheduling with deadline constraint in dvfs-enabled heterogeneous clusters”. In: *arXiv preprint arXiv:2104.00486* (2021).

[155] Lena Mashayekhy et al. “Energy-aware scheduling of mapreduce jobs”. In: *2014 IEEE International Congress on Big Data*. IEEE. 2014, pp. 32–39.

[156] Yikun Hu et al. “Slack allocation algorithm for energy minimization in cluster systems”. In: *Future Generation Computer Systems* 74 (2017), pp. 119–131.

[157] Venkateswaran Shekar and Baback Izadi. “Energy aware scheduling for DAG structured applications on heterogeneous and DVS enabled processors”. In: *International Conference on Green Computing*. IEEE. 2010, pp. 495–502.

[158] HV Raghu, Sumit Kumar Saurav, and Bindhumadhava S Bapu. “PAAS: Power aware algorithm for scheduling in high performance computing”. In: *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE. 2013, pp. 327–332.

[159] Intel. *Building for Zettascale with Intel and the Cambridge Open Zettascale Lab*. URL: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-09/intel-cambridge-zettascale-lab.pdf.

[160] Faisal Shahzad et al. “A Survey of Checkpoint/Restart Techniques on Distributed Memory Systems”. In: *Parallel Process. Lett.* 23 (2013). URL: https://api.semanticscholar.org/CorpusID:44999017.

[161] Franck Cappello. “Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities”. In: *Int. J. High Perform. Comput. Appl.* 23.3 (2009), 212–226. ISSN: 1094-3420. DOI: 10.1177/1094342009106189. URL: https://doi.org/10.1177/1094342009106189.

[162] Ifeanyi P. Egwutuoha et al. “A Survey of Fault Tolerance Mechanisms and Checkpoint/Restart Implementations for High Performance Computing Systems”. In: *J. Supercomput.* 65.3 (2013), 1302–1326. ISSN: 0920-8542. DOI: 10.1007/s11227-013-0884-0. URL: https://doi.org/10.1007/s11227-013-0884-0.

[163] Mirza Mohammed Akram Baig. “An Evaluation of Major Fault Tolerance Techniques Used on High Performance Computing (HPC) Applications”. In: *International Journal of Intelligent Systems and Applications in Engineering* 11 (2023), 320–328. URL: https://ijisae.org/index.php/IJISAE/article/view/2696.

[164] Israel Koren and C Mani Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2020.

[165] Wendy Bartlett and Lisa Spainhower. “Commercial fault tolerance: A tale of two systems”. In: *IEEE Transactions on dependable and secure computing* 1.1 (2004), pp. 87–96.

[166] Jean-Claude Laprie et al. “Definition and analysis of hardware-and-software fault-tolerant architectures”. In: *Predictably Dependable Computing Systems*. Springer. 1995, pp. 103–122.

[167] D Milojicic et al. “Process Migration, Technical Report”. In: *TOG Research Institute* (1996).

[168] Christopher Clark et al. “Live migration of virtual machines”. In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 2005, pp. 273–286.

[169] Flavin Cristian. “Understanding Fault-Tolerant Distributed Systems”. In: *Commun. ACM* 34.2 (1991), 56–78. ISSN: 0001-0782. DOI: 10.1145/102792.102801. URL: https://doi.org/10.1145/102792.102801.

[170] Flavin Cristian. “Understanding fault-tolerant distributed systems”. In: *Communications of the ACM* 34.2 (1991), pp. 56–78.

[171] Jasim A Ghaeb, Mahmoud A Smadi, and Jalel Chebil. “A high performance data integrity assurance based on the determinant technique”. In: *Future Generation Computer Systems* 27.5 (2011), pp. 614–619.

[172] J.C. Sancho et al. “Current practice and a direction forward in checkpoint/restart implementations for fault tolerance”. In: *19th IEEE International Parallel and Distributed Processing Symposium*. 2005, 8 pp.–. DOI: 10.1109/IPDPS.2005.157.

[173] 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*. IEEE. 2009, pp. 1–12.

[174] J. Duell, Paul Hargrove, and Eric Roman. “The Design and Implementation of Berkeley Lab”s Linux Checkpoint/Restart”. In: (Jan. 2003).

[175] Gabriel Rodríguez et al. “CPPC: A Compiler-assisted tool for portable checkpointing of message-passing applications”. In: *Concurrency and Computation: Practice and Experience* 22 (Apr. 2010), pp. 749–766. DOI: 10.1002/cpe.1541.

[176] Sourav Chakraborty et al. “EReinit: Scalable and efficient fault-tolerance for bulk-synchronous MPI applications”. In: *Concurrency and Computation: Practice and Experience* 32.3 (2020). e4863 cpe.4863, e4863. DOI: https://doi.org/10.1002/cpe.4863. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4863. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4863.

[177] Nawrin Sultana et al. “MPI Stages: Checkpointing MPI State for Bulk Synchronous Applications”. In: Sept. 2018, pp. 1–11. DOI: 10.1145/3236367.3236385.

[178] Balkrishna Ramkumar and Volker Strumpen. “Portable Checkpointing for Heterogeneous Architectures”. In: July 1997, pp. 58 –67. ISBN: 0-8186-7831-3. DOI: 10.1109/FTCS.1997.614078.

[179] Ritu Arora, Purushotham Bangalore, and Marjan Mernik. “A technique for non-invasive application-level checkpointing”. In: *The Journal of Supercomputing* 57 (Sept. 2010), pp. 227–255. DOI: 10.1007/s11227-010-0383-5.

[180] Ritu Arora and Trung Nguyen Ba. “ITALC: Interactive Tool for Application-Level Checkpointing”. In: *Proceedings of the Fourth International Workshop on HPC User Support Tools*. HUST’17. Denver, CO, USA: Association for Computing Machinery, 2017. ISBN: 9781450351300. DOI: 10.1145/3152493.3152558. URL: https://doi.org/10.1145/3152493.3152558.

[181] James Plank et al. “Libckpt: Transparent checkpointing under UNIX”. In: *TCON* (Dec. 1995).

[182] Tanzima Islam et al. “McrEngine: A Scalable Checkpointing System Using Data-Aware Aggregation and Compression”. In: *Scientific Programming* 21 (Jan. 2013). DOI: 10.3233/SPR-130371.

[183] Kento Sato et al. “A User-Level InfiniBand-Based File System and Checkpoint Strategy for Burst Buffers”. In: May 2014, pp. 21–30. ISBN: 978-1-4799-2784-5. DOI: 10.1109/CCGrid.2014.24.

[184] Dries Kimpe et al. “Integrated in-system storage architecture for high performance computing”. In: *ROSS ’12*. 2012.

[185] Bogdan Nicolae et al. “VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale”. In: May 2019, pp. 911–920. DOI: 10.1109/IPDPS.2019.00099.

[186] Faisal Shahzad et al. “CRAFT: A Library for Easier Application-Level Checkpoint/Restart and Automatic Fault Tolerance”. In: *IEEE Transactions on Parallel and Distributed Systems* PP (Aug. 2017). DOI: 10.1109/tpds.2018.2866794.

[187] Marc Gamell et al. “Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* 2015 (Jan. 2015), pp. 895–906. DOI: 10.1109/SC.2014.78.

[188] Adam Moody et al. “Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System”. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’10. USA: IEEE Computer Society, 2010, 1–11. ISBN: 9781424475599. DOI: 10.1109/SC.2010.18. URL: https://doi.org/10.1109/SC.2010.18.

[189] Josef Weidendorfer, Dai Yang, and Carsten Trinitis. “Laik: A library for fault tolerant distribution of global data for parallel applications”. In: *PARS-Mitteilungen: Vol. 34, Nr. 1* (2017).

[190] Lubomír Říha, Tomáš Brzobohatỳ, and Alexandros Markopoulos. “Hybrid parallelization of the total FETI solver”. In: *Advances in Engineering Software* 103 (2017), pp. 29–37.

[191] Arturo Gonzalez-Escribano et al. “An Extensible System for Multilevel Automatic Data Partition and Mapping”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.5 (2014), pp. 1145–1154. DOI: 10.1109/TPDS.2013.83.

[192] Ian Karlin. *Lulesh programming model and performance ports overview*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2012.

[193] Anders Clausen et al. “An Analysis of Contracts and Relationships between Supercomputing Centers and Electricity Service Providers”. In: *Workshop Proceedings of the 48th International Conference on Parallel Processing*. ICPP Workshops ’19. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450371964. DOI: 10.1145/3339186.3339209. URL: https://doi.org/10.1145/3339186.3339209.

[194] Natalie Bates et al. “Electrical Grid and Supercomputing Centers: An Investigative Analysis of Emerging Opportunities and Challenges”. In: *Informatik-Spektrum* 38.2 (Apr. 2015), pp. 111–127. ISSN: 1432-122X. DOI: 10.1007/s00287-014-0850-0. URL: https://doi.org/10.1007/s00287-014-0850-0.

[195] Jonathan Muraña et al. “Characterization, Modeling and Scheduling of Power Consumption of Scientific Computing Applications in Multicores”. In: *Cluster Computing* 22.3 (2019), 839–859. ISSN: 1386-7857. DOI: 10.1007/s10586-018-2882-8. URL: https://doi.org/10.1007/s10586-018-2882-8.

[196] Kashif Nizam Khan et al. “Rapl in action: Experiences in using rapl for power measurements”. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3.2 (2018), p. 9.

[197] Jan Treibig, Georg Hager, and Gerhard Wellein. “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments”. In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE. 2010, pp. 207–216.

[198] A. C. Harvey. “ARIMA Models”. In: *Time Series and Statistics*. Ed. by John Eatwell, Murray Milgate, and Peter Newman. London: Palgrave Macmillan UK, 1990, pp. 22–24. ISBN: 978-1-349-20865-4. DOI: 10.1007/978-1-349-20865-4_2. URL: https://doi.org/10.1007/978-1-349-20865-4_2.

[199] Joos Korstanje. "The SARIMAX Model". In: *Advanced Forecasting with Python: With State-of-the-Art-Models Including LSTMs, Facebook's Prophet, and Amazon's DeepAR*. Berkeley, CA: Apress, 2021, pp. 125–131. ISBN: 978-1-4842-7150-6. DOI: 10.1007/978-1-4842-7150-6_8. URL: https://doi.org/10.1007/978-1-4842-7150-6_8.

[200] Prajakta S. Kalekar. "Time series Forecasting using Holt-Winters Exponential Smoothing". In: 2004.

[201] Stuart Mitchell, Michael OSullivan, and Iain Dunning. "PuLP: a linear programming toolkit for python". In: *The University of Auckland, Auckland, New Zealand* (2011).

[202] H. Shah et al. *RFC 7306: Remote Direct Memory Access (RDMA) Protocol Extensions*. USA, 2014.

[203] IETF. *TRANSMISSION CONTROL PROTOCOL*. URL: https://www.ietf.org/rfc/rfc793.txt.

[204] Apratim Purakayastha and Carla Schlatter Ellis. "Characterizing and Optimizing Parallel File Systems". AAI9701265. PhD thesis. USA, 1996. ISBN: 0591075210.

[205] *Libfabric library*. URL: https://github.com/ofiwg/libfabric.

[206] OpenFabrics Alliance. *OpenFabrics Interfaces*. URL: https://www.openfabrics.org/openfabrics-interfaces/.

[207] Intel. *High Performance Fabrics by Intel*. URL: https://www.intel.com/content/www/us/en/products/network-io/high-performance-fabrics.html.

[208] ofiwg. $fi_endpoint - Fabric end point operations$. URL: https://ofiwg.github.io/libfabric/v1.1.1/man/fi_endpoint.3.html.

[209] Dewan Ibtesham, Kurt B Ferreira, and Dorian Arnold. "A checkpoint compression study for high-performance computing systems". In: *The International Journal of High Performance Computing Applications* 29.4 (2015), pp. 387–402. DOI: 10.1177/1094342015570921. eprint: https://doi.org/10.1177/1094342015570921. URL: https://doi.org/10.1177/1094342015570921.

[210] perf wiki. *perf: Linux profiling with performance counters*. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[211] *SuperMUC*. URL: https://www.lrz.de/services/compute/museum/supermuc/systemdescription/.

[212] Lenovo. *NeXtScale System M5 with Water Cool Technology*. URL: https://lenovopress.lenovo.com/lp0544-nextscale-water-cool-e5-2600-v4.

[213] Intel. *Intel Xeon Prozessor E5-2697 v3*. URL: https://ark.intel.com/content/www/de/de/ark/products/81059/intel-xeon-processor-e52697-v3-35m-cache-2-60-ghz.html.

[214] IBM. *Introducing General Parallel File System*. URL: https://www.ibm.com/docs/en/gpfs/4.1.0.4?topic=guide-introducing-general-parallel-file-system.

[215] Leibniz-Rechenzentrum. *SuperMUC Petascale System*. https://www.lrz.de/services/compute/supermuc/.

[216] wikichip. *Skylake SP - Cores - Intel*. URL: https://en.wikichip.org/wiki/intel/cores/skylake_sp.

[217] Intel. *Intel Xeon Platinum 8174 Processor*. URL: https://www.intel.com/content/www/us/en/products/sku/136874/intel-xeon-platinum-8174-processor-33m-cache-3-10-ghz/specifications.html.

[218] Ansible. *Automation for everyone*. URL: https://www.ansible.com/.

[219] Terraform. *Automate infrastructure on any cloud with Terraform*. URL: https://www.terraform.io/.

[220] Kubernetes. *Kubernetes*. URL: https://kubernetes.io/.

[221] Suraj Prabhakaran. "Dynamic Resource Management and Job Scheduling for High Performance Computing". en. PhD thesis. Darmstadt: Technische Universität Darmstadt, 2016. URL: http://tuprints.ulb.tu-darmstadt.de/5720/.

[222] Linux man page. *munge(7) - Linux man page*. URL: https://linux.die.net/man/7/munge.

[223] William TC Kramer. "Percu: A holistic method for evaluating high performance computing systems". PhD thesis. University of California, Berkeley, 2008.

[224] Yiannis Georgiou. "Contributions for Resource and Job Management in High Performance Computing". PhD thesis. Nov. 2010.

[225] Ao Mo-Hellenbrand et al. "A large-scale malleable tsunami simulation realized on an elastic MPI infrastructure". In: *Proceedings of the Computing Frontiers Conference*. 2017, pp. 271–274.

[226] Wikipedia. *Laplace's equation*. URL: https://en.wikipedia.org/wiki/Laplace%27s_equation.

[227] Wikipedia. *Jacobi method*. URL: https://en.wikipedia.org/wiki/Jacobi_method.

[228] Wikipedia. *LU decomposition*. URL: https://en.wikipedia.org/wiki/LU_decomposition.

[229] Wikipedia. *Leibniz formula for*. URL: https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80.

[230] Christoph Niethammer et al. "ls1 mardyn: The Massively Parallel Molecular Dynamics Code for Large Systems". In: *Journal of Chemical Theory and Computation* 10.10 (2014). PMID: 26588142, pp. 4455–4464. DOI: 10.1021/ct500169q.

[231] Ian Karlin, Jeff Keasler, and Rob Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Livermore, CA, 2013, pp. 1–9.

[232] Spyros Makridakis et al. "The accuracy of extrapolation (time series) methods: Results of a forecasting competition". In: *Journal of forecasting* 1.2 (1982), pp. 111–153.

[233] American National Standards Institute. *IEEE standard for information technology: Portable Operating System Interface (POSIX) : part 2, shell and utilities*. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1993, pp. xvii + 1195. ISBN: 1-55937-255-9.

[234] Hongzhang Shan, Katie Antypas, and John Shalf. "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008. ISBN: 9781424428359.

[235] Marco A. S. Netto et al. "HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges". In: *ACM Comput. Surv.* 51.1 (2018). ISSN: 0360-0300. DOI: 10.1145/3150224. URL: https://doi.org/10.1145/3150224.

[236] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (2009), 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi.org/10.1145/1498765.1498785.

[237] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. "Case Study on Co-scheduling for HPC Applications". In: *2015 44th International Conference on Parallel Processing Workshops*. 2015, pp. 277–285. DOI: 10.1109/ICPPW.2015.38.

[238] Bin Liu et al. "Accelerating High Performance Computing Applications: Using CPUs, GPUs, Hybrid CPU/GPU, and FPGAs". In: *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*. 2012, pp. 337–342. DOI: 10.1109/PDCAT.2012.34.

[239] Nvidia. *Developing a Linux Kernel Module using GPUDirect RDMA*. URL: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[240] Tonmoy Dey et al. "Optimizing Asynchronous Multi-Level Checkpoint/Restart Configurations with Machine Learning". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2020, pp. 1036–1043. DOI: 10.1109/IPDPSW50202.2020.00174.

[241] Zhengzhang Chen et al. "NUMARCK: Machine Learning Algorithm for Resiliency and Checkpointing". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 733–744. DOI: 10.1109/SC.2014.65.

[242] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

Webliography

[684] *MPICH: High-Performance portable MPI*. URL: https://www.mpich.org/.

[685] *OpenMPI: Open-Source High-Performance computing*. URL: https://www.open-mpi.org/.

[686] *MVAPICH:MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE*. URL: http://mvapich.cse.ohio-state.edu/.

[687] *MPI: A Message-Passing Interface Standard*. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.