TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

Fault Tolerant Optimizations for High Performance Computing Systems

Dai Yang

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:
Prof. Dr. Martin Bichler

Prüfende der Dissertation:

1. Prof. Dr. Dr. h.c. (NAS RA) Arndt Bode

2. Prof. Dr. Dieter Kranzlmüller
   Ludwig-Maximilians-Universität München

Die Dissertation wurde am 16.09.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18.12.2019 angenommen.

Acknowledgments

This extensive work could only be accomplished thanks to the successful teamwork of present and past colleagues at TUM, KIT, RWTH, and Uni Mainz. I can only express my wholehearted thanks to all my colleagues, professors, students, and friends.

First, I would like to thank my advisor, Prof. Dr. Dr. h.c. Arndt Bode, for his inspiring advice and help throughout these years. In addition, I would like to thank Dr. Carsten Trinitis, who was always extremely helpful whenever I needed him. I would like to thank Prof. Dr. Dieter Kranzlmüller for his role as secondary advisor. Finally, I want to thank Prof. Dr. Martin Schulz, who has only recently joined the chair, for his support and help in improving my work, especially with the MPI-related topics.

I also want to thank all of my other colleagues at the Chair of Computer Architecture and Parallel Systems at TUM and LRZ, especially Josef Weidendorfer, Tilman Küstner and Amir Raoofy, who contributed many papers and helped me a lot. I will never forget the technical discussions with Josef and Amir, and all the encouraging words from Til which guided me through thick and thin. I would like to thank Beate Hinterwimmer and Silke Albrecht for their best support in the chair that anyone could imagine.

I also want to thank my other colleagues for their help: Andreas Wilhelm, Alexis Engelke, Marcel Meyer, Jürgen Obermeier, and 有間 英志. Furthermore, I want to thank several colleagues at the Chair of Astronautics at TUM: Sebastian Rückerl, Nicolas Appel, Martin Langer, and Florian Schummer for all the fun they sparked.

In addition, I would like to thank the Federal Ministry of Education and Research of the Federal Republic of Germany for providing the grant for the project ENVELOPE under the grant title 01|H16010D. I gratefully acknowledge the Gauss Centre for Supercomputing e.V. for funding this project by providing computing time on the GCS Supercomputer SuperMUC and Linux Cluster at the Leibniz Supercomputing Centre.

Finally, I want to especially thank my family and friends for providing me all kinds of support. They helped me to make crucial decisions. Just to mention some names: Nina Harders, Clemens Jonischkeit, Michael Schreier, Simon Roßkopf, Leonhard Wank, Lukas von Sturmberg, Marcel Stuht, and 安博. I also want to thank all the Bachelor and Master students, IDP Project students, and the guided research students for your contributions to my research.

Dai Yang
September 10, 2019

Abstract

On the road to exascale computing, a clear trend toward a greater number of nodes and increasing heterogeneity in the architecture of High Performance Computing (HPC) systems can be observed. Classic resilience approaches which are mainly reactive cause significant overhead on large scale parallel applications.

In this dissertation, we present a comprehensive survey on the state-of-the-practice failure prediction methods for HPC systems. We further introduce the concept of data migration as a promising way of achieving proactive fault tolerance in HPC systems. We present a lightweight application library – called LAIK – to assist application programmers in making their applications fault-tolerant. Moreover, we propose an extension – called MPI sessions and MPI process sets – to the state-of-the-art programming model for HPC applications – the Message Passing Interface (MPI) – in order to benefit from failure prediction.

Our evaluation shows that there is no significant additional overhead generated by using both LAIK and MPI sessions for the example applications *LULESH* and *MLEM*.

Zusammenfassung

Auf dem Weg zum Exascale Computing ist ein deutlicher Trend bezüglich hoher Parallelität und hoher Heterogenität in den Rechnerarchitekturen der Höchstleistungsrechnensysteme (HLRS) zu beobachten. Klassische Ansätze zur Behandlung von Fehlertoleranz, die hauptsächlich reaktiv sind, verursachen erheblichen Mehraufwand bei großen parallelen Anwendungen.

In dieser Dissertation wird ein umfassender Überblick über den Stand der Technik zur Fehlervorhersage für HLRS präsentiert. Darüber hinaus stellen wir das Konzept der Datenmigration als vielversprechenden Weg zur proaktiven Fehlertoleranz in HLRS vor. Wir führen eine leichtgewichtige Anwendungsbibliothek – LAIK – ein, die den Anwendungsprogrammierer dabei unterstützt, seine Anwendungen fehlertolerant zu machen. Außerdem schlagen wir eine Erweiterung – genannt MPI-Sessions und MPI-Prozesssets – für die Standardkommunikationsbibliothek für HPC-Anwendungen – das Message Passing Interface (MPI) – vor, um von der Fehlervorhersage zu profitieren.

Unsere Auswertungen zeigen keinen signifikanten Mehraufwand, welcher durch LAIK oder MPI sessions für die Beispielanwendungen LULESH und MLEM eingeführt wurde.

Contents

1. Introduction

HPC is an essential branch of computer science, which covers processing and computation of complex scientific and engineering problems using powerful systems. A wide range of computationally intensive tasks are the major application area of HPC systems, including but not limited to weather forecasting, climate simulation, oil and gas exploration, biomedical modeling, fluid failure prediction, astrophysics, and mechanical engineering. With its roots back in the 1960s, when Seymour Cray designed the first supercomputer – the *Cray–I* at Control Data Corporation [HTC+89], HPC has grown rapidly and become one of the most important research areas in computer science. Superpowers in the world – such as China and the United States – are racing head to head in research and construction of the most powerful HPC systems. Nowadays, HPC systems are massively parallel supercomputers consisting of millions of Commercial-off-the-Shelf (COTS) processors.

Currently, HPC systems are in the last decade of the *petascale*[1] era, and are about to reach their next milestone – the *exascale*[2] era. With the introduction of *Sierra*[3] and *Summit*[4], the US has regained the champion position in the race of supercomputers after a five-year lead by China. Moreover, these two machines are believed to be the last systems before the *exascale* era. With an increasing number of COTS components, such as processors, memory chips, nodes, and network components, HPC systems are becoming increasingly complex.

Similar to all complex systems, uncertainties – such as potential component faults – may propagate throughout the entire system, resulting in significant impact on the system's availability. The US Defense Advance Research Projects Agency (DARPA) has published a report [Ber+08] in 2008, stating that "Energy and Power", "Memory and Storage", "Concurrency and Locality", and "Resiliency" will be the major challenges in the exascale computing era. For resiliency, classic approaches to address faults, which are typically based on *Checkpoint & Restart*, require a significant amount of resources,

[1] *petascale*, a computer system that is capable of at least one petaFLOPS. One petaFLOPS is 10^{15} Floating Point Operations per Second (FLOPS).

[2] *exascale*, a computer system that is capable of at least one exaFLOPS. One exaFLOPS is 10^{18} Floating Point Operations per Second (FLOPS).

[3] `https://hpc.llnl.gov/hardware/platforms/sierra`, accessed in March 2019

[4] `https://www.olcf.ornl.gov/summit/`, accessed in March 2019

which is contradictory to the goals in efficiency [Ber+08]. Therefore, new approaches for achieving fault tolerance in HPC systems must be developed.

In this dissertation, we focus on the resiliency challenge in HPC systems. We analyze the state-of-the-practice of fault management in HPC systems and introduce two approaches to provide more efficient fault management in future HPC systems.

1.1. Technical Background of Computer Architecture

1.1.1. Types of Parallelism

To understand the development of the architectures in HPC systems, we first introduce the different types of hardware parallelism. While detailed descriptions of different technologies of parallelism can be found in textbooks on computer architecture [HP11], we introduce the most prominent type of parallelism used for HPC systems. For a better understanding, we present a taxonomy of parallelism in Figure 1.1. The definition of these different types of parallelism is mostly taken from Hennessy and Patterson [HP11]. A detailed description of these types of parallelism is presented below [HP11].

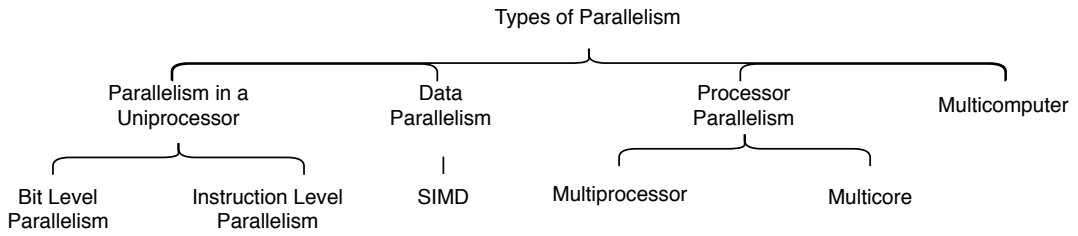

Figure 1.1.: Taxonomy of Parallelism

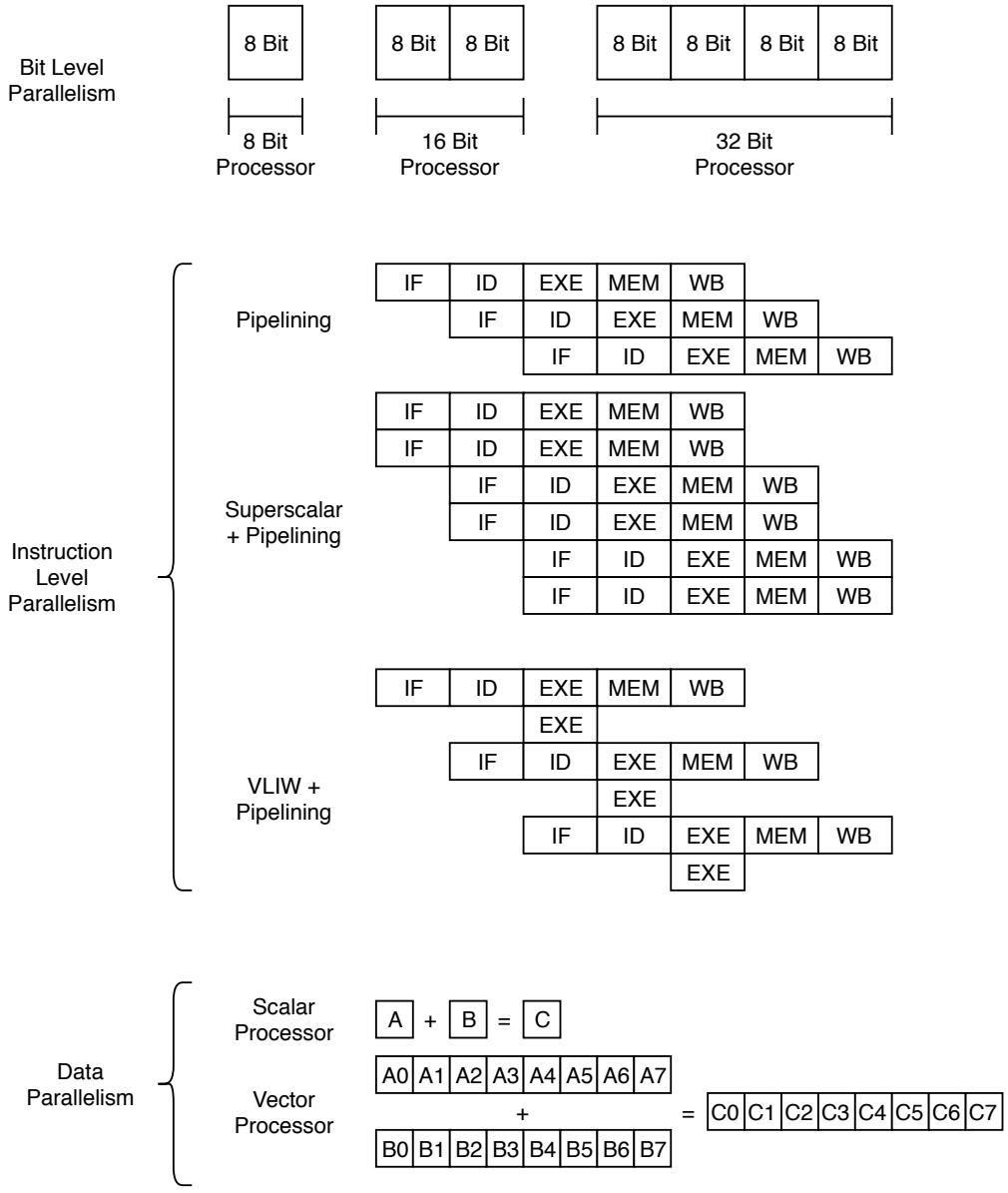

Figure 1.2.: Parallelism in a Processor

- Parallelism in a Uniprocessor: This group includes parallelism techniques that are used to improve the performance of a single processor (core). The most prominent examples are shown in Figure 1.2, and described below:
  - Bit level parallelism (cf. Figure 1.2 top): This technique aims at increasing

the computer performance by doubling the *word size* – the unit of data used by a particular processor. This way, the amount of information which can be processed by the processor is increased. The trend of increasing bit level parallelism has come to an end for Central Processing Units (CPUs) with the introduction of the 64-bit architecture. However, this type of parallelism is still used to increase the performance of other components, such as in the High Bandwidth Memory [GM03].

  - Instruction level parallelism (cf. Figure 1.2 middle): This is designed to increase the number of instructions that can be processed within one clock cycle. Any program can be treated as a stream of instructions. Furthermore, each instruction can be divided into multiple stages on modern processors. For example, in a classic processor, a single instruction is divided into five stages [HP11]: Instruction fetch (IF), instruction decode (ID), execution (EXE), memory access (MEM), and writeback (WB). There are three well-known types of instruction level parallelism. *Pipelining* is used to partially overlap multiple instruction execution. *Superscalar* techniques introduce multiple units for each of the instruction stages to provide parallelism. *Very Long Instruction Word (VLIW)* introduces multiple units for the execution (EXE) stage.

- Data parallelism (cf. Figure 1.2 bottom): This inherits the concept of vector processors introduced in the *CRAY-I* [Rus78] computer. Instead of processing *scalars*, which consist of a single value, data parallelism allows the processing of multiple data values – called a *vector*. Modern processors provide special units for vector processing to improve performance, e.g., Advanced Vector Extensions (AVX) [Fir+08] in Intel x86 processors and Advanced SIMD Extension (NEON) [Red08] in ARM processors. Graphical Processing Units (GPUs) greatly benefit from data parallelism to achieve high computational performance.

- Processor parallelism: This is the technology where multiple physical processors (or processor cores) are deployed on the same computer (node) to provide concurrent execution of applications. There are two different subtypes: Multicore and multiprocessor, which are illustrated in Figure 1.3 (top). While multiprocessors comprise multiple full processors including IO controller and memory subsystem, a multicore processor usually shares the same memory subsystem and I/O units.

- Multicomputer or cluster: This is the technique used to combine multiple independent computers with the same or different hardware configuration to concurrently solve the same problem. Figure 1.3 (bottom) shows an example configuration for a multicomputer.

Today, the above-mentioned parallelism technologies are widely used in supercomputers. Instead of the deployment of a specific technology, usually different technologies are combined to enhance the performance of supercomputers. However, in recent developments in the architecture of supercomputers, some techniques, such as *multi-core* and *data parallelism*, are rather common. This development is a result of many different factors. Besides the physical limits such as power and thermal design, the parallel performance of applications – which is usually analyzed by *Amdahl's* and *Gustafson's Law* – is also a contributing factor in the development of parallel computers.

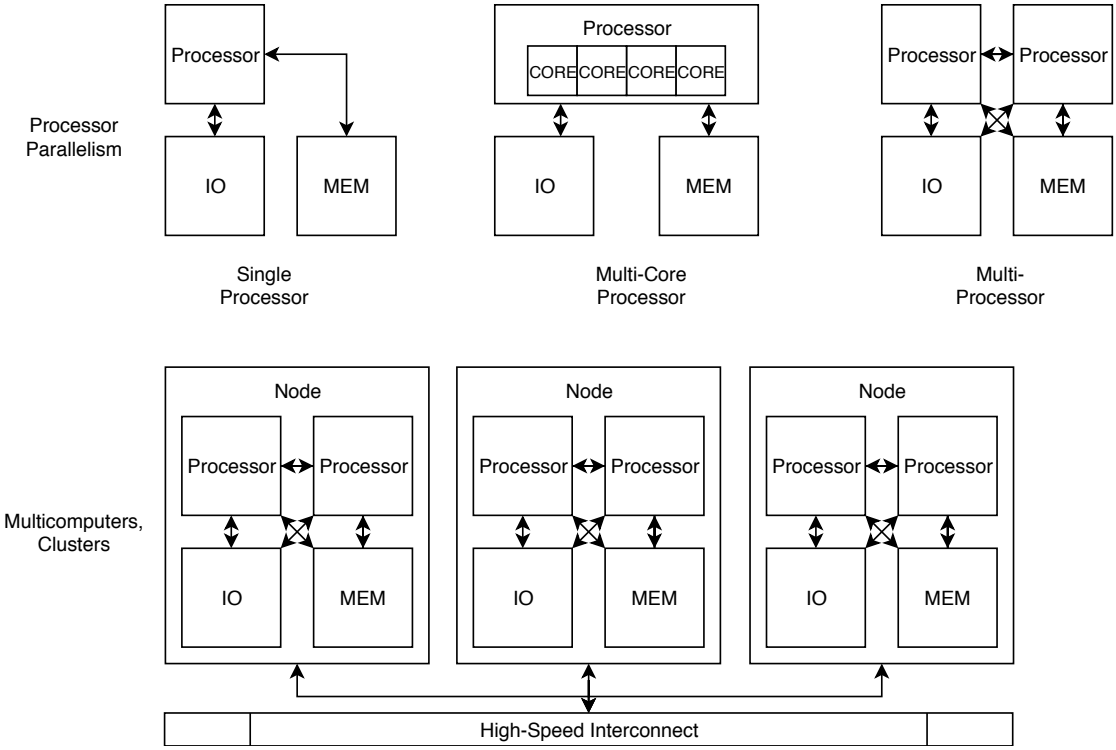

Figure 1.3.: Multicore, Multiprocessor and Multicomputer

1.1.2. Amdahl's Law

The *Amdahl's Law* [Amd67] is a well-known formula for calculating the theoretical maximum speedup of the execution of a given task for a fixed amount of workload. It is the standard way to predict the theoretical speedup for a parallel applications. It is defined in Equation 1.1, where S is the theoretical speedup of an application, s is the speedup of the part of the task that benefits from parallelization, and p stands for the proportion of the part of the task which benefits from parallelization in the original

application.

Amdahl's law states that the theoretical speedup of the execution of a parallel task increases with the number of resources in the system and that it is limited by the non-parallel portion of the task. A graphical representation of Amdahl's law with different percentages of the parallel portion and numbers of processors is given in Figure 1.4.

$$\begin{aligned} S(s) &= \frac{1}{(1-p)+\frac{p}{s}} \\ S(s) &\leq \frac{1}{1-p} \\ \lim_{s\to\infty} S(s) &= \frac{1}{1-p} \end{aligned} \tag{1.1}$$

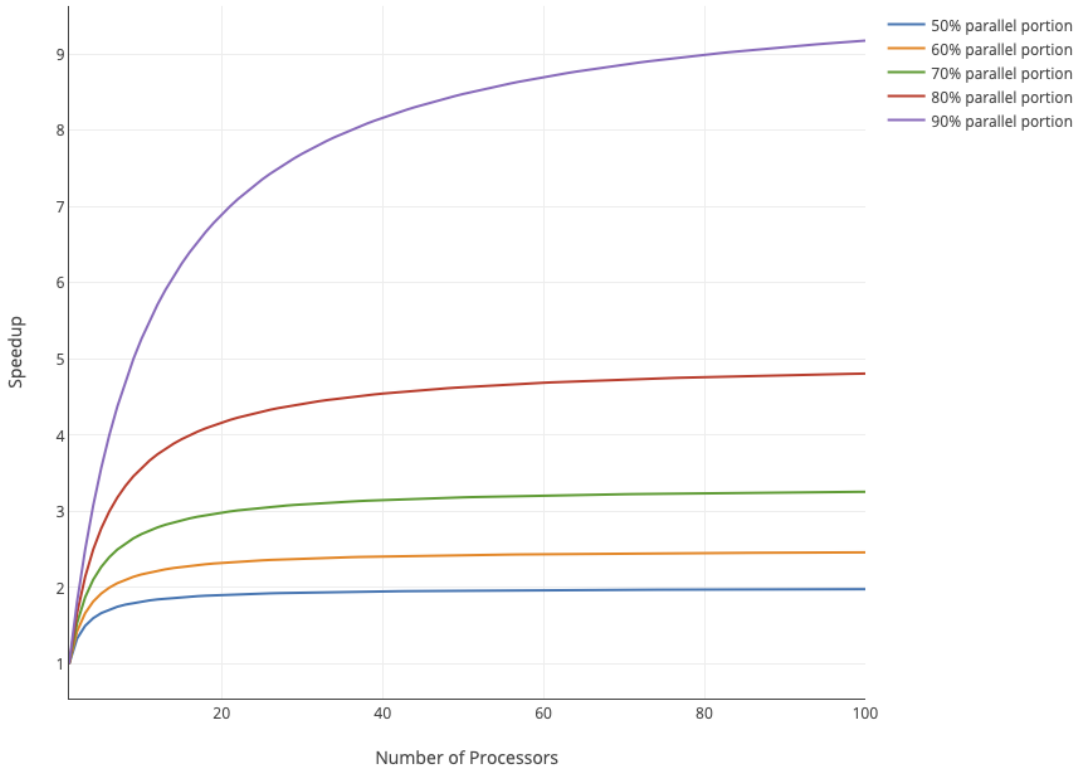

Figure 1.4.: Amdahl's Law

1.1.3. Gustafson's Law

In parallel computing, as an extension of *Amdahl's Law*, Gustafson's Law [Gus88] gives the theoretical speedup of the execution of a given task at a **fixed execution time** with respect to parallelization. It is shown in Equation 1.2, S_{theo} is the theoretical speedup of the application, s is the speedup of the parallel part of the application that benefits from parallelization, and p stands for the proportion of the part of the task which benefits from parallelization in the original application.

Unlike Amdahl's law, Gustafson's Law aims to estimate the maximum amount of work a parallel system can process, with a given parallel application in a fixed amount of time. It also shows that the maximum amount of work increases with the number of resources and is still limited by the non-parallel portion of the application. Figure 1.5 shows the estimates according to Gustafson's law for total speedup with a different number of processors and a different portion of application that is not sequential.

$$S_{theo}(s) = 1 - p + sp \tag{1.2}$$

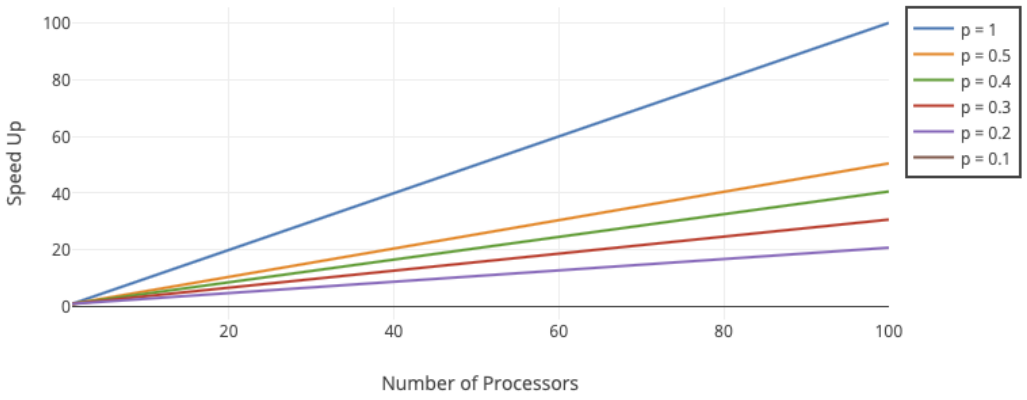

Figure 1.5.: Gustafson's Law

1.1.4. Heterogeneous Computing

Heterogeneous computing is a term that refers to a system which uses more than one type of processor [Sha06]. ARM big.LITTLE [CKC12] technology, which includes both high-performance and high-efficiency cores in a single chip computer, is an example

of this from the consumer market. In the context of HPC, a system which consists of different processing units (such as CPU and GPU) is considered to be heterogeneous.

The benefit of a heterogeneous system architecture is to provide the best performance and efficiency by dividing application tasks into different roles: While the CPU can take over general processing roles, a GPUs can take over all computationally intensive tasks. This way, a much higher system performance can usually be achieved.

Nevertheless, heterogeneous system architectures introduce many new challenges [KK11]. Different programs have to be developed to support the different types of processing units according to their instruction-set architecture and their application binary interface. Furthermore, libraries may differ on different types of systems. Finally, data management and communication organization greatly impact the performance gain on heterogeneous systems.

1.1.5. Other Factors in Processor Design

One limiting factor for the architectural design of modern processors is *heat flux*, which describes the flow of energy per unit of area per unit of time [Lie11]. While more energy is required to power a more complex processor (which consists of more transistors), the size of the chip does not necessarily grow due to the (currently) never-ending shrink of the semiconductor manufacturing process. The per unit heat flux therefore increases. Furthermore, as the yield of healthy processor chips significantly drops with increasing size of the chip [Mit16], the physical size of the processor cannot grow without any limit. Therefore, parallelism in a uniprocessor alone is no longer enough for modern HPC systems.

Furthermore, an increase in clock frequency (which usually leads to an increase in single-core performance) would also lead to significantly higher power dissipation, according to the following equations [Int04]:

$$P \sim C_{dyn} * V^2 * f \tag{1.3}$$

and

$$f \sim V \tag{1.4}$$

with Equation 1.4 in Equation 1.3 we have:

$$P \sim C_{dyn} * V^3 \tag{1.5}$$

where P = power, C_{dyn} = dynamic capacitance, V = voltage, = f frequency. This means that even a slight increase in voltage, which is required to make the processor faster, will cause a significant increase in heat dissipation. Consequently, increasing the processor speed by increasing the clock speed is also not an option for performance

improvement. Combined with the heat flux problem previously mentioned, an increase in processor frequency cannot be easily achieved.

1.2. Modern HPC System Architectures

1.2.1. *TOP500* and the High Performance LINPACK (HPL) Benchmark

For investigating the trends and developments in HPC architectures, the *TOP500* list[5] is **the** best source of representative data. *TOP500* is a project which was launched back in the early 90s to publish and present recent statistics on the development of HPC systems [TOP19]. The central idea is to use a benchmark, now well known as the *High Performance LINPACK (HPL)* [Don+79] to determine the performance of a given HPC system. This way, a comparable ranking of different machines can be created. Nevertheless, there is no common definition of what an High Performance Computing (HPC) system (supercomputer)[6] is. For everyday use and the sake of simplicity, people usually refer to HPC for all general-purpose computers that can achieve a high Floating Point Operations per Second (FLOPS) measure.

The HPL [Don+79], which was introduced by Jack Dongarra et al. in 1979, is an old but well-known benchmark. The basic idea of HPC is to benchmark a system's floating point number computing power, which is commonly known as *FLOPS*. The benchmark itself uses a kernel – that is the subroutine which performs the main computational work – a linear equation solver $A \cdot x = b$ for a dense symmetric n by n system matrix. This kernel is a common routine required by many engineering and physics applications. The HPL version used for benchmarking is typically parallelized with the Message Passing Interface (MPI) [WD96] and utilizes Basic Linear Algebra Subprograms (BLAS) [Law+77] as its math backend. However, as the deployment of *accelerators* has become common today, many other specifically optimized versions exist.

Benchmarking with HPL is effective, but is frequently criticized by many researchers. First, HPL only represents the application class of dense linear algebra, which is typically known to be compute intensive. However, many applications have different requirements regarding resources, such as memory bandwidth and interconnect bandwidth. Second, as the main source of funding for HPC systems is mostly governmental entities, the demand for a high rank in *TOP500* and a ranking list purely based on a single compute intensive benchmark may lead to a wrong focus when procuring an HPC system. Finally, the HPL version used for benchmarking is usually developed

[5] `https://www.top500.org`, accessed in March 2019

[6] In the scope of this thesis, the terms HPC and supercomputer are used as exact synonyms

and optimized carefully for a given specific system. The additional optimization effort is usually carried out by special system experts. This is typically not the case for applications from a real production environment. Targeting these problems, newer benchmarks, most notably the High Performance Conjugate Gradient (HPCG) benchmark [DHL15], are designed for a better ranking of HPC systems when compared with a real-world application environment.

Nevertheless, *TOP500* and the HPL are still seen as **the** metric for judging HPC systems. Hence, the results and discussions in this work about the development and trends of HPC systems are still based on *TOP500*, which was published in November, 2018[7].

1.2.2. Parallelism and Heterogeneity in Modern HPC Systems

To cope with the increasing demand for performance in HPC systems, one of the most common trends is the **large amount of cores** deployed, which comes with no big surprise. While the early systems such as the *Cray-I* [Rus78] relied on compact design with local parallelism using vector processors under the principle of SIMD (cf. Section 1.1.1), today's systems are based on the massively parallel architecture, which features a tremendous amount of compute units. Parallelism is everywhere in modern HPC architecture, from each processor core to the entire system (cf. Section 1.1.1, *multicomputer*).

Inside each core, instruction level parallelism and data parallelism approaches, such as AVX, have become the standard. Furthermore, bit level parallelism is used to increase memory bandwidth. At the processor level, more and more cores are being incorporated into a single chip. The increasing number of cores per processor results in an increasing complexity per chip. More transistors and die space are required to house the cores and their management units. At the system level, the number of nodes is increasing rapidly. The increase in number of nodes results in a high number of components, which leads to higher aggregated failure rates.

Furthermore, accelerators, such as GPUs, which benefit from data parallelism, are being deployed into many of state-of-the-art HPC systems, e.g., in *Sierra* and *Summit*. Heterogeneous system architectures have become one of the best ways to increase performance in HPC systems. Besides the system design with CPUs and GPUs, other accelerators (such as the manycore architecture - *Xeon Phi* [JR13] - and the Chinese *Matrix-2000*[8]) are becoming additional sources of compute power in HPC systems.
To visualize the correlation between the number of cores and maximum performance, we have plotted these respective values for the first 100 systems on the *TOP500* list in

[7] `https://www.top500.org/lists/2018/11/`, accessed in March 2019

[8] `https://en.wikichip.org/wiki/nudt/matrix-2000`, accessed in July 2019

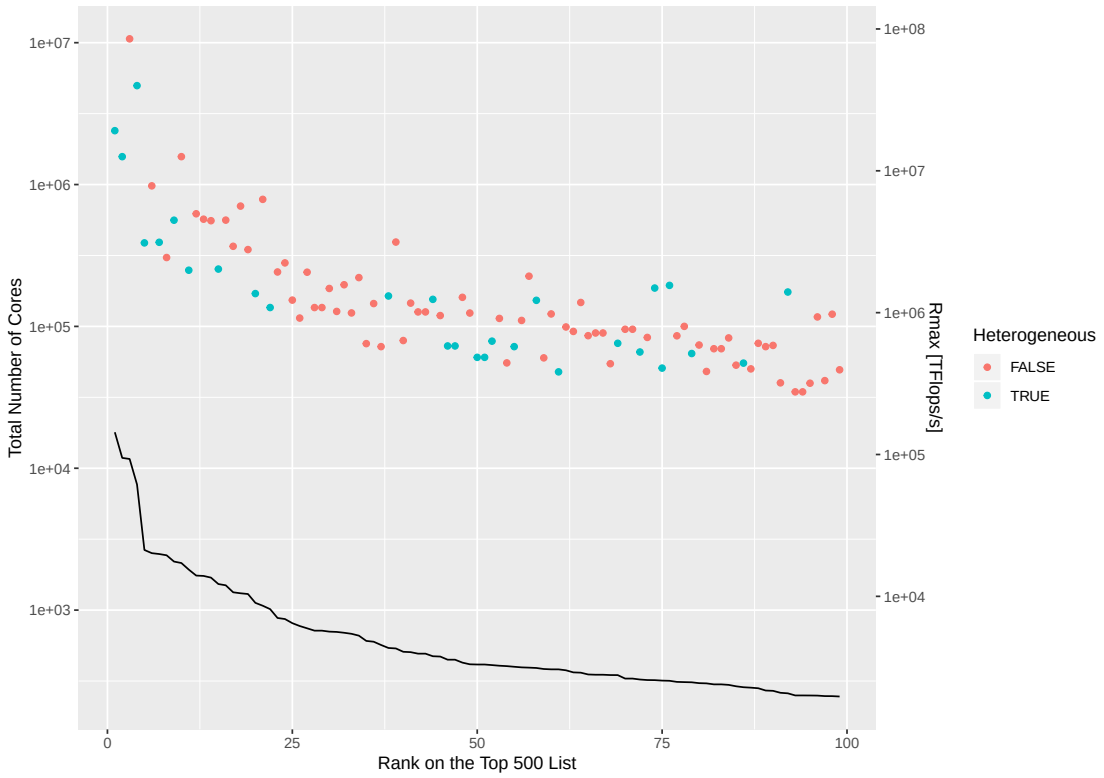

Figure 1.6.: Overview of Top 100 Systems on the November 2018 *TOP500* list

Figure 1.6. In Figure 1.6, the dots represent the number of cores of a given system, and the trend line below represents the maximum performance (of HPL) for these systems in the *TOP500* list published in November, 2018. Note that the y-axis of this figure has a *log* scale. The dots represent the number of cores of a given system, and the color of the dots represents its heterogeneity. The trend line represents the maximum LINPACK performance (R_{max}) in TFLOPS (Teraflops per Second). Moreover, as shown in Figure 1.6, many of the fastest systems on the *TOP500* are heterogeneous, most notably *Summit* and *Sierra*.

As shown in the figure, a clear correlation between these trends can be observed. In Figure 1.7, we further observe that no clear correlation between flops per core and their corresponding ranking can be observed. This observation supports our conclusion that performance gain in modern HPC systems is mainly due to increased multiprocessor parallelism, rather than to improved design of a uniprocessor.

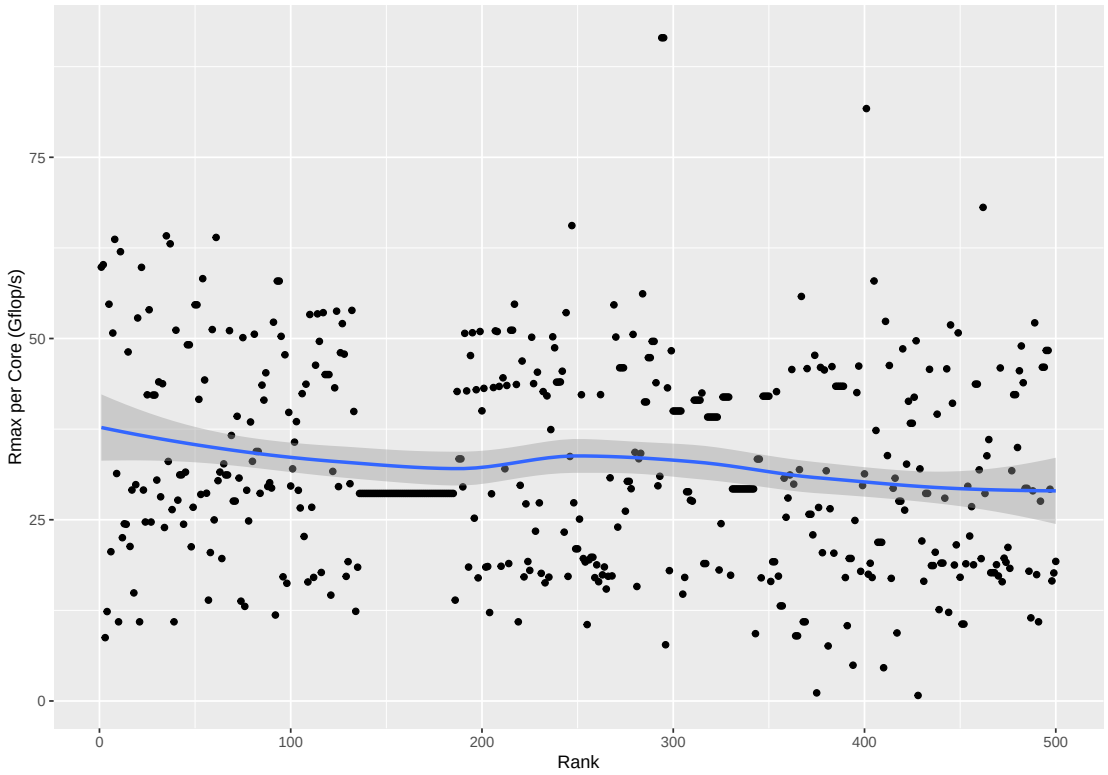

Figure 1.7.: Maximum Performance per Core for all Systems on the November 2018 *TOP500* list

1.3. Motivation

The trends in the architecture of modern HPC systems show that an increasing parallelism based on an increasing number of processor cores, independent nodes, and accelerators is to be expected in future HPC systems. Reliability, besides the management of parallelism and energy efficiency, has become a major goal in exascale computing [Ber+08]. Although there is no indication of an increasing failure rate in modern HPC systems yet, the challenge in resilience remains. Existing approaches suffer from the challenge of being mostly reactive, since failures can happen at any time with a higher probability due to the increased number of components in exascale computing. Therefore, a wide body of research targets predictive fault tolerance, which aims at forecasting upcoming failures and performing proactive measurements to prevent a failure. This way, mitigation mechanisms are applied when the system is still in nominal operation, which eliminates the need for an expensive recovery mechanism.

However, existing work in the field of failure prediction is diverse and lacks structure. Many different sources of failures, as well as data, and a large amount of different techniques for failure prediction exist. Some of the works are more general and target various scenarios, while others are tuned to solve a specific problem. To conclude, the current state-of-the-practice of failure prediction and the gaps and remaining challenges in failure prediction are still unclear. In the first part of this dissertation, we will look closely at existing work in failure prediction, and identify the gaps and challenges in the current state-of-the-practice in failure prediction.

Sufficient failure prediction is only a premise of efficient proactive failure prevention. An efficient approach for fault mitigation is the other integral part. Existing techniques such as *checkpoint&restart*(cf. Section 4.2.3), which cannot take advantage of any failure prediction, will require an extensive amount of resources and contradict to the high efficiency goals of HPC. Therefore, it is no longer sufficient to handle the evolving requirements in fault tolerance and reliability [Ber+08]. More recent techniques, which are transparent to the application, include process level migration [Wan+08] or virtual machines [Cla+05]. Nevertheless, these solutions still demand a significant amount of resources similar to the classic *checkpoint&restart*. Furthermore, these approaches impose a high overhead on the application, limiting an application's scalability and thus providing only limited applicability to emerging exascale requirements. In contrast to application transparent methods, an application can be modified to support fault tolerance and react to the prediction of an upcoming error. This way, resource management overhead by the framework that provides fault tolerance is reduced. The overall impact on performance and scalability is limited to a minimum. In the second part of this dissertation, we will present the concept of data migration, an application-integrated

approach for proactive fault tolerance. We further propose an application-integrated library – called *LAIK* – to help application programmers to utilize data migration to achieve proactive fault tolerance. Moreover, we present a potential extension to the *de facto* standard programming model in parallel programming Message Passing Interface (MPI) – The MPI sessions and MPI process sets, which are capable of utilizing the advantages in failure prediction and provide a sufficient interface for programmers to achieve proactive fault tolerance.

1.4. Contribution

In this dissertation, we will provide two new different methods for fault tolerance based on *data migration* according to the trends in the development of exascale HPC systems. With this work we hope to address a gap and point out promising new approaches to this emerging topic.

In particular, the main contributions in this dissertation are as follows:

- We provide a high quality in-depth survey using a large amount of recent literature on state-of-the-art failure prediction methods. Furthermore, research gaps, limitations, and applicability of these methods to modern HPC systems are discussed and analyzed.
- We present an overview of fault tolerance methodologies at different levels: system level, application level, and algorithm level.
- We present the concept of *Data Migration*, a proactive fault tolerance technique based on the repartitioning and redistribution of data combined with fault prediction.
- We introduce *LAIK*, an application-integrated fault tolerance library. The main goal of LAIK is to assist application programmers in achieving fault tolerance based on data migration by redistributing the data and removing any failing node from parallel execution. We evaluate the efficiency and overhead of LAIK using two existing applications: *MLEM* and *LULESH*.
- We introduce *MPI sessions*, a proposal to extend the MPI standard, and our extension to MPI sessions named *MPI process sets*. We present a prototype implementation for MPI sessions and MPI process set. Moreover, we show that the added functionalities can be used to achieve data migration based fault tolerance. We present evaluation results on the effectiveness of our prototype for data migration.

- We extend the concept of data migration and its role in creating more dynamic and malleable applications in the exascale era.

This dissertation is not the result of a standalone work flow. It is the combined result of a wide body of our publications in the last three years. A full list of our publications is given in Appendix F.

1.5. Structure of This Dissertation

The remainder of this dissertation is structured as follows:

- Chapter 2 introduces and explains important terminology used in this dissertation.
- Chapter 3 provides an in-depth overview of failure prediction methods in HPC systems and a discussion on their effectiveness.
- Chapter 4 outlines an overview of different fault management techniques.
- Chapter 5 presents the key concept for fault tolerance that this dissertation focuses on – *data migration*.
- Chapter 6 introduces and describes LAIK, a library for application-integrated fault tolerance based on index space abstraction and data migration. We also present the effectiveness and efficiency of LAIK by porting two existing applications – *MLEM* and *LULESH*.
- In Chapter 7, we propose a possible extension to the Message Passing Interface (MPI), the de facto standard in parallel programming, in order to support fault tolerance based on data migration. The effectiveness and efficiency of our extension is also evaluated with an adapted version of both *MLEM* and *LULESH*.
- Chapter 8 includes a discussion on fault management based on failure prediction and data migration in modern HPC systems.
- Chapter 9 provides a range of related work featuring similar topics and approaches for fault tolerance.
- Chapter 10 and 11 conclude this dissertation and provide an outlook on future work.

2. Terminology and Technical Background

The terminology used in this work is presented, clarified, and discussed in this chapter.

2.1. Terminology on Fault Tolerance

2.1.1. Fault, Error, Failure

Among different literature, there are many different definitions of the terms *fault, error, and failure*. The definitions introduced by Avizienis et al. [Avi+04] are used as references in this dissertation. Deviations from these definitions used in this dissertation are indicated explicitly in the remainder of this dissertation. Note that the terms of *fault*, *error*, and *failure* are used inconsistently across the literature.

- *Faults* are often referred to as a static defect in software or hardware [Avi+04], which can be the cause of an error. A fault is not directly observable, however, its manifestation – the *error* is visible. A fault in a system does not necessarily lead to an error in a system. Such systems which are resistant to faults, are called *fault-tolerant*. Examples of a fault in a computer system include a bit-flip in a Dynamic Random Access Memory (DRAM), or a transient failure in a transistor in the CPU.

- An *error* occurs when the current (internal) state of the system deviates from the correct state. Errors do not necessarily cause failures, however, they may even go entirely unnoticed (undetected error). The root cause (or each root cause in combination with the root cause of an error) is typically a fault (or multiple faults).
  An example of an error would be an erroneous result calculated by the CPU due to the impact of a transient fault on a transistor in the CPU.

- A *failure* is any incorrect system behavior with respect to system response or a deviation from the desired functionality [Avi+04]. The same failure can be manifested in different behaviors, e.g., a node being unavailable or delivering wrong results to users and the rest of the system due to a faulty network card.
  Sometimes the root cause of a failure may remain unspecified. For example,

Schroeder and Gibson [SG10] interpret every event as a failure that requires the attention of an administrator, and in many studies relying on log files simply all events tagged with a specific string. Klinkenberg et al. [Kli+17] use a so-called "lock event", that is an event preventing the user from using a node as the key description for *failure*.

To summarize, the relation between fault, error, and failure is shown in Figure 2.1. A fault may lead to an error in a system, which may propagate through the system and cause a failure. A failure is caused by one or multiple errors, which can eventually be tracked back to one or multiple faults as root causes.

Figure 2.1.: Relation between Fault, Error, and Failure

2.1.2. Fault tolerance

Fault tolerance is the capability of a given system, which allows the system to continuously operate even after failure of some components. Most fault-tolerant systems are designed to cope with specific types of failures in the system [Avi76]. For example, a computer is fault-tolerant against power failure, if it is equipped with an Uninterruptible Power Supply (UPS) system [GDU07].

The most trivial way of fault tolerance is *fault avoidance*, also known as *fault intolerance* [Avi76]. Here are some examples: By selecting enterprise-grade hardware which provides higher reliability, potential manufacturing faults in hardware components can be avoided, effectively making the system more reliable.

Another technique among the most common fault tolerance techniques is **redundancy**. Quoting Avizienis [Avi76]: "The redundancy techniques that have been developed to protect computer systems against operational faults may assume three different forms: hardware (additional components), software (special programs), and time (repetition)." The use of redundancy is widely spread in all kinds of computer systems: The UPS system already mentioned is a redundancy in the power supply system. The widely used Redundant Array of Independent/Inexpensive Disks (RAID) [Pat+89] technology is redundancy of a storage system. The well-known and used *Checkpoint & Restart* in HPC applications is a kind of software redundancy, which secures relevant data on persistent storage. Having some degree of redundancy might be the most efficient way of improving reliability. It is manifested as the motto of *avoiding single points of failure* in systems engineering.

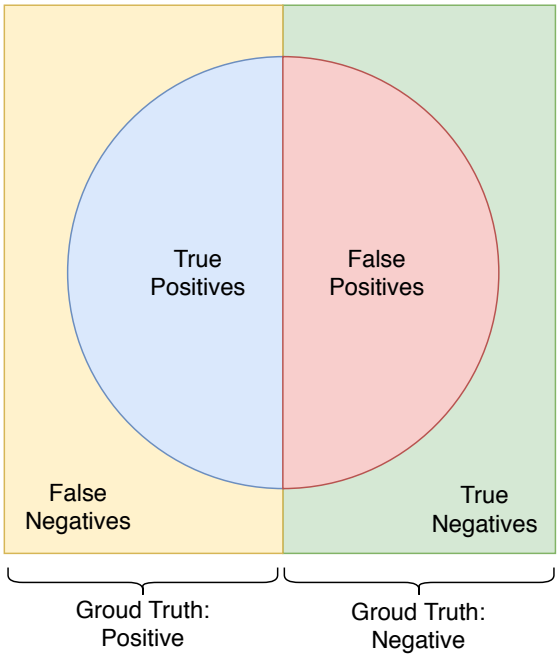

Figure 2.2.: Venn Diagram on the Relation between *True Positive*, *False Positive*, *True Negative*, and *False Negative*

2.2. Terminology on Machine Learning and Failure Prediction

The first part of this dissertation is an in-depth analysis of the state-of-the-practice in failure prediction. Most prediction methods are based on machine learning techniques. For better understanding, the basic concepts used for a machine learning device for a binary classification (prediction) problem, which is also known as a *perceptron*, are explained below.

In a binary classifier for failures, *True Positives (TP)* refer to correctly classified failures and *True Negatives (TN)* refer to correctly classified non-failure (in the following: ok) events. Similarly, *False Negatives (FN)* refer to misclassified failures and *False Positives (FP)* to ok events that are misclassified as failures.

To help understand the concepts of TN, TP, FN, and FP, Figure 2.2 visualizes their relation. In Figure 2.2, the *square* represents the total set of events in an experiment. The left half of the square represents the set of events whose ground truth is "positive". The right half of the square represents the set of events whose ground truth is "negative". The *circle* represents the prediction results "*positive*", and the *difference set* of the square and circle are the events predicted to be "*negative*". With this premise, true positives are

represented in blue, false positives in red, true negatives in green, and false negatives in yellow.

The relation between the events and their predictions is also illustrated in the *confusion matrix* in Table 2.1, where events are partitioned into failure events (E) and non-failure (ok) events (K). A binary classifier can therefore predict events to be positive (failure, $(+)$) or negative (non-failure, $(-)$). False positives are false alarms that would trigger a checkpoint even if no failure were imminent in this case.

Table 2.1.: Confusion Matrix for Failure Prediction

		Ground Truth	
		Failure (E)	Non-Failure (K)
Predicted	Failure (+)	True Positive (TP)	False Positive (FP)
	Non-Failure (-)	False Negative (FN)	True Negative (TN)

The literature that discusses failure prediction methods uses the following key metrics to evaluate their quality [Faw06]:

- *Precision:* the probability of a positive prediction to be correct, also expressed as the positive predictive value, calculated with $\frac{TP}{TP+FP}$.
- *Recall:* the probability of correctly classifying errors, also expressed as the true positive rate, calculated with $\frac{TP}{TP+FN}$.
- *Miss rate:* the probability of misclassifying errors as ok events, also expressed as the false negative rate, calculated with $\frac{FN}{TP+FN}$.
- *Specificity:* the probability of correctly classifying ok events, also expressed as true negative rate, calculated with $\frac{TN}{TN+FP}$.
- *Fall-Out:* the probability of misclassifying an ok event as an error, also expressed as false positive rate, calculated with $\frac{FP}{FP+TN}$.
- *Accuracy:* the proportion of correctly predicted events among the total number of events, calculated with $\frac{TP+TN}{TP+FP+FN+TN}$.

As Salfner et al. [SLM10] remark, such basic statistical metrics are less useful, especially if failure events are rare, which is the case for HPC systems. For example, a classifier prediction that was always true would have a perfect precision, yet zero recall. However, these metrics – especially precision and recall – are considered as quality criteria for failure prediction systems. In general, a higher precision value with a good

recall value is considered good for a classifier. However, there is no general threshold for these metrics to be considered "good". By contrast, these metrics only underline the effectivity of a classifier in a given setup.

Other researchers [Tae+10; Zhe+10; EZS17] have criticized these measures as being unsuitable for HPC as they do not account for lost compute time. They also suggest that a weighted metric might be a better solution for HPC. Despite the known deficiencies in classic metrics for failure prediction systems, these metrics are widely used by most researchers. Therefore, we still stick to these metrics when evaluating the failure prediction methods in this work.

2.3. Terminology on Parallel Computer Architecture

2.3.1. Flynn's Taxonomy of Computer Architectures

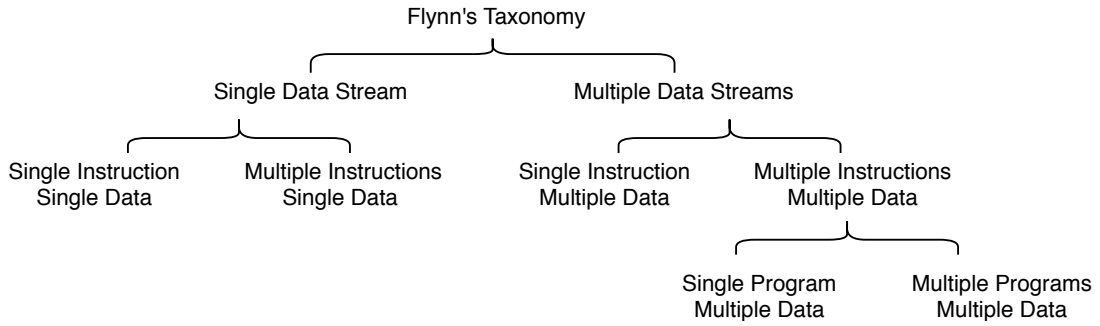

Figure 2.3.: Extended Flynn's Taxonomy [Fly72; Ata98]

Flynn's taxonomy is a classification of computer architectures, which was first introduced by Michael Flynn in the 1972 [Fly72]. This taxonomy has been widely used in the design of modern processor architectures, especially for those with a high degree of parallelism. Figure 2.3 illustrates Flynn's taxonomy. The four original categories defined by Flynn [Fly72] are based on the number of concurrent instructions and the number of parallel data streams available in the architecture. They are:

- Single Instruction Stream Single Data Stream (SISD).

  An SISD computer is a sequential computer which provides no parallelism at all. The Control Unit (CU) can only fetch a single instruction at once, and the Processing Unit (PU) can only process one single data stream. An example of an SISD computer is the early *x86* processor such as the *Intel 8086*. Figure 2.4 (top left) illustrates an SISD architecture.

- Single Instruction Stream Multiple Data Streams (SIMD).

  An SIMD computer can still fetch a single instruction at a time, but many PUs execute the instruction to multiple data streams in parallel at the same time. An example of SIMD is Intel's *AVX/2/512 instructions*, which can compute up to eight floating point operations at once. It is capable of executing the same instruction using multiple hardware threads concurrently. Figure 2.4 (bottom left) shows an SIMD architecture.

- Multiple Instruction Streams Single Data Stream (MISD).

  An MISD computer operates on one data stream, while many PUs execute different instructions on that data stream. This architecture is uncommon, with the *Space Shuttle flight control computer* [SG84] being the most prominent example of an MISD architecture. Figure 2.4 (top right) highlights an MISD architecture.

- Multiple Instruction Streams Multiple Data Streams (MIMD).

  An MIMD computer features many PUs which can execute different instructions on different data streams independently in parallel. It is the most widely used technique to exploit parallelism in modern computer architectures. While each multicore processor is an MIMD example, the *Intel Xeon Phi* manycore processor is a prominent example of an MIMD processor. All *TOP500* supercomputers are based on the MIMD architecture nowadays. Figure 2.4 (bottom right) represents an MIMD architecture.

  The MIMD architecture can be further divided into subcategories [Ata98]:

  – Single Program Multiple Data (SPMD)

    SPMD [Dar+88] is the most frequently used architecture in parallel programming. Multiple independent PUs execute the same program asynchronously on different data sets. When compared with SIMD, the programs in SPMD does not necessarily run at the same time point, while in SIMD a single instruction is applied to multiple data streams in *lockstep*. The most prominent programming model for SPMD architecture is the Message Passing Interface (MPI).

  – Multiple Program Multiple Data (MPMD)

    In MPMD, multiple PU executes different programs concurrently. A typical programming model for MPMD architecture is "*Master-Slave*", where the master runs a program which facilitates the distribution of workload and the slave calculates the workload using another program. The best-known example is the *Cell* microarchitecture [Gsc+06].

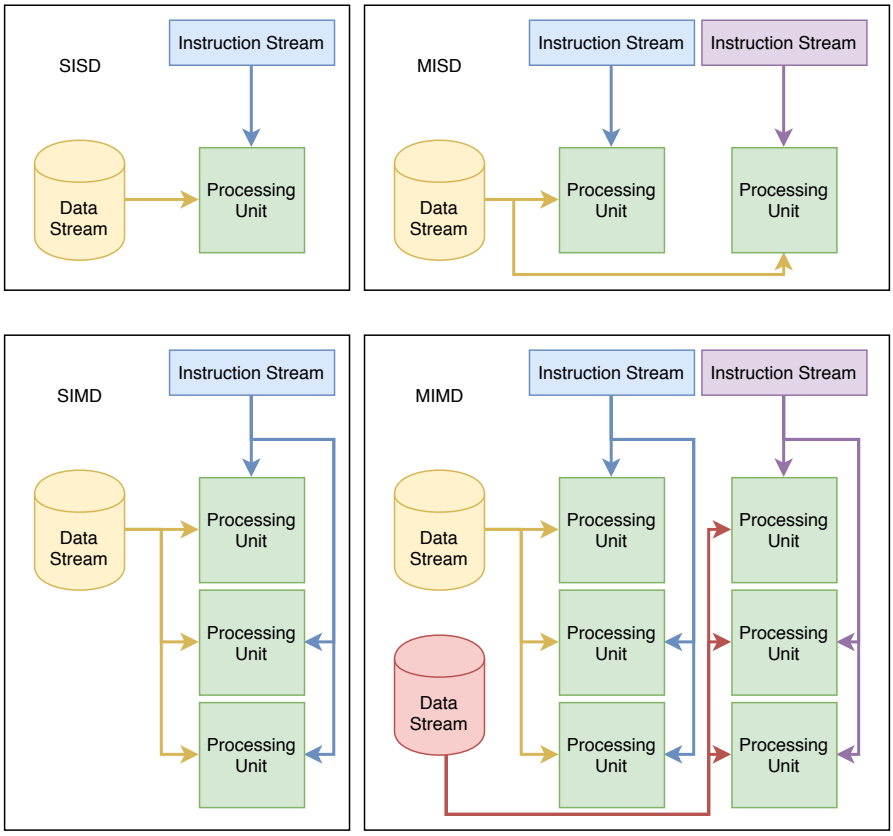

Figure 2.4.: Visualization of SISD, SIMD, MISD, and MIMD

2.3.2. Memory Architectures

In the context of parallel computing, two memory architectures are usually referred to:

- Shared Memory: Shared memory architecture refers to the design of a memory subsystem where memory can be accessed by multiple PUs at the same time. It is an efficient means of data transfer across different PUs, as no redundant data storage is required. An example of a shared memory design is the L3 Cache or the main memory of a state-of-the-art Intel multi-core processor, such as in the *Skylake* microarchitecture. Figure 2.5 (left) illustrates the shared memory architecture. There are three different types of access patterns for a shared memory system [EA05]:
  - Uniform Memory Access (UMA), where all PUs share the physical memory equally with the same latency.

  - Non-uniform Memory Access (NUMA), where memory access time is different depending on the locality of the physical memory relative to a processor.
  - Cache-Only Memory Architecture (COMA), where the local memories for the processor are used as cache only (instead of main memory). This is also known as scratchpad memory.

- Distributed Memory: A distributed memory architecture refers to a system with multiple PUs, in which each PU has its own private memory region, which cannot be accessed directly from another PU. Any application running on a specific PU can only work on its local data, and data which belongs to other PUs must be communicated through a (high speed) interconnect. Figure 2.5 (right) depicts the distributed memory architecture.

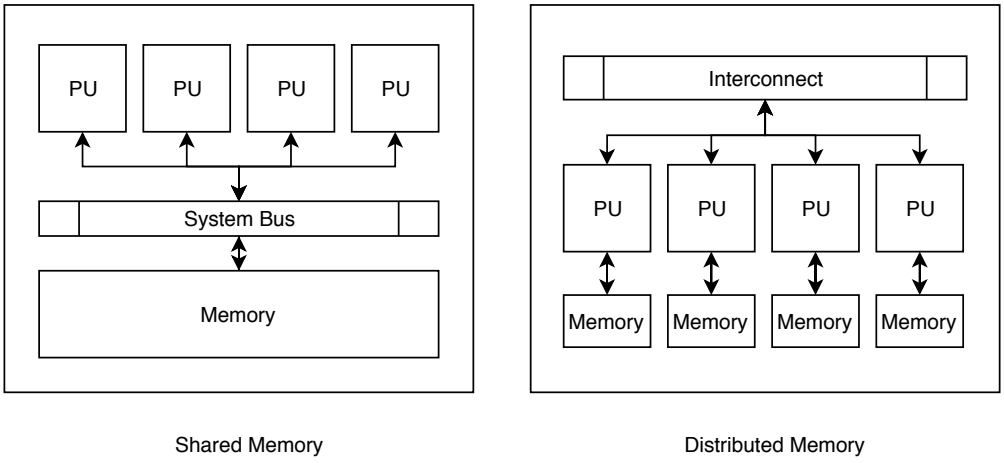

Figure 2.5.: Shared Memory vs. Distributed Memory

2.3.3. Scalability

In parallel programming, scalability is the property of a (parallel) application to handle an increasing amount of workload by adding more computational resources to the system [Bon00]. In the context of High Performance Computing (HPC), there are two common types of scalability:

- *Strong scaling* is defined as the change in program execution time when varying the number of processors for a *fixed* size of problem. The definition is based on *Amdahl's law* (cf. Section 1.1.2), which states that the upper limit of speedup for a fixed problem size is determined by the sequential part of the code.

- *Weak scaling* is defined as the change in program execution time when varying the number of processors for a *fixed size of problem per processor*. The definition is based on *Gustafson's law* (cf. Section 1.1.3), which states that in the ideal case, the (scaled) speedup in weak scaling increases linearly with respect to the number of processors without an upper limit.

2.4. Terminology in Parallel Programming

We use two major programming techniques in this work – namely MPI and OpenMP. An overview of these is given below.

2.4.1. Message Passing Interface

MPI is a portable standard for parallel programming, and is mostly used in HPC systems. It was introduced by Walker et al. in the 1990s [WD96]. Since then, the *MPI Forum* has been the governing body responsible for the MPI standard.

Technically, MPI is a communication protocol for a parallel application running on parallel computers. Today, MPI is the de facto standard for communication in parallel applications designed for systems with distributed memory (cf. Section 2.3.2). It also serves as the most dominant programming model for HPC systems [SKP06]. The most current version of the MPI standard is *MPI-3* [Mes18].

MPI covers a wide range of functionalities and concepts. Currently, the MPI standard defines the syntax and semantics for functions in the programming languages *C*, *C++*, and *FORTRAN*. The most basic concepts in MPI are process, group, and communicator. MPI follows the *SPMD* architecture in *Flynn's Taxonomy* (cf. Section 2.3.1). Each process is an instance of the application, which is usually an application process in the operating system. An MPI group (`MPI_Group`) is an abstract object which represents a collection of processes. An MPI communicator (`MPI_Comm`) is an object which provides a connection for a group of processes. Each communicator assigns every process within the communicator with a unique identifier called *Rank*. Users can create new communicators from an existing communicator, e.g., using `MPI_Comm_Split`. There are two trivial communicators in MPI: `MPI_COMM_WORLD`, in which all processes in an MPI instance are included; and `MPI_COMM_SELF`, in which only the calling process is included. Within a given communicator, both point-to-point based (such as send and receive) and collective communication (such as all-reduction) can be performed.

A detailed introduction to MPI is way beyond the scope of this dissertation. For detailed information, please refer to the MPI standard document [Mes18].

2.4.2. OpenMP

Open Multi-Processing (OpenMP) is a portable Application Programming Interface (API) designed for parallel programming on shared memory systems (cf. Section 2.3.1). Similar to MPI, it also provides language bindings in the programming languages *C*, *C++*, and *FORTRAN*. The standardization body of OpenMP is the *OpenMP Architecture Review Board*, which is a consortium of major computer hardware and software vendors. The first version of OpenMP was published in 1997 (for FORTRAN) and 1998 (for C) [DM98], respectively.

Technically, OpenMP is an implementation for multithreading following the fork-join model. At execution time, a master thread (usually the main process) forks a given number of threads. Each thread is uniquely identified by an id, where the master holds the id 0. Using the ids, the workload can be divided into partitions, where each thread works on a different partition.

Correctly deployed, an application can be written with using *hybrid model*, in which MPI is used for handling communication on the distributed memory part, and OpenMP is used to achieve high efficiency on a shared-memory node.

A detailed introduction to OpenMP is also beyond the scope of this dissertation. A variety of textbooks such as [Cha+01] teach parallel programming with OpenMP.

3. Failure Prediction: A State-of-the-practice Survey

3.1. Methodology and Scope

Failure prediction is a widely researched area in different fields because the emerging era of deep learning and advanced machine learning techniques has become a major focus of the recent research body. The idea of failure prediction is rather simple: By utilizing existing knowledge and heuristics of system behavior and information on the current system state, one can predict an upcoming event. This way, a dynamic failure avoidance method can be initialized, preventing the system from entering an erroneous state and hence preventing failure.

In HPC systems, the possible upsides of being able to predict future failures accurately are significant:

- It is proactive: It simplifies any mitigation mechanism by being able to react while the system is still working;
- It is efficient: It eliminates the need for frequent storage of the application state. Furthermore, combined with hot-swap hardware, it can simplify system management and increase utilization.

The work involved in carrying of this literature survey as part of this dissertation in this area is quite diverse: It covers a wide range of failure scenarios, systems and prediction techniques, to a point where it is hard to understand the coverage achieved with existing work and to identify gaps that are still present. As there are already many different failure prediction methods out there, we do not want to provide yet another failure prediction mechanism which demonstrates its capability on exactly one system. Instead, we are focusing on a cross analysis of the current state of the practice.

This chapter is related to and similar to the previous work in studies conducted by Salfner et al. [SLM10] and Xue et al. [Xue+07]. However, both of these studies mentioned are outdated and do not specifically focus on the current generation of large-scale HPC systems. For our analysis, a total of over 70 papers from 2010-2018 were screened, more than 30 of which were then selected for detailed interpretation.

The major outcome is a two dimensional matrix of prediction methods, which provides a categorization of current failure prediction methods. Unlike Salfner et al. [SLM10], this paper does not provide a taxonomy of methods. Previous effort [Jau17] shows that our 2-D classification provides a more comprehensive visualization of fault prediction methods, as many of the methods do not necessarily target a single specific problem. The main aims of the classification are:

1. The type of failure it covers, i.e., aims at predicting, and
2. the class of prediction method used.

It is important to mention that although the main search scope of our literature survey is HPC systems, we also include some research covering Cloud Computing (CC) systems. This is motivated by the fact that we are currently seeing a convergence trend between HPC and CC [GR17]. Moreover, it is generally expected that the convergence of these two systems will continue to accelerate in the near future.

A survey on failure modes that current work has been targeting is presented in Section 3.2 in the remainder of this chapter. These results are then used as drivers for forming the classification of the survey work, along with the prediction methods used. The result is a two dimensional matrix, which we will derive in Section 3.3 and present in Table 3.4.

The basic literature research part of this chapter was previously covered and introduced by Jauk [Jau17]. Partial results from the evaluation used in this chapter were also published in our previous paper [JYS19].

3.2. Survey on Failure Modes in HPC Systems

Raw failures - that is a set of data which clearly marks failures - in any supercomputer are hard to acquire. Hence, secondary literature currently is the main source of insight into this matter. Notable work in this field was done by Bianca Schroeder and various coauthors [SG10; ES13; EZS17], analyzing raw failure data from a cluster of 11k machines deployed at Google[1] (in the following called the "Google dataset") and a Hadoop Cluster at Carnegie Mellon University[2]. Furthermore, some other datasets regarding failures in (HPC) Data Centers are provided in the *Computer Failure Data Repository (CFDR)*[3]. Currently, a total number of 13 datasets are mentioned in the CFDR.

[1] `https://research.googleblog.com/2011/11/more-google-cluster-data.html`, accessed in June 2019

[2] `ftp://ftp.pdl.cmu.edu/pub/datasets/hla/`, accessed in June 2019

[3] `https://www.usenix.org/cfdr`, accessed in June 2019

3.2.1. Failure Modes

As expected, most existing research focuses on memory errors in HPC systems, due to their high occurrence. There are slightly fewer studies on disk failure. Nevertheless, disk errors have been the research focus of much independent research in failure prediction. Against trends in the HPC community, though, two areas have received surprisingly little attention: GPU failures and failures caused by the Operating System (OS).

Node failures have been the research focus of many papers. However, most of the papers, which deal with node failures, treat all types of failures equally and do not distinguish between the actual errors occurred, nor do they pinpoint the root causes of the failures.

As a prominent example, Di Martino et al. [Di +15] analyzed more than five million application runs in the span of 517 days on the 22,640-node *Blue Waters* Cray system. Their findings regarding application success and failure are listed in Table 3.1, which also includes another earlier study [LHY10] that examined 2,084 jobs from the *Franklin* Cray system. As shown in the Table 3.1, both studies provide similar findings on failure distribution.

Table 3.1.: Causes of Application Failure on Two Cray systems [Di +15; LHY10]

Termination	**Blue Waters**	**Franklin**
successful	66.47%	61.8%
user	18.75%	25.0%
system	1.53%	1.5%
user / system	8.53%	11.7%
walltime	4.71%	-

In their 2010 study [SG10], Schroeder and Gibson investigated failure data from 22 HPC systems at Los Alamos National Laboratory (LANL) over 10 years, gathered from 1996 until 2005, as well as a dataset with failure data from an unnamed supercomputing site over a period of one year. The Los Alamos National Laboratory (LANL) cluster consisted of 4,750 nodes with 24,101 processors. The anonymous site had 20 nodes with 10,240 processors. Failure reasons for the two facilities along with the respective percentages of their occurrence are presented in Table 3.2. The authors concluded that some systems showed a decrease in failures with unknown causes over time. However, the relative percentages of the other failure categories remained constant.

Table 3.2.: Root Causes of Failures in Two High Performance Systems [SG10]

LANL	(%)	**Unnamed Site**	(%)
hardware	64	hardware	53
software	18	software	22
network	2	-	-
environment	2	-	-
human	< 1	human	2
unknown	14	unknown	23

In addition, their data show that "failure rates do not grow significantly faster than linearly with system size." [SG10] Furthermore, failure rates on a per-node basis show significant variance, depending on the node's workload. The Cumulative Distribution Function (CDF) of the failure rate per node is best shown as a normal or log-normal distribution.

By analyzing failure rates over time, the systems follow one of two patterns [SG10]:

1. An initial high failure rate with a significant drop after approximately three months. This is consistent with the commonly known concept of "burn in" or "infant mortality": Initial failures, which remain undetected during the testing period, occur and are subsequently eliminated.

2. Increasing failure rates during the first approximately 20 months, followed by continuously decreasing failure rates afterward (time toward full operational system state). The continuous process of adapting the system to full production is most likely to have caused this observation.

Earlier results from two IBM Blue Gene systems by Hacker et al. [HRC09] are confirmed by Schroeder and Gibson, who also point out that node failures in general are still best modeled by a Weibull or gamma distribution[4].

In an earlier study, H.-C. Lin et al. [SG10] did not find any spatial correlation for errors of the six categories - *hardware, software, network, environment, human, unknown*. This result is contrary to a later study, which found that "when the number of correlated failures [...] is relatively high, the spatial correlation dominates" [Ghi+16].

3.2.2. On Root Causes Analysis

By analyzing failure modes, approaches for predicting failure occurrence can be derived. Some examples on this are provided by El-Sayed et al. [EZS17] and Pinheiro

[4] also known as *bathtub curve*

Table 3.3.: Most Common Root Causes of Failures [SG10]

Hardware	(%)	**Software**	(%)
Memory Dimm	30.1	Other Software	30.0
Node Board	16.4	OS	26.0
Other	11.8	Parallel File System	11.8
Power Supply	9.7	Kernel software	6.0
Interconnect Interface	6.6	Scheduler Software	4.9
Interconnect Soft Error	3.1	Cluster File System	3.6
CPU	2.4	Resource Management System	3.2
Fan Assembly	1.8	Network	2.7
Router Board	1.5	User Code	2.4
Fibre Raid Controller	1.4	NFS	1.6

et al. [PWB07]. Schroeder and Gibson [SG10] list out some of the most common root causes of hardware and software failures. Table 3.3 lists these root causes. They indicate that hardware problems are the most probable cause of failures in HPC systems. They further point out that "memory was the single most common 'low-level' root cause for all systems (across all failures not only hardware failures)..." However, their study does not treat errors which are from the user software itself. Most errors that lead to application abortions are the applications themselves [Fra+19].

In the following we present results from different studies covering the most probable root causes.

Memory

Many studies, such as by Schroeder et al. [SPW09] and Sridharan et al. [Sri+13], deal with memory errors because they are the most frequent hardware error. In [SPW09], Schroeder et al. analyzed two years of DRAM *error* data from Google's computing infrastructure (i.e., not HPC data). In [Sri+13], Sridharan et al. investigated DRAM *faults* from the *Jaguar* and *Cielo* supercomputers. The main findings from both studies include the following:

- DRAM age correlates with failure rates. According to Schroeder et al. [SPW09], an increasing rate for correctable errors in the first twenty months and uncorrectable errors in the first three to five months can be observed. This failure rate is then plateaus out afterward. In contrast, according to Sridharan et al. [Sri+13], total failure rates decrease for approximately the first two years of operation.

- [SPW09] and [Sri+13] agree that there is no clear correlation between error rates and memory technology (e.g., DDR1, DDR2, and DDR3), with [SPW09] examining DDR1, DDR2, and FBDIMM and [Sri+13] examining DDR2 and DDR3.
- A conclusive correlation between error rates and DRAM vendors is not observable.
- Although the effect is small, Schroeder et al. [SPW09] point out a correlation between error rates and temperature which again correlates with CPU utilization.
- While no spatial correlation of memory errors themselves is found in these two studies, this correlation is existent and documented in many different studies [Bau+16; Pat+17; HSS12].

Disks

Notable studies on disk failures include work by Schroeder et al. [SG07a], Pinheiro et al. [PWB07], and Ma et al. [Ma+15]. A large number of drives is used as a dataset by Schroeder&Gibson and Pinheiro et al.., while Ma et al. introduce some prediction methods, which analyze more than 100,000 hard drives each. Schroeder and Gibson use data from seven different sites - four HPC sites and three cloud providers - and gather data on Serial AT Attachment (SATA), Small Computer System Interface (SCSI), and Fibre Channel (FC) drives from four different vendors. Pinheiro et al. use data from *Google*'s computing infrastructure, which includes SATA and elder parallel ATA drives from nine different vendors. A more recent paper covering Solid State Drives (SSDs) within the HPC environment is not known by the time of writing.

Both Pinheiro et al. and Ma et al. point out that stress tests provided by manufacturers do not reflect the realistic picture, since the definition of *drive failure* varies between different manufacturers [SG07a; PWB07]. Furthermore, customers tend to express a "better safe than sorry" mentality, demanding "overly reliable" components. Hard drives tend to be returned even when the manufacturer finds them to be in a perfectly fine working condition. In addition, the pressure on the manufacturers regarding warranty is high. These phenomena further lead to a high margin in endurance specification. Both studies take the same approach by defining failure as the necessity of a drive to be replaced and calculate Annualized Replacement Rate (ARR) instead of Annualized Failure Rates (AFR). The ARR is calculated as the number of disks returned to the manufacturer, while the AFR based on the number of disks that is likely to fail. Also the AFR is defined as a fix quotient from the Mean Time Between Failure (MTBF):

$$AFR = 1 - \exp(\frac{-8766}{MTBF}) \tag{3.1}$$

Both studies come to a conclusion that ARRs are significantly higher than the respective AFRs. In contrast to the AFR of 0.58% to 0.88% commonly found in manufacturers' data sheets [SG07a]: [SG07a] reports ARRs of 0.5% up to 13.5% and an average ARR of 3.01% while [PWB07] reports ARRs of 1.7% to 8.6%.

All studies mentioned further investigate different aspects of drive failures, where no significant difference in replacement rate is observed with different (rotating Hard Disk Drive (HDD)) technologies. Error rates in most hard drives show an increasing trend over time with a peak at year 3-4. The replacement rates of SATA and SCSI / FC disks reject the commonly assumed "bathtub" model for failure rates. Infant mortality is relatively low and there is no flat bottom: failure rates continuously increase for the first few years of the drive's life. Another work [Ma+15] finds the same trend for most of the drive population, with failure rates reaching a peak in their third or fourth year of operation. Less than 50% of the population, however, show no increasing trend, but exhibit an almost constant failure rate over the observation period. Furthermore, a significant autocorrelation for disk failure rates within a period of 30 weeks is observed, most probably because of parameters affecting the entire population [SG07a].

Pinheiro et al. [PWB07] propose to model the failure rates as a function of utilization and temperature. Surprisingly, the authors find that drive failures are correlated with high utilization only in brand new and very old drives. Moreover, failure rates increase both with increasing and decreasing temperature, although a higher temperature appears to be more harmful.

Finally, several investigations [PWB07; Ma+15] suggest that Self-monitoring, Analysis and Reporting Technology (SMART) parameters of hard drives reveal that the number of reallocated sectors is highly correlated with disk failure.

GPUs

To our knowledge, the study by Tiwari et al. [Tiw+15b] and a "sister study" by Tiwari and other co-authors [Tiw+15a] on the *Titan* supercomputer are the only comprehensive studies of hardware GPU errors in HPC systems. This area has received surprisingly little attention, although there is a clear trend toward deploying GPU-based systems. In addition, Martino et al. [Di +15] show that GPUs produce more faults.

Tiwari et al. researched in their study on GPU failures (which are defined as GPU errors that led to an application crash) on 18,688 GPUs installed at the Titan supercomputer at Oak Ridge National Laboratory (ORNL) over 18 months. First, the authors find that GPU errors occur approximately once every two days, which is rarer than the manufacturer's MTBF would suggest. A Weibull distribution with high "infant mortality" also models their failure data best. Further, proper stress testing may reveal faulty GPUs prior to production operation Finally, newer GPUs (e.g., Kepler vs. Fermi)

show a better fault behavior.

The study also yields several policy recommendations [Tiw+15b]:

- Rigorous stress testing before and during the production run (e.g., for cards which experience a high number of errors) improves the MTBF of the system significantly. (Notably, six GPUs experienced 25% of all double bit errors, and ten cards experienced 98% of all single bit errors). Soldering of cards also helped reduce the high number of errors whenever connection to the host was lost.
- NVIDIA's Kepler architecture shows several improvements over the previous Fermi architecture. The increased resiliency is reflected in with a comparable rate of double bit errors despite the smaller transistor size, a lower vulnerability to radiation-induced bit corruption, and a better dispersal of workloads.

Although nothing has been published yet, the failure rate and distribution among GPUs have been under study by some researchers in the context of bitcoin mining.

Software and Application

Software and application failures in HPC were examined by Yuan et al. [Yua+12] and recently by El-Sayed et al. [EZS17], with the first study gathering data from eight sites and the latter covering three sites. [EZS17] explores failure prediction as well. Both studies agree that users are the main reason for job failures. El-Sayed concludes that many users are "optimistic beyond hope, wasting significant cluster resources" when it comes to re-executing failed tasks, consuming significantly more CPU time than successful jobs. A limitation on job retries may help to solve this problem. Moreover, they propose a *speculative execution* for tasks in the form of redundant scheduling. This is normally because a user only terminates hanging jobs only after they consume more time than usual. The configuration of jobs submitted within an Hadoop framework is also analyzed by El-Sayed et al. [EZS17]. Their result shows that most of the failed jobs deviate from the default configuration. Schroeder et al. also suggest the implementation of limits on the retries, as well as additional job monitoring.

In a most recent study, Alvaro et al. [Fra+19] point out a potential way to filter out job failures caused by application failure by using the combination of the Simple Linux Utility for Resource Management Workload Manager (SLURM) log and cluster monitoring data.

3.3. Survey of Failure Prediction Methods

A survey by Salfner et al. from 2010 [SLM10], which also provides a taxonomy of failure prediction, is also related to this work. However, it has two shortcomings:

1. Certain prediction methods cannot be classified to a specific branch of the taxonomy;
2. The type of failure, as well as the affected components, are not considered. Consequently, no recommendation can be made for a certain system.

The results presented in this chapter are also included in our recent paper [JYS19].

To eliminate these shortcomings, Table 3.4 shows a classification of failure prediction methods in HPC by using a matrix representation of failure types vs. prediction methods. The horizontal dimension of the proposed classification categorizes failures, thereby showing *what* is failing. We took the failure modes from Chapter 3.2 with these changes [JYS19]:

- SW/S stands for Software or System Failure.
- Node stands for hardware failure. In both cases, a root cause was unspecified by the corresponding predictor (non-pinpointable).
- On the pinpointable side, we have subdivided surveyed work into Disk, Memory and Network failure prediction. Since we only found very limited work on failure prediction for GPUs/accelerators, we have omitted this column for space reasons (see discussion in Section 3.2.2).
- The category "Log" indicates methods that predict upcoming events in log files, regardless of whether this entry actually correlates with an actual failure. This is especially the case for many of the prediction methods based on *IBM Blue Gene* log files. There are several studies which do not predict failures, but the occurrence of a log message with severity FATAL or FAILURE.
- Other studies may predict any fault in a given system or several faults at once, these are illustrated by the underlying blue fields within in multiple columns.

On the vertical axis, we group prediction methods in different categories each representing one major concept of prediction. The structure of their vertical direction is denoted as the follows:

- "Correlation, Probability" represents methods that are based on correlation of probabilities.
- "Rule" represents methods that are based on rules.
- "Math./Analytical" shows purely mathematical modeling methods that do not use a statistical approach.

Table 3.4.: A Classification of Literature on Failure Prediction in High Performance Computing [JYS19]

Class		SW/S	Node	Disk	Mem.	Net.	Log
Root cause		unspecified		pinpointable			
Correlation, Probability	[Lia+06]						- / 82
	[Bra+09]	- / -					
	[GCK12]	93 / 43					
	[Cos+14]				- / -		
	[Fu+12]$^{A,B,C}_{D,*}$						78 / 75 69 / 58 88 / 46
	[Gai+12] ^{D,*}	91.2 / 45.8					
	[Fu+14]$^{A,D}_{*}$	88 / 75 77 / 69 81 / 85					
Rule	[RBP12]		- / -				
	[Wat+12]^{D,*}						80 / 90
	[WOM14]^{D,*}						58 / 74
	[Ma+15]^{D}			- / -			
Math., Analytical	[Zhe+10]^{C}						40 / 80
	[Tho+10]						- / -
Decision Trees / Forest	[Rin+17]			- / -			
	[Gan+16]			* / *			
	[NAC11]		74 / 81				
	[Kli+17]^{C}		98 / 91				
	[SKT15]	94.2/85.9					
	[EZS17]^{B,D}	79.5/50 95 / 94					
	[CAS12]	53 / -					
	[GZF11]	- / -					
	[SB16]^{C}		72 / 87				

(Table continued)

Class		SW/S	Node	Disk	Mem.	Net.	Log
Root cause		unspecified		pinpoint-able			
Regression	[Liu+11b]	- / -					
Classification	[Lia+07][C,D]						55* / 80*
	[Xu+10]		66.4/59.3				
	[Zhu+13][A,C]			- / -			
	[Pel+14]	* / *[C]	* / *[C]				
	[Tho+10][D]						- /-
Bayesian Network, Markov	[AWR15]		93/91				
	[Yu+11][B,D]						82.3/85.4 64.8/65.2
Neural Network	[Zhu+13]			- / -			
	[CLP14]	-/-					
	[IM17][A]	89/- 80/-					
Meta-Learning	[Gu+08][C]						90*/70*
	[Lan+10]						-/-

[A]: Results for different data sets.
[B]: Results for different training parameters.
[C]: Results vary greatly with different parameters.
[D]: Paper lists several methods or setups.
*: Many results provided, see reference.
–: No numerical result is given.

- "Meta" stands for approaches based on *Meta Learning*.
- As a large amount of prediction methods are based on decision tree and random forest, these methods are selected and categorized specifically.

The blue cells indicate whether a specific method is used to predict failures within a category. Multiple blue cells within one row indicate that this method can predict failures related to multiple error/root causes. Respective values for precision and recalls (precision/recall) are shown in the table. "-" denotes that no value is given in the reference. "*" means that many values are given for different setups. Many of these references utilize more than one method or dataset, which is why a given respective value is futile in many cases. In addition, different training parameters also affect the precision/recall values. These references are marked in table 3.4 with “B” and “C” . Furthermore, values are only listed if the respective paper explicitly list precision and recall values according to the definition in Section 3.1 to ensure comparability [JYS19].

A detailed discussion on references based on the classification of prediction methods is presented in the remainder of this chapter. However, this classification is not perfect, as many of these classes are dependent on each other in the sense of mathematics.

3.3.1. Probability and Correlation

Probability theory is a mathematical branch which deals with probability. Probability is the quality or state of being probable, that is, the likelihood that a specific event will occur [SW89]. The most common understanding of probability is the quantification using a number between 0 and 1, for which the larger the number, the more probable an event will occur. Probability theory provides concepts and axioms in terms of mathematics. Probability theory also provides the foundation of modern machine learning approaches [JYS19].

Correlation is any statistical relationship between different random variables. The random variable itself is a function that maps a specific outcome of an event to a probability. In practical use, correlation commonly refers to the linear relation between variables. A typical way to explore correlation between the probability distribution of these random variables [JYS19].

The calculation of probability and correlation scores is an easy way to gain insight into potential failures. It is an integral part of many failure analysis methods, such as Fault Tree Analysis (FTA) and Markov Analysis [JYS19].

Many of the papers mentioned above estimate the probabilities of failures. Pinheiro et al. [PWB07] find that the probability of disk failure with 60 days after observing a certain SMART value increases 39 times after one scan error, 14 times after one reallocation count, and 21 times after one offline reallocation count. Schroeder et al. [SPW09]

point out that a correctable memory error increases the probability of an uncorrectable error occurring in the same month by 27-400 times and by 9-47 times in the following month.

Liang et al. [Lia+06] propose simple prediction methods based on spatial and temporal skewness in the distribution of failures in a BlueGene/L system. For example, 50% of network failures occur within 30 minutes of the previous failure; 6% of midplanes encounter 61% of network failures. The strategies trigger increased monitoring (for a certain time / for a specific midplane) after a failure has been encountered. Another proposed strategy based on the correlation between fatal and non-fatal events triggers monitoring when two non-fatal events occur in a job.

Gainaru et al. [GCK12] introduce a signal-based approach for failure prediction. Considering events as signals, they extract periodic, silent, and noisy signals from system logs, which usually correspond to daemons or monitoring information; bursts of error messages; and warning messages, which are generated both in case of failure and normal behavior. Using data from LANL, they predict failures for the six categories in Table 3.2 (left) and achieve a precision / recall of 93% / 43% at a lead time of 32 seconds on average. In follow-on work [Gai+12], they use correlation analysis building on their previous work, where the authors merge their approach with data mining algorithms. The prediction model uses the work of Gainaru et al. (2011) [GCK12] as a first step to generate signals. Then, after removing erroneous data, they use an algorithm to mine gradual association rules [DLT09] in order to compute temporal correlations between events as well as simple correlation analysis for spatial correlation. This hybrid approach of signal analysis and data mining yields a precision of 91.2% and a recall of 45.8%. Further discussion of this approach is given by the authors in [Gai+13] and in [Gai+14], and the approach is extended to consider locality of failure.

Using OVIS[5], an HPC monitoring tool developed at Sandia National Laboratory (SNL), Brandt et al. [Bra+09] show that statistical abnormalities in system behavior can indicate imminent resource failure. The "goal is to discover anomalous behavior in some metric(s) that precede failure of a resource far enough in advance and with enough reliability that knowledge of the behavior could allow the system and/or application to take proactive action" [Bra+09]. Abnormalities are detected by comparing data to a previously established model of the system; the comparison is done using descriptive statistics, multivariate correlative statistics or a probabilistic model. Testing their approach for Out-of-memory errors on the *Glory* Cluster at SNL, they found that

[5]https://github.com/ovis-hpc/ovis - accessed March 2019

one problem could have been detected two hours before the first manifestation in the log file. As a result, they could predict an Out-Of-Memory (OOM) error. The authors examined the case of Out-of-Memory errors; collecting data over 16 days, they found that when monitoring active memory, one problem could have been predicted two hours before an entry in the corresponding log file appeared, which could be used for further analysis. Multivariate Mahalanobis distance was then used to determine whether any values had deviated from their ideal values, i.e., values a healthy system would show. Such ideal values had been gathered in a previous monitoring phase.

Costa et al. [Cos+14] use spatial and temporal correlation of memory errors to identify unhealthy memory chips on a Blue Gene/P system at Argonne National Laboratory (ANL). If either repetition coincides with an error rate greater than 1 error/second, they find that in 100% of the cases, *chipkill* – a technology used to protect memory [Del97] by reconstructing data from a redundant storage – will be automatically activated. This strategy covers more than 80% of chipkill occurrences and gives better results than using spatial correlation alone. Using data from the *Intrepid (Blue Gene)* system at ANL, they first examine spatial correlation. Not surprisingly, if multiple repeated errors occur at the same DRAM address, chipkill is more likely to be activated. Thus, they attempt to provide predictions based on repeated error occurrence at the same DRAM address. However, prediction based solely on spatial characteristics only yields a prediction coverage of 40%. Taking temporal correlation (i.e., errors per second) into account, however, they were able to predict 100% of chipkill occurrences at a prediction coverage of more than 80%. In addition, the authors deployed a migration-based page protection by reconfiguring the interrupt controller and creating a custom interrupt handler. The authors conclude with an average of 76% of chipkill being avoided by using their approach.

A more advanced method based on correlation and modeled in graphs is provided by Fu et al. [Fu+12]. They deploy an a-priori algorithm to gather sequences of distinct and correlated events based on data from three sites. Furthermore, a modified mine association rule with a standard Apriori algorithm, which only considers events happening at the same node or application or have the same type (hardware, system, application, file system, network), is used to create association rules. These rules are modeled as a slightly modified Apriori algorithm. The rules are modeled as directed, acyclic event correlation graphs, where each vertex represents an event or the co-occurrence of events (e.g. $A \wedge B$ for events A, B) and each edge $A \rightarrow B$ is annotated with the correlation of A and B. They then calculate probabilities along the edges in the graph and, if a specified threshold is reached, they issue a failure prediction. In a later refinement of their approach [Fu+14], where casual dependency graphs are introduced, they achieve

slightly better prediction results.

3.3.2. Rule-based Methods

Rule-based prediction is intended to establish a set of rules, i.e., IF-THEN-ELSE-statements, which trigger a failure warning if certain conditions are met. Such rules are (usually) automatically generated from a training data set and called Rule-based Machine Learning (RBML). Ideally, the set of rules should be as small as possible; a larger set of rules usually leads to the overfitting of the problem, which means that the algorithm is "remembering the training set" [KZP06] rather than using learned relationships [JYS19].

Rajachandrasekar et al. [RBP12] use a rule-based approach to predict node failures with hardware root causes. They offer a lightweight implementation of their method with only around 1.5% CPU utilization overhead on a 160-node Linux cluster. They collect sensor data by periodically querying Intelligent Platform Management Interface (IPMI) and categorizing sensor data by assigning severity levels. Communication is done using a publish/subscribe system on top of a self-healing tree topology. The Fault-Tolerant Backplane (FTB) - "a common infrastructure for the Operating System, Middleware, Libraries and Applications to exchange information related to hardware and software failures in real time." [RBP12] - is then used to publish a warning and notifies subscribed components if a change of sensor state is detected. An example of a rule is, that it triggers a warning if three messages in a row with severity WARNING are collected from the same sensor. Additional rules correct for false flags: CRITICAL events where a value of 0 is read are discarded, because the most probable cause is a sensor failure. Furthermore, a modified `MVAPICH2` version is implemented to demonstrate the effectiveness of their approach. This version triggers proactive process migration upon a predicted failure. However, the work does not include an evaluation of precision and recall metrics, as these rules are mostly human-selected.

Watanabe et al. [Wat+12] propose an online approach to failure prediction based on message pattern recognition. In this work, "Failure" refers to any event classified as such in the system under inspection on a commercial cloud platform across multiple nodes. First, they group messages by identifying overlapping contents, which they assume to be clear text for the targeted application. Then, they identify message patterns, where a pattern is an unordered set of messages within the same time window. The temporal order of incoming messages is ignored as there is no inter-nodal time synchronization. Using classic Bayes' theorem, they then calculate the probability of

a failure occurring when observing a certain pattern. Each incoming message is compared against predefined patterns and the system issues a warning if the probability for this pattern exceeds a specified threshold. Almost 10 million messages were gathered over 90 days; 112 failures were observed. Varying the failure prediction threshold from 0.1 to 0.99 yields a precision of 5% to 37% and a recall of 70% to 67% (time window of 5 min.), outperforming a naive Bayes classifier, which achieves a precision / recall of 4% / 30%. The authors further refine their approach in a follow-on paper [WOM14], where they consider the problem of rare events by calculating the conditional probability of $P(\neg failure|\neg messagepattern)$ to include information on whether a message pattern is "typical" (which is a second threshold they introduced to specify this). Unfortunately, no overall precision / recall is given; authors only state some values for different failure types. Precision ranges from approx. 10% to 60%; recall lies between 0% and over 90%.

Ma et al. [Ma+15] introduce an algorithm *RaidShield* to predict hard disk failure. The algorithm has two components: *PLATE*, which detects deteriorating disk health and allows proactive measurements, and *ARMOR*, which prevents multiple-disk failures in RAID arrays. The authors find that some SMART attributes are highly correlated with disk failure, especially the number of reallocated sectors. Replacing disks only based on the reallocated sector count yields a false positive rate between 0.27% and 4.5%. Considering the median time for disk replacement in a typical data center and the cost incurred by unnecessarily replaced drives, they conclude that a reallocated sector count of 200 is an optimal threshold for disk replacement. However, *PLATE* does not take into account that disks may fail before the threshold of specific SMART values is reached or multiple disks may fail at the same time. For *ARMOR*, the second part of *RaidShield*, this approach is extended to multiple disks. This is motivated by the fact that many common RAID levels can handle at most two disks simultaneously failing; if a third disk fails while RAID data is being restored, data loss is inevitable. Their approach is then extended to first calculate the probability of failure for a healthy disk group – that is the disk groups with zero individual failures and bad groups – then they exam the disk groups with at least one individual failure. This technique captures 80% of disk group failures.

3.3.3. Mathematical/Analytical Methods

Besides methods based on statistics, other mathematical concepts are also being used to predict failures. In this work, two of them are presented. One example is the first method presented, that is based on a genetic algorithm, which is a biology-inspired modeling method. Genetic Algorithm represents the evolution of a population under the "survival of the fittest" rule, where the fitness function is defined by the user. In this section, two examples using non-

statistical math methods are presented [JYS19].

Zheng et al. (2010) [Zhe+10] apply a *genetic algorithm* approach on RAS (Reliability, Availability, Serviceability) metrics and job termination logs from a 40,960 node IBM Blue Gene/P system at ANL with 163,840 cores. First, they design a set of rules to predict, from a sequence of non-fatal events, that a failure will happen after a specific lead time at a specific location. Second, they collect an initial population, both by selecting promising rules by hand to speed up convergence. As a fitness function, $fitness = (w_1 \cdot recall + w_2 \cdot precision)$ is defined. Weights w_1, w_2, reflect their preferences regarding precision and recall. Evolution then consists of three operations: Selection of promising individuals; crossover of these individuals to generate children; and mutation to increase genetic diversity. From 520 fatal events collected, they use 430 for training and the remaining 190 for evaluation. They redefine positive and negative rates to consider whether the correct location (i.e., rack) was predicted and whether enough lead time was given. They report precision values of 0.4 to 0.3 and recall values of 0.8 to 0.55, both decreasing with lead time (which varies between 0 and 600 seconds). Precision and recall decrease with lead time, which varies between 0 and 600 seconds. Precision for the same algorithm trained with the standard definitions of precision and recall achieves a significantly lower precision and approximately the same recall (worse for shorter lead times). Furthermore, they estimate that their algorithm can decrease service unit loss (i.e., computing time lost because of checkpointing, etc.) by as much as 52.4%.

Thompson et al. [Tho+10] use the multivariate state estimation technique (MSET) to identify failure events on a *Blue Gene/P* system at ORNL. With MSET, a matrix of input data is used to define a transformation Φ mapping input data onto a so-called similarity space. A new datapoint that is similar to existing data is left relatively unchanged by Φ; however, if the new datapoint differs greatly, so does its mapping. The residual, which is the difference between a data point and its image, can be used to predict a failure if it exceeds a predefined value. Using this method, the authors were able to correctly predict four out of six failures and also predict a seventh failure, which was missed by standard monitoring tools.

3.3.4. Decision Trees/Forests

Decision trees use a tree data structure to store classification rules. A tree is a graph data structure without any circles. A non-empty tree contains one designated node that has no incoming vertices - the root node. Based on the analysis of a set of instances, interior nodes are generated, which contain decision rules, and exterior nodes (leaves), which contain the predicted outcome. Starting with the root node, each node directs to one of its subtrees, depending on the

outcome of the rule / test that was mapped to the specific node. Once a leaf is reached, the result mapped to that leaf is returned as the result [Bab+00]. Random Forests were introduced in the 1990s [Ho95] to avoid overfitting complex decision trees. With this, an ensemble of decision trees is generated using a randomized algorithm; the final prediction is then made using a combination of results from individual trees [JYS19].

Rincón et al. [Rin+17] use a decision tree to predict hard disk failures in a dataset from a cloud storage company. The authors use 2015 data (approx. 62k drives, 17m records) to train their model and 2016 data (approx. 81k drives, 24m records) to evaluate their model. The disk records contained 88 SMART parameters, out of which six were chosen after statistical analysis to build a decision tree:

1. Reallocated Sectors Count: Number of bad sectors that have been remapped,
2. Reported Uncorrectable Errors: Number of errors which ECC was not able to correct,
3. Command Timeout: Number of failed operations due to disk timeout,
4. Reallocation Event Count: Number of attempts to transfer data from a reallocated to a spare sector,
5. Current Pending Sector Count: Number of sectors designated to be remapped and
6. Uncorrectable Sector Count: Number of uncorrectable errors when accessing a sector

The authors estimate a regression model and train a neural network. Holes in the training set – that is the missing SMART values – make a regression model or a neural network unsuitable to being able to predict arbitrary disk failures. Therefore, the authors only analyze decision trees further. They construct a binary decision tree with seven levels (root node + six SMART attributes) for prediction. Their prediction results (cf. Table 3.4 correspond to the findings of Pinheiro et al. [PWB07], who identify four promising SMART attributes for failure prediction: Scan Errors, Reallocation Count, Offline Reallocations, and Probational Counts. However, over 56% of all failures do not show a count in these attributes. Even with additional several other SMART attributes such as Power Cycles, Seek Errors and CRC Errors, 36% of the failed drive population is not covered. The authors conclude that SMART signals alone are insufficient for failure prediction of individual drives.

Motivated by practical reasons such as cost reduction and improving customer service, Ganguly et al. [Gan+16] from Microsoft Corporation build a two-stage prediction model (decision tree followed by logistic regression) to predict disk failure. Notably, they use performance measures (e.g., *Avg. Disk sec/Read*, *Avg. Disk sec/Transfer*, and *Avg. Disk sec/Write*) as an additional data source besides SMART values. In addition, *features*, such as derivatives, are used apart from the raw values of SMART attributes. To avoid overfitting, the decision tree (1st stage) is comparably shallow with a depth of four levels (including the root node). In the second stage, the result from the fourth level of the tree is used as an input vector for a logistic regression model. The authors do not provide no numerical measurement of accuracy. However, they do state that the Receiver Operating Characteristic Curve (ROCC) of the two stage model is "distinctly better" than an Support Vector Machine (SVM) model for the same dataset, which has a precision of 75% and a recall of 62%.

Nakka et al. [NAC11] build a decision tree to predict node failure up to one hour in advance. They use the same dataset from LANL previously analyzed by Schroeder et al. [SG10], and also include usage logs in their dataset. After cleaning, curating, and merging logs, the authors use a variety of machine learning techniques on the data, among them REPTree, Random Trees and Random Forests. The random forest approach yields the best result when including both the root cause of the failure as well as usage *and* failure information. They achieve a precision of 73.9% and recall of 81.3%.

Klinkenberg et al. [Kli+17] from the RWTH Aachen University present a node failure prediction approach using time windows in a recent study. They gather a timeline of unlock and lock events from compute nodes of a cluster. Lock events refer to an event where a node stops accepting batch jobs; after an unlock event, batch jobs can be run on that specific node again. The system consisted of two different node types using either *Broadwell* or *Westmere* processors. Dividing this trace timeline into equally sized frames yields many non-critical frames (normal operation) and a few critical frames (immediately prior to failure). For each trace they extract a set of features which represents descriptive statistics such as median, variance, kurtosis, and root mean square. They apply a variety of statistical methods, including logistic regression, Random Forests, SVMs, and multi-layer perceptrons, of which Random Forests yield the best results, but results differ for the two types: On *Broadwell* nodes, increasing time frame duration has no significant effect on precision, while on *Westmere* nodes precision decreases. Mean precision ranges from 90.8% to 96.6%; mean recall is between 91.0% and 95.6%.

The authors also elaborate on the infrastructure used: A five-node Hadoop cluster was used with *HBase* and *OpenTSDB* for data storage, *Spark* for computation and *IMPI-*

tool and *sar* for data gathering.

Soualhia et al. [SKT15] use one month of publicly available job failure data from Google's server fleet to predict job failure. For each job, they extract attributes such as job ID, waiting time, number of finished, killed, failed, etc. tasks within the job, and total number of tasks of this job to test several prediction models. For each job, they extract attributes, such as job ID and waiting time, to test several prediction models. With 94.2% and 85.9%, respectively, Random Forests deliver the best precision and recall, outperforming other prediction algorithms including conventional trees, conditional trees, neural networks and generalized linear models. Prediction results for task failures are even better. The authors further put their prediction model into practice with *GloudSim*, a tool developed by Google to simulate the typical workload on Google's computing infrastructure. Using a scheduler equipped with their prediction approach, the authors were able to reduce job failure rates and optimize execution time by proactively rescheduling jobs predicted to fail. El-Sayed et al. [EZS17] present similar results using the same dataset. Assuming that the Random Forest approach can be used to reschedule failing tasks to a node with more spare resources ahead of task failure, they implement their prediction method into *GloudSim*. They find that all in all, the number of failures decreased whereas the number of successes increased both on job and task level. Additionally, job execution times were optimized.

El-Sayed et al. [EZS17] base their analysis of job failures in part on the same dataset as Soualhia et al. - it comes as no surprise that they reach similar conclusions. Using a Random Forest model, they achieve a precision of 80% and recall of 50% by only taking into account the information known at the start time of the job and the resource usage in the first five minutes of job execution. Introducing an additional flag into the model which takes the value TRUE as soon as a single task failure of one of the tasks of the job is detected increases precision and recall to 95% and 94%, respectively. Slightly lower values are achieved when predicting the failure of individual task attempts.

Chalermarrewong et al. [CAS12] use a two-stage model to predict hardware failures including overheating, resource exhausion, bad sector access or power supply failure. First, they calculate prediction values for several of the input parameters using an AutoRegressive Moving Average (ARMA) model. Additionally, the results of the model are regularly compared to the actual data. If a statistical *t-test* indicates significant deviation of the predicted values from the actual data, they initiate a re-training of the model. The ARMA predictions are then used to conduct a FTA [RS15]. The authors define threshold values for the outputs of the ARMA model and according to these thresholds they map the leaves of the tree to 0 or 1. These boolean values are propagated toward the root node, where, finally, a binary prediction is calculated. For evaluation,

the authors use the simulator system *Simics*[6] to simulate eight virtual machines, where their model yields a precision of 100% and a recall of 93%. No evaluation in a real-world physical system is done.

Guan et al. [GZF11] present an integrated approach that uses a collection of Bayesian models to detect anomalies which are then classified as failure by a system administrator. A decision tree is further used for prediction. This decision tree is generated algorithmically by defining a *gain function* $G(x_i, n) = H(n) - H(x_i, n)$, where $H(n)$ is the entropy of node n; and $H(x_i, n)$ is the sum of entropies of all child nodes of node n when splitting at attribute x_i. Since the algorithm may create an overfitted or unnecessary complex tree, a part of the dataset unused for training is then used for pruning. The work uses a cloud consisting of eleven Linux clusters at the University of North Texas as testing ground where performance data is gathered using `sysstat`[7]. No quantitative statement for precision and recall are given.

Sîrbu and Babaoglu [SB16] use one month of Google job failure data to predict node failures within 24 hours using an ensemble of Random Forests. Data points were classified as SAFE (no failure) and `FAIL` (failure). Since the `SAFE` class was much larger, random subsampling was used for these points. Attempts at reducing the feature set via principal component analysis or correlation analysis were not successful. Sîrbu and Babaoglu found that using a simple Random Forest approach did not yield promising results, which is why they opted for an *ensemble* approach combining individual predictions using precision-weighted voting. With this approach, precision ranges between 50.2% and 72.8% and recall between 27.2% and 88.6%, depending on the size of the SAFE dataset.

3.3.5. Regression

Linear regression models the influence of one explanatory variable or a linear combination of several variables $x_1, ..., x_n$ on a dependent variable y. Several variations of this approach exist: the standard linear regression model assumes that y is continuous, whereas the logistic regression model assumes that y is categorical, making this model especially useful for (binary) failure prediction. The autoregressive model assumes that the independent variable y depends linearly on its own preceding values. Together with moving average models, which assume that the output value depends linearly on the current and past values of a stochastic term, these two models form the so-called AutoRegressive Moving Average (ARMA) models, which are frequently used in time series analysis [JYS19].

[6] `https://www.windriver.com/products/simics/`, accessed in December 2018
[7] `https://github.com/sysstat/sysstat`, accessed in December 2018

Expanding on their prior work [Liu+11a], which uses a simple moving average, Liu et al. [Liu+11b] use an autoregressive model to predict failures in long-running computing. They further use a bootstrapping approach [ET93] to avoid problems coming from small sample sizes and a Bayesian framework to account for the fact that in long-running systems there is a learning effect, which manifests itself in increasing reliability and failure rates following an exponential function (with negative power). Using data from LANL, they find an accuracy of 80% for their method.

3.3.6. Classification

Statistical classification - or commonly known as perceptions - aims to classify a new observation into a specific category when a number of observations whose category is known already exists. Classification is similar to clustering, where determining the categories is part of the problem. Support Vector Machine (SVM) are binary linear classifiers that aim at maximizing the distance between observations and (linear) classification line. The idea behind SVM is to project the dataset into a higher dimensional space where a linear hyperplane separating the classes can be found [BGV92] [JYS19].

The k-nearest neighbor method is another classification technique commonly used for higher-dimensional data. Each datapoint is classified by examining the `k` *nearest neighbors as measured by a given distance metric in the training data set.* k-means clustering *is a technique where each datapoint is assigned to the cluster whose mean is nearest to the datapoint. In this approach, no classification is given [JYS19].*

Zhang et al. [Lia+07] use a statistical classification to predict failures in a *Blue Gene/L* system, where "failures" are any event declared as FAILURE in the system logs. 142 days of logging data are divided into equal-sized time windows. Using data from the current window and a specified number of previous ones, a failure is predicted for the next time window. A time window of several hours gives enough head time to perform prophylactic measures. After identifying features (key features that include the number of events of any severity in the current time window / observation period; time distribution of events; and time elapsed since last fatal event) and normalizing numerical values, the authors apply three classification techniques: (1) RIPPER, a rule-based classification algorithm [Coh95], (2) a support vector machine approach with radial basis function, which comes with a significant drawback in the form of high training cost (ten hours in this specific case), and (3) a bi-modal nearest neighbor predictor, which uses two distance values for classification. With a twelve hour prediction window, the bi-modal nearest neighbor, RIPPER, and the support vector machine perform reasonably well with a precision / recall of approx. 55% / 80% (approx. 70%

for RIPPER). As the time windows decrease in size, however, performance becomes significantly worse, with SVM facing the fastest decline. The bi-modal nearest neighbor predictor works well, achieving better results at six hours, four hours, and one hour than the other classifiers.

Nonnegative matrix factorization, a generalization of k-means clustering [DHS05], is used by Thompson et al. [Tho+10] to predict failure (as defined via log events) at Oak Ridge National Laboratory's *Blue Gene/P* system. First, data from both logfiles and hardware is gathered in a data matrix D, where each column represents a different point in time and each row represents measurements from a different sensor. Since all entries of D are positive, nonnegative matrix factorization yields two matrices W and H with $D \approx W \cdot H$. W then gives the mean of the clusters and H indicates whether a datapoint belongs to a cluster. The classification into fault and non-fault data is then refined using a Generalized Linear Discriminant Analysis. Using a sliding window approach, which raises an alarm if two out of the last three signals indicate fault, this method correctly predicted five out of six failures.

Zhu et al. [Zhu+13] use both an SVM approach and a neural network approach (cf. Section 3.3.8) to build prediction models for disk failures. Using data from a 20,000 disk population at a Baidu[8] datacenter in China, they both consider SMART values and changes in SMART values when building their models. An SVM with a radial basis function is able to achieve false alarm rates (FAR) lower than 0.03% with a recall of 80%; higher recall values come at the expense of a greater false alarm rate or a longer training window. The average lead time for a prediction with a FAR / recall of 68.5% / 0.03% is 334 hours.

Apart from predicting node failure, Pelaez et al. [Pel+14] focus on low overhead and online capabilities of their prediction method, which is a decentralized online clustering algorithm running on distributed nodes. The n-dimensional observation space is divided into several regions, where each region shows a higher-than-average density of observation points. Then, each region is assigned to one node that determines clusters and outliers and, if necessary, communicates with other nodes covering adjacent regions. To lower the false positive rate of the algorithm, the authors apply two additional techniques: Temporal correlation, where several time intervals are used to detect outliers; and multiple clustering which works on several different feature sets. A precision of 52% and a recall of 94% are achieved for predicting node failure on 32 processing nodes of the *Stampede* supercomputer. The runtime of the algorithm grows sublinear in the

[8] `www.baidu.com`

number of events and higher than linearly with increasing numbers of nodes. Still, a test on several thousand notes yields acceptable results of approx. 2% CPU utilization.

Xu et al. [Xu+10] chose a k-nearest neighbor method to predict failures in a distributed system with 200 nodes. Notably, their approach involves mapping higher-dimensional input data to a lower-dimensional space and using Supervised Local Linear Embedding (SLLE) to extract failure features. They find that SLLE outperforms a simple k-NN predictor and achieves a precision / recall of 66.4% / 59.3% when predicting failure in a file-transfer application.

3.3.7. Bayesian Networks and Markov Models

A Bayesian network is "a graphical model for representing conditional independences between a set of random variables." [Gha02]. A directed arc from a node A to a node B means that B is dependent or conditional on A; the absence of an arc means independence. The Hidden Markov Models (HMM) [Gha02], which have been prominently used in speech, handwriting and image recognition, introduce a further assumption to regular Markov models: They assume that the process generating the model is unobservable. For each state S_t of this process, however, the Markov property still holds: The current state depends ***only*** *on a limited number of previous states [JYS19].*

Yu et al. [Yu+11] differentiate between period-based and event-driven prediction. For both, they apply a Bayesian network to predict events of severity FATAL using RAS logs from the *Intrepid (Blue Gene/P)* system at ANL. They find that the optimal duration of the observation window differs significantly for both approaches (10 min vs. 48 hrs), with the event-driven approach being more suitable for short-term predictions and the period-based approach being more suitable for long-term predictions. The period-based approach is less strongly influenced by lead time than the event-driven approach.

Agrawal et al. [AWR15] use a Hidden Markov Model to predict failure at the software level of a Hadoop cluster. In a preprocessing step, they use clustering to categorize errors from log files into six categories: Network, Memory Overflow, Security, Unknown, Java I/O, and NameNodes. Then, they apply the Viterbi algorithm [For73] to find the optimal (hidden) state sequence of the model. Using 650h hours of training data from an 11-node Hadoop Cluster, the authors find a precision of 93% and a recall of 91% for their prediction model. Regarding scalability, they find that with their approach, execution time increases in data size and decreases with number of nodes, although

not linearly.

3.3.8. Neural Networks

Most machine learning and statistics textbooks (e.g., [Pat98; FHT01; RN03]) discuss Neural Networks (NNs), which have become a trendy topic in recent years. Although neural networks can be still classed into the group of perception (that is classifiers), I decided to pull this method out to emphasize its prominence. Similar to Support Vector Machine (SVM), NN is based on the idea of projecting (data) into higher dimension to find (linear) separation between these data [JYS19]. Zhang [Zha00] summarizes three important points, which should suffice for this section: (1) Neural networks "are data driven self-adaptive methods". They are (2) "universal functional approximators in that [they] can approximate any function with arbitrary accuracy", and (3) they "are nonlinear models", which makes them especially compelling for practical applications [JYS19].

Using the above mentioned Google dataset, Chen et al. [CLP14] use a Recurrent Neural Network (RNN) to predict job failures from task failures. An advantage of RNNs compared to other methods is that they consider interdependencies across time. Task priority and number of resubmissions, resource usage (including only mean CPU, disk and memory usage, unmapped page cache, and mean disk I/O time) and user profile, which is available in the Google dataset, are used for prediction. The prediction achieves an accuracy of 84% and a recall of 86% at the task level. Accuracy at the job level rises from more than 30% after one quarter of the job execution to more than 40% at the end of the execution; recall rises from more than 40% to approximately 70%. The authors estimate that their prediction could save 6% to 10% of resources at the job level when making predictions at halftime using prediction thresholds that are not too aggressive.

In comparison with an SVM, Zhu et al. [Zhu+13] test a three-layer neural network with a sigmoid function as activation function on hard disk failure data from 20,000 drives. While offering a higher recall (94% to 100%), the neural network also comes with a higher false alarm rate of 0.48% to 2.26%. The authors suggest to using a boosting method such as *AdaBoost* to further improve the performance of the neural network.

In a recent study, Islam et al. [IM17] propose a Long Short-Term Memory (LSTM) network to predict job failures using the Google dataset. Extending recurrent neural networks with short-term memory, LSTM networks are especially suitable for long-term temporal dependencies. Both raw resource measures and task attributes are fed into

the network, which achieves slightly better results for task failure prediction than for job failure prediction. A precision of 89% and a recall of 85% for task failure prediction is given. Results for job failure prediction are slightly worse with a precision / recall of 80% / 83%. Both were found to outperform an SVM using the same data. Finally, the authors find that even when resubmitting tasks, their approach leads to considerable resource savings (e.g. 10% of CPU time even when resubmitting failed tasks 5 times). They also state that up to 10% of CPU time and other resources can be saved using their predictive approach.

In the latest study by Nie et al.. [Nie+17], the authors present an in-depth analysis of the correlation between temperature, power and soft errors in GPUs and propose a neural network based prediction method, `PRACTISE`, for predicting soft errors in GPUs. According to our knowledge, this is the only reference dealing with GPU failure prediction. The evaluation on 4 cabinets (384 nodes) on the *TITAN* supercomputer shows a precision of 82% and a recall of 95%.

3.3.9. Meta-Learning

Meta-Learning is a method, in which the metadata of a given dataset is used for machine learning. An intuitive interpretation is that meta learning is the combination of several individual predictors to improve overall precision accuracy. In modern machine learning terms, the term deep learning is often used as a synonym [JYS19].

Gu et al. [Gu+08] present one such approach: On 130 weeks of data gathered from an IBM *Blue Gene/L* system, they test an algorithm that consists of three steps:

1. Meta-Learning: The meta-learner gathers several "base predictors" that are individually calculated from the last k weeks of data (sliding window).
2. Revision: The Reviser uses the phenomenon that failures in high performance computing usually show temporal correlation to generate an effective rule set by applying the rules gathered in the first step to data from the k-th week.If a rule does not perform well according to some measure (e.g., ROC analysis), it is discarded.
3. Prediction: Finally, the predictor monitors runtime events and predicts failures for the $(k+1) - th$ week.

The authors test a Java implementation with the *Weka Data Mining Tool*[9] for prediction

[9] `https://www.cs.waikato.ac.nz/ml/weka/`

and an Oracle database for storing knowledge on a *Blue Gene/L* system. In step (1), they choose one rule-based classifier and one statistical method. Precision and recall stabilize at 0.9 and 0.7, respectively, after increasing for the first ten weeks for $k \in \{2, 4, 6, 8, 10\}$. While a small value of k is initially necessary to build a rule set, the window size can be increased later on when the prediction accuracy has saturated.

In follow up work [Lan+10], Gu et al. provide a more detailed analysis of the meta-learning approach mentioned above. Now, they consider that severity levels of IBM *Blue Gene/L* logs might not correctly indicate underlying failures and develop a categorization of fatal events in cooperation with system administrators. A second data preprocessing step involves temporal and spatial filtering of log data and for the meta-learning step they use a third base predictor: They calculate a CDF of failure occurrence and give a warning if the probability of a failure occurring is higher than a specified threshold. For evaluation, two systems at San Diego Supercomputing Center (3,072 dual-core compute nodes) and ANL (1,024 dual-core compute nodes) are used. A dynamically increasing training set yields better prediction results than the static training but then training has a considerable overhead. Precision and recall range between 70% - 83% and 56% - 70%, respectively.

Alvaro et al. [Fra+19] introduce a Deep Neural Network (DNN) method using the combination of different neural networks to predict upcoming failure in the *MOGON* cluster at JGU Mainz. Different networks using different length of lead time windows are trained using a dataset collected over 6 months. These networks are then combined together using a "majority voter"-like structure. The authors focus on the previously under-researched problem of *false positive prediction*, which becomes prominent for large-scale jobs requiring many nodes. While recall values (ranged between 85% and 73%) decreases with increasing number of networks used, the precision values (ranged between 96% and 99.5%) increases. With four networks, the false positive rate becomes neglectable with 0.04%.

3.4. Insights and Discussions on Failure Predictions in High Performance Computing Systems

By screening more than 70 publications regarding failure prediction systems in HPC and Cloud environments, a total of over 30 papers with best-performance results were selected and presented above. These trends and gaps are presented and discussed in the next chapter.

3.4.1. Effectiveness of Failure Prediction System

With Table 3.4, it is easy to observe that tree-based methods, such as random forest algorithms, have shown excellent prediction results in terms of *precision and recall*. Classic machine-learning methods such as NN and SVM perform worse than tree-based methods. The reason for this is the unbalanced distribution of features and classes used for training, as the failure events are rare [SB16]. The emergence of tree-based methods is also observable because the most dominate methods in the 2010 survey of Salfner et al. [SLM10] use neural networks. Analytical and rule based methods also provide good results with easy setup bound to specific systems. However, due to the selection of rules, these methods are hard to generalize. Meta learning methods are gaining increasing prominence with the recent boom of research in the field of *deep learning*. By combining many sources and features, these methods are able to deliver good results. Furthermore, these methods [Gu+08; Lan+10; Gai+14; AWR15; GCK12; Pel+14] show low runtime overhead, making them useful as online learning methods [JYS19].

With Table 3.4, it is easy to see that most of the prediction methods do not pinpoint the specific fault or error (that is, the root cause) of the failure. This is most likely due to multiple reasons: First, most data collected from datacenters in its raw format do not provide information on the root cause. They just contain operation data regarding system state and failure information. Exploring the root cause is a manual and expensive step. Second, the backtrace of the root cause usually does not add much value because the aim of the failure prediction itself is to prevent user applications from failing. Nevertheless, we think it is important to pinpoint the root causes for the operators of these systems, in order to understand the weakest part within the "Liebig's Barrel". While disk and memory failures are easy to predict nowadays due to advanced failure avoidance and monitoring, *RAID*, *chipkill* and *registered ECC*, emerging components including GPU is under-researched. With even newer technology such as Non-volatile Memory (NVM) being deployed in data centers, the understanding and prediction of component failures may become challenging yet important for modern data centers [JYS19].

A further note is, that similar algorithms do not necessarily produce the same prediction results; the training parameters (known as *parameter tuning* in modern machine-learning textbooks) are significant for the prediction results. In addition, as Klinkenberg et al. [Kli+17] point out, the feature extraction and selection steps are also vital for the performance of a prediction system. However, many papers do not show the detailed parameter tuning and feature selection steps and do not provide analysis on the potential correlation between bias in the measurements and potential failure conditions. This leads to contradictionary conclusions, making it hard to comment on a the general-ability of specific phenomena [JYS19].

3.4.2. Availability of Datasets, Reproducibility of Research

In other research fields dealing with machine learning, a common dataset is used to ensure the reproducibility and generality of results. For example, the Common Objects in Context (COCO)[10] [Lin+14] is used in most of the publications in the fields image segmentation, text recognition, computer vision, and much other image-related research. This large-scale controlled dataset makes the comparison and reproduction of results very easy. _This is clearly not the case in failure prediction. Although over 70 papers have been screened in this work, only a limited amount of undocumented and outdated data sets exists. Prior to the publication of the Google Job Dataset in 2011, no studies on job failures had existed. Half of the datasets provided in the CFDR database are not accessible and all of them lack suitable metadata or proper description. This limitation significantly hardens the research in this field and closes the door to researchers who do not have access to such systems. Moreover, modern machine-learning approached such as *transferred learning* cannot be used [JYS19].

3.4.3. Metrics and the Effect of False Positive Rate

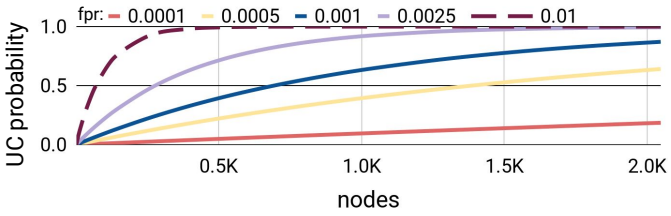

Figure 3.1.: Probability of Triggering an Unnecessary Migration or Checkpoint Due to Certain False Positive Prediction Rates [Fra+19].

Researchers criticized these metrics in the past because they did not account for the loss of compute time or the overhead for migrations and checkpointing [SLM10; Tae+10; Zhe+10; EZS17]. The high false positive rate is one prominent issue because it triggers unnecessary checkpoints or proactive migrations [EZS17]. The *fallout* is always calculated based on the probability of misclassifying an ok event for a single node. The problem therefore significantly increases with the maximum job size and consequently with the number of nodes, making failure predictions useless for practical use in large-scale HPC centers. Most notably, in a previous work [Fra+19] we point out that with any large job an unnecessary fault-mitigation becomes almost certain.

[10] `http://cocodataset.org/home`, accessed in December 2018

Although the significance of the false positive rate for the overall performance of a fault prediction system is very high, none of our surveyed paper focus on this problem. In fact, our previous work [Fra+19] was the first work to introduce a metric that had practical importance for a large-scale batch cluster system.

3.4.4. Failure Prediction and Fault-mitigation Techniques

To conclude, current fault prediction methods presented in literature are able to predict errors with practical performance. However, the problem of false positives is not well understood and needs to be further researched. Since the overhead of performing fault-mitigation upon any predicted error varies, a sufficiently low overhead fault-mitigation method is required to support the fault-predictions, which produce a fairly large amount of false positives nowadays. To understand the different fault-mitigation techniques, we will present the state-of-the-art methods for fault-mitigation and failure avoidance in the next chapter.

4. Fault Tolerance Strategies

In Chapter 3, we worked through a summary of the current state of the practice in the field of failure prediction. Several shortcomings have also been identified. Most prominently, the false positive rates are neither covered nor discussed in existing literature or methods [JYS19]. Without this information, an accurate evaluation of a real-world system with different fault mitigation techniques cannot be conducted.

To introduce different state-of-the-art methods involved in fault management, the scope of this dissertation - the Batch Job Processing Clustering System (BJPCS) is introduced in this chapter. Followed by a taxonomy of methods for failure handling, the concepts of *checkpointing* and *migration* are explained in detail.

It is not possible to compare different fault mitigation techniques without a proper context. Consequently, we do not aim to provide a manual on fault mitigation strategies. Instead, the focus and requirements are introduced and discussed in this chapter.

4.1. System Architecture of Batch Job Processing System

The focus of this dissertation is the Batch Job Processing Clustering System (BJPCS), which includes both classic HPC systems, but also some of the modern cloud systems such as the *Amazon AWS Batch System*[1] and *Google Cloud Platform*[2]. We define BJPCS as stated in Definition 4.1.1. The analogy among these systems is the use case, which is illustrated in Figure 4.1.

Definition 4.1.1. Batch Job Processing Cluster System A *Batch Job Processing Clustering System (BJPCS)* is a distributed cluster system, consisting of a set of physical computing nodes (servers), that can handle a batch (scripted) job from a potential user synchronously or asynchronously. The batch job is executed according to some schedule on the processing cluster, and the execution results are delivered to the user.

A user of such a system provides a particular application and corresponding data, which is called a *job*. This job is submitted to a *job processing queue* through a frontend node. According to a certain *scheduling algorithm*, the jobs in the job queue are scheduled to a

[1] `https://aws.amazon.com/batch/`, accessed in March 2019

[2] `https://cloud.google.com/dataflow/`, accessed in March 2019

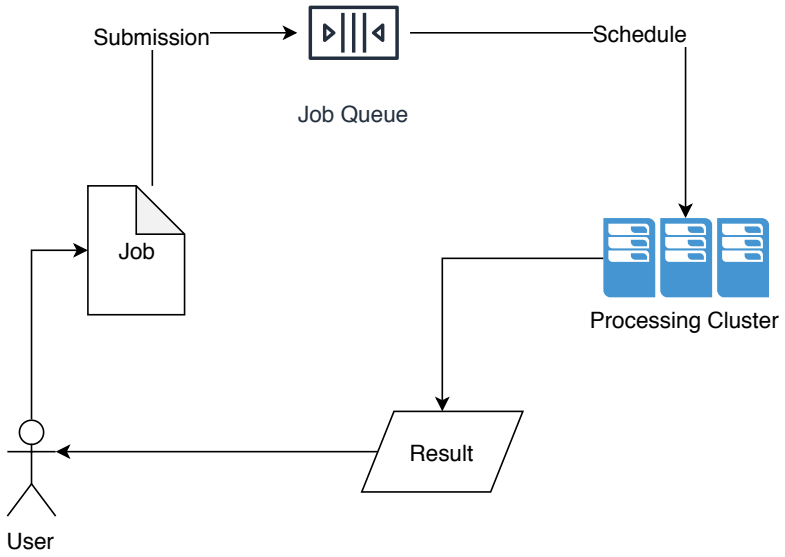

Figure 4.1.: Typical Use Case for a Batch Processing System

processing cluster. The cluster then processes each job and returns the result according to the respective user's instruction.

In the scope of this work, only failures which occur in the *cluster* are considered because it is the most crucial part of such a system. Furthermore, as most of the servers and computers in such a system are in the processing cluster, failures are also more likely to occur there. Moreover, since the user does not have the control nor is allowed to access the processing cluster, a "hot " failure cannot be addressed by the user online. Hence, user-provided applications must be capable of overcoming these failures.

With this background, one needs to have an overview of the hardware and software architecture of such a system. Although this is an emerging field, there is a generic design, which is visualized in Figure 4.2. Components such as frontend (login) nodes and management nodes are not included in Figure 4.2. However, these components are specialized nodes that can also be treated as a simple "node" for modeling purposes.

The components of such a generic cluster, including the applications and its components running on the cluster, are described as follows:

- Node: Node is the hardware entity of this cluster. It is typically a *bare metal server*, which contains a specific configuration of the hardware.
- Operating System (OS): The system software on each node, including the software

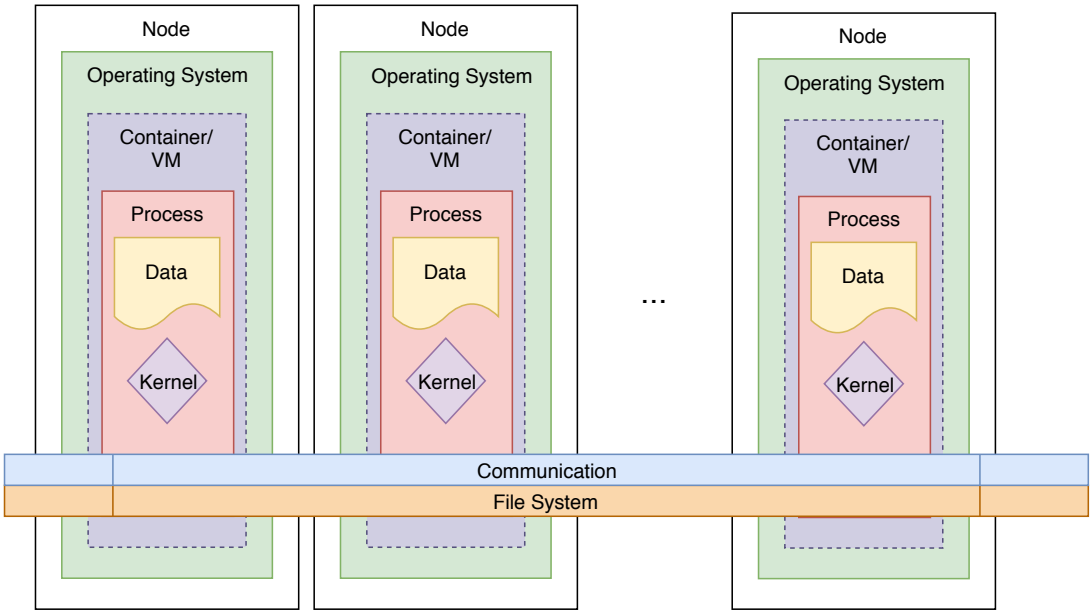

Figure 4.2.: System Architecture Overview of a Generic Batch Processing Cluster

that is responsible for working with the intra-node resource manager such as *SLURM*[YJG03].

- Container and Virtual Machine (VM): Recent approaches [Pic+18; Pic+16a; Pic+14] suggest placing the actual application in containers or VMs. This way, with proper support of the underlying OS and hardware, these containers can be migrated to another node. The dashed line in Figure 4.2 also indicates that it is optional. In fact, as far as the author knows, these techniques are not used in any productional environment for an HPC system yet; however, almost all cloud systems are based on either VMs or container.
- (Application) Process: In this case, we mean the user applications' process. There can be multiple processes on the same node.
- Data: Any useful application in a batch processing job is designed to process some data. By *Data* we refer to the application data that is to be processed by a specific kernel.
- Kernel: In a useful application, the (computer) kernel is the subroutine (or part of the program) that is responsible for processing the data. In a well-engineered application, a kernel should be decoupled from the data.

- Communication: This is the crucial part of connecting the independent server nodes to a whole processing cluster system. Typically, this includes both hardware and software working together. Examples of the hardware (physical layer) include standards such as *InfiniBand* and *OmniPath* for HPC systems, and usually *Gigabit Ethernet (GbE)* or *Terabit Ethernet (TbE)* for the cloud systems. Software for user-level communication (transport layer and above) is usually a library that handles data transfer, such as *Message Passing Interface (MPI)* for HPC or *Transport Control Protocol (TCP)* and *User Datagram Protocol (UDP)* in cloud systems.

- File system: A distributed or shared file system is also crucial for the operation of a batch processing cluster. Hardware for the shared file system includes Network Accessed Storage (NAS) and Storage Area Network (SAN). As for the software, alongside with the well known *NFS*, examples include *IBM Spectrum Scale (GPFS)*[3], *Intel Lustre*[4], *Google Cloud Filesystem*[5], and *Distributed File System (DFS)*[6].

Typically, such a batch cluster is designed to tolerate a single point of failure. However, from our knowledge, some of the crucial parts are more likely to propagate faults and errors to another part of the system. Among these parts, the communication components and the file system are common troublemakers, as these are shared resources from all nodes. For example, if a network switch is malfunctioning, all the nodes connected to that specific switch may suffer from the fault. Any application running on those nodes would fail, and most likely there would be no prediction available on a per-node basis.
In the following section, different methods for fault mitigation are introduced and discussed.

4.2. Fault-mitigation Mechanisms

Fault-mitigation is the required step of reacting to a failure condition. Fault-mitigation can be achieved on both hardware and software levels. For example, the well-known Error Correcting Code (ECC)-Memory is a hardware technology that adds parity bits to detect and mitigate internal data corruption. In this dissertation, the focus of fault-mitigation techniques is on the software and the user application.

[3] `https://www.ibm.com/us-en/marketplace/scale-out-file-and-object-storage`, accessed in March 2019

[4] `http://lustre.org`, accessed in March 2019

[5] `https://cloud.google.com/filestore/`, accessed in March 2019

[6] `https://docs.microsoft.com/en-us/windows-server/storage/dfs-namespaces/dfs-overview`, accessed in March 2019

In the terminology of systems engineering, there are two types of strategies:

- Fail-Safe: The fault does not lead to harming of other components, the environment or human life.
- Fail Operational: The system can provide continuous operation, despite the occurrence of a fault. A fail-operational system is often also referred as a **fault-tolerant** system.

However, this terminology does not fit directly for fault topics on HPC and cloud systems. In this dissertation, we introduce the the following definitions for fault-tolerant applications:

Definition 4.2.1. Fail-Safe Applications
An application (for a batch cluster system) is considered **fail-safe**, if it is terminated properly upon an error or fault and does not propagate its fault across the system, affecting other parts of the whole system. Besides, it should be able to be re-executed without user intervention.

Definition 4.2.2. Fail-Operation Applications
An application (for a batch cluster system) is considered **fail-operational**, if it can continue execution even after encountering a fault.

The simplest example of a *fail-safe* method is termination. An application instance terminates if any fault is detected. This way, it is certain that no further error propagation would happen. Today's job schedulers would kill any job that has encountered a fault. Moreover, most applications are not designed to tolerate node failure, causing them to fail automatically due to the unsuccessful access to resources. However, this has no value in terms of fault-mitigation. Any reasonable BCS nowadays provides at least the *auto-requeue* function or known as *retry*, which automatically sends the failed job back to the job queue and reschedules it to be done later.

For all fail operational strategies, two subcategories are the most important:

- Fault-mitigation: A fault-mitigation approach is also known as **reactive** fault tolerance. It is based on the detection of a fault occurrence and reactively deploys a mechanism to handle any failure that occurs.
- Failure avoidance: Failure avoidance is also called **proactive** fault tolerance. It is based on the idea of forecasting upcoming potential faults and actively avoids any failure. The usefulness of failure avoidance methods relies heavily on the quality of the prediction about these faults.

4.2.1. Overview of Fault Tolerance Techniques

To help to provide and overview of the vast amount of different techniques for fault tolerance, we have created a taxonomy of strategies shown in Figure 4.3. Recalling the Definitions 4.2.1 and 4.2.2, we have grouped the fault strategies *termination* and *auto-requeue* into the fail-safe strategies because they only limit error propagation and do not save previous work. The concept of *checkpoint&restart* is a strategy which provides basic fail-operation, but a significant overhead is expected as the application needs to be restarted and potentially a significant workload has to be redone. Both *migration* and *Algorithm-based Fault Tolerance (ABFT)* are included in the fail-operation category because they provide continuous operation, even when encountering a fault. However, *migration* relies on the support of modern hardware and software, and typically some prediction on *upcoming* faults.

VM and *container migration* are fault avoidance methods, which focus on the prevention of any upcoming fault. Therefore, they rely on sufficient prediction systems to predict an upcoming event. However, they have the advantage of being *application transparent*, which means that the existing application does not need to be changed for supporting fault tolerance.

Checkpoint&restart is **the** state-of-the-art technology for fault tolerance and the sole efficient method for handling random, unpredicted faults. It can be both *application transparent* and *application cooperative* by checkpointing on different levels. On the system level, components such as VMs and containers can be checkpointed. On the application level, user-space applications can checkpoint their intermediate result (data). This method can also be combined with migration based methods to provide both proactive and reactive fault tolerance.

Algorithm-based Fault Tolerance (ABFT) is an application-integrated approach for providing fault-tolerant algorithms in math-intensive applications. The nature of it makes it unsuitable for more service-oriented cloud systems. These methods are discussed in detail in the following section.

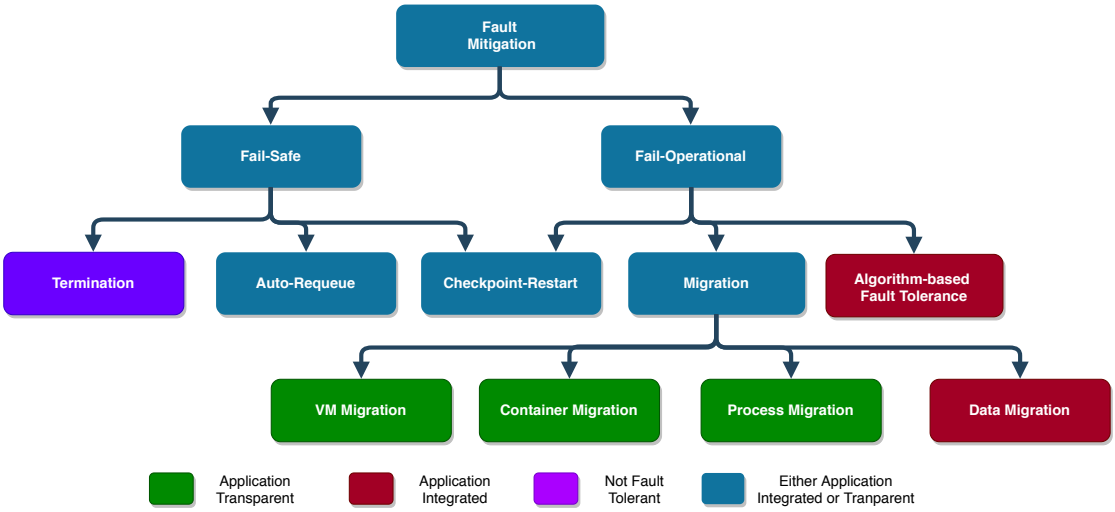

Figure 4.3.: Taxonomy of Different Fault-handling Techniques

4.2.2. Application-integrated vs. Application-transparent Techniques

Another way often used to classify different fault-handling approaches is bound to the role of applications. As previously shown in Figure 4.2, there are many (logical) components in which faults can be handled. As a result, two classes of fault tolerance approaches can be derived:

- Application-integrated: An application-integrated fault tolerance strategy requires the modification of the implementation.
- Application-transparent: An application-transparent fault tolerance strategy relies on the system-level components such as OS, VM or containers.

For new applications, application-integrated methods usually deliver better performance, as fault tolerance can be programmed tightly into the application. However, for an existing application, an application-transparent strategy often yields better cost efficiency because the redevelopment of an existing large-scale application is very costly. The selection of method depends heavily on the user case.

4.2.3. Checkpoint and Restart

Checkpoint&Researt [Ell+12; JDD12; Wan+10] is still the state-of-the-art technique for fault-mitigation both in the fields of HPC and cloud systems. The principle is straightforward: The application creates a checkpoint periodically. The period for a checkpoint

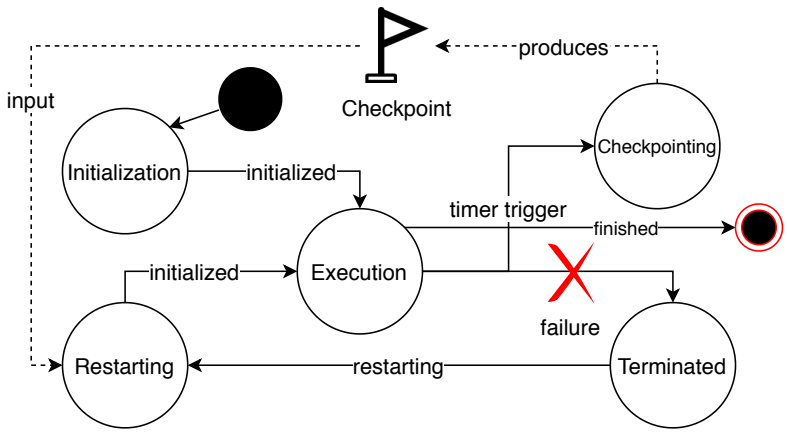

Figure 4.4.: State Chart for Classic *Checkpoint&Restart*

is related to the size of the application, the MTBF of the system and many other factors. Upon encountering a fault, the application can be terminated and restarted from a previously saved checkpoint on a different set of nodes. This way, depending on the frequency of checkpointing, only a minimum amount of work needs to be redone on a failure. This mechanism is visualized in the state chart in Figure 4.4.

The process of *checkpoint&restart* is described as follows:

- The application is initialized and executing normally.
- At a given frequency and a predefined timepoint a checkpoint is created to save current work progress (checkpoint).
- The application continues executing.
- If a fault occurs, the application is restarted and initialized using the previously secured checkpoint (restart).
- The execution then continues until the application finishes.

Combined with a fault prediction system, the frequency for a checkpoint can be reduced. With excellent predictions, it is also possible to trigger a checkpoint on demand. This further reduces the overhead caused by *checkpoint&restart*, adding scalability of *checkpoint&restart*. The adapted state chart for *checkpoint&restart* with fault prediction is shown in Figure 4.5.

As *checkpoint&restart* is a very sound and well understood method, many well-known examples exist in the literature. The most used examples for HPC systems include *Checkpoint/Restart in Userspace (CRIU)* [LH10], *Berkeley Lab Checkpoint Restart (BLCR)* [HD06] and *Distributed Multi-Threaded Checkpointing (DMTCP)* [AAC09].

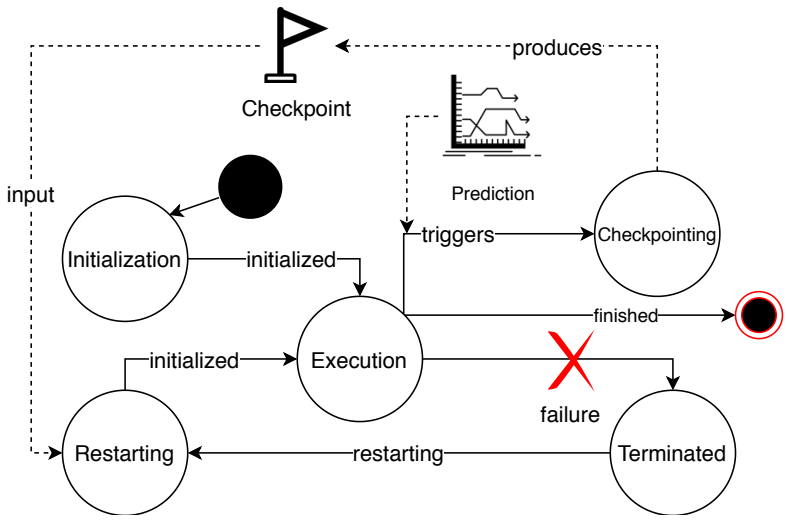

Figure 4.5.: State Chart for *Checkpoint&Restart* with fault Prediction

4.2.4. Migration

Migration is an attempt to reduce the overhead of *checkpoint&restart*. Depending on *what* is being migrated, there are two different approaches well-known among researchers: *process migration* and *VM/Container migration*. Any migration framework would require someone to actively trigger a migration. Ideally, using a fault prediction system, migration can be triggered upon a positive prediction for fault. As the application is never terminated, the overhead of the concept of migration is theoretically lower than *checkpoint&restart*. Furthermore, since the affected parts of an application are only small (that is usually on one node suffering from an upcoming error), the scalability is also better than *checkpoint&restart*. This argument is also used repeatedly in many studies in migration research [Wan+08; Wan+12; Pic+14; Pic+18; Pic+16a]. Due to their nature, since both state information and application data need to be migrated during the migration process, most migration techniques are *application transparent*.
The process of migration is as follows:

- The application is executed normally on the BJPCS.
- A trigger system (such as a fault prediction system) informs the migration system that migration needs to be performed.
- (Optional) The application is informed, synchronized and halted.
- Migration is performed.

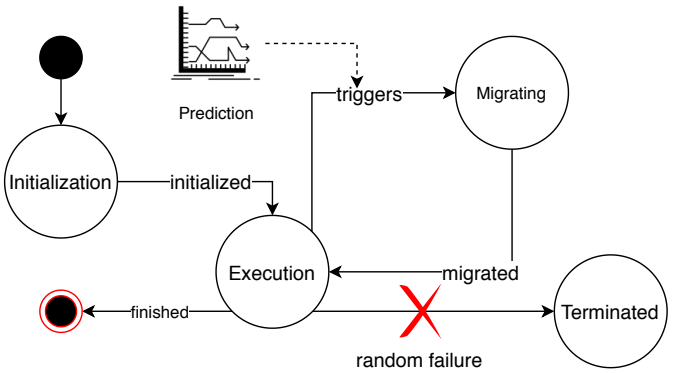

Figure 4.6.: State Chart for Migration

- The application continues normal execution.

This process is also visualized in Figure 4.6.

The big drawback of these migration-based methods is that spontaneous errors which are not covered by the prediction cannot be handled by the migration framework as there is no stateful checkpoint stored on the system for recovery. In this sense, migration is a **failure avoidance** method rather than fault-mitigation. Nevertheless, migration can be used alongside with *checkpoint&restart* to reduce the frequency of checkpoints, thus decreasing the overall overhead resulting from the fault-mitigation.
In the remainder of this section, we will explore the difference between process migration and VM/container based migration method.

Process Migration

Process migration is the way of moving a process to another physical machine [DO91; Mil+00]. It is not a new concept. Consequently, many attempts [Ste96; CLG05] have been made to extend existing programming models such as MPI. The idea of process migration is that with proper support of the operating system and runtime system, a process can be migrated to another machine transparently at runtime. However, process migration is not common in real-world applications, as there are so-called *residual dependencies* on the machine where a process is being migrated from [Mil+00].

One most notable example of such work is from Wang et al. [Wan+08], who extend MPI with migration functionality. To accomplish this job, they extended an MPI implementation with BLCR. A running process state and data is secured using BLCR, and the process is terminated. This checkpoint is then transferred to another node, and the state and data are restored. Afterward, the process can be restarted

on the new node. The proposed solution requires the modification of MPI, as well as the support from a operating system. This limitation is quite significant because the modification of MPI and the particular system requirement are not possible everywhere.

Virtual Machine Migration

Migration of VMs is a well-developed technology nowadays. There is a significant amount of research in VM Migration, including [NLH+05; Cla+05; Woo+07; Pic+18; Pic+16a; Pic+14; XSC13; Voo+09]. The most important use case for VM migration is in the field of cloud computing, where resource elasticity is required, and the VM might be resized and moved to another physical location at runtime. Everyone who has moved a virtual machine from one physical machine to another and restarts the execution has already performed a VM migration manually. One of the first Virtual Machine Managers (VMMs) that supports live migration is *Xen*[7] [Cla+05].

In a modern VM migration approach such as in [Cla+05], live migration is typically performed in the following manner: First, the required resources are verified on the migration target. If all requirements hold, a quick snapshot is created for the VM. The data (such as memory pages), resource requirement information (such as I/O devices) and state information regarding the VM are sent to the new destination, and all the dirty pages and changes are sent incrementally to the destination. At a certain point, the VM on the origin machine is halted, and all remaining states and changes are committed to the target. Afterward, the VM on the origin machine can be terminated, and the VM on the target machine can be started and continues the operation. This way, the downtime (service outage) can be kept within the millisecond range [Cla+05].

VM migration has been tested for HPC systems as well [Nag+07; Pic+16b; Pic+14]. The major problem when deploying VM migration is, that it may consist of special communication interfaces such as *InfiniBand* and *OmniPath*, which do not support a live migration. A major reason for the lack of support is, that as there are requirements on latency and throughput, technologies such as Remote Direct Memory Access (RDMA) are deployed in these systems. Any driver does not yet support the rebuilding of the mappings for RDMA without modification. As a workaround, Pickartz et al. [Pic+18] propose to switch to TCP/IP based communication before migration and switch back to the *InfiniBand* connection after migration. However, this requires the active support by the MPI library or the user application. Another issue is that accelerators in heterogeneous compute nodes such as GPUs and Field Programmable Gate Arrays (FPGAs) cannot be halted easily.

[7] `https://xen.org`, accessed in March 2019

To conclude, VM migration is a prominent and solid method of fault management. In the area of cloud computing, VM migration is already a state-of-the-art technology. However, in the field of HPC, the overhead of a virtual machine and the lack of support for HPC specific hardware and technology are drawbacks.

Container Migration

A modern, modified "flavor" of VM migration is the so-called container migration. A *container* is a technology through which the OS provides multiple isolated user-space instances on top of the same kernel space. The main advantage is reduced overhead compared to VM technology. The most famous examples for containers are *Docker*[8], *LXC*[9], and *Singularity*[10]. The migration of a container requires similar steps. However, only user-space data and state need to be migrated from one to another host. This concept has also been extensively investigated for cloud computing [MYL17; Qiu+17; Nad+17]. In the field of HPC systems, initial work also exists, such as [Pic+14; Pic+16a]. However, the same limitations from system and hardware support are required for container migration.

Combining Migration with *Checkpoint&Restart*

Migration-based methods can also be combined with *checkpoint&restart*. In a cloud environment, this is often used for testing purposes. By checkpointing the system state at a given time, a restart (or rollback) can be initiated. This way, the migration based method can be extended to support random faults[11], which are hard to be supported using migration only. The state chart for this combined method is given in Figure 4.7. The downside of checkpointing (a VM or container) is the significant overhead, as not only application data, but also data that is required by the operating system to facilitate the application instance is subject to being checkpointed.

An example for migration based fault tolerance combined with *checkpoint&restart* is given in Figure 4.7. Its functional principle is summarized as the follows:

- Assuming that the application is running normally, a checkpoint is created at a given frequency.
- A fault prediction system continuously produces predictions regarding upcoming faults.

[8] `https://docker.io`, accessed in March 2019
[9] `https://linuxcontainers.org`, accessed in March 2019
[10] `https://www.sylabs.io/docs`, accessed in March 2019
[11] In this context, by *random fault* we refer to faults which are not predicted by a fault prediction system.

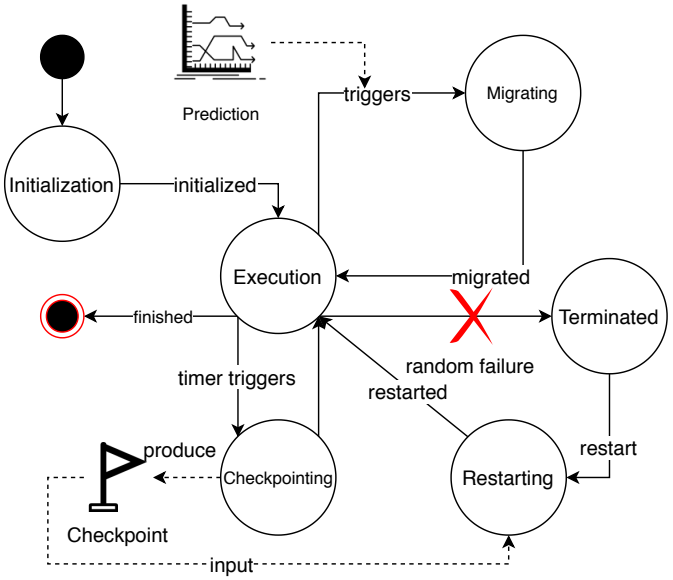

Figure 4.7.: State Chart for Migration with Checkpointing

- If a fault predicted, a migration is done to avoid the failure of the application.
- If there is a random failure, a restart routine is executed to restart the application from the last checkpoint saved.

To summarize, with VM and container migration, two different application transparent means of failure avoidance techniques are introduced. Their most significant advantage is, that the user application does not need to be changed, which increases the usability from the user's point of view. One of their major disadvantages is the relatively high overhead because both migration methods are virtualization-based methods. In addition, these two migration methods rely on the support of the hardware platform. With process migration, the runtime overhead is eliminated because the method directly operates on the application process. However, process migration demands both OS and runtime library support. At migration time, all of the above mentioned migration methods have to migrate more data than just only the useful application data, as they have to reconstruct the application process at the destination location.

It is easy to see that with proper application support, overhead at migration time can be reduced by migrating only useful application data. In addition, the application can make use of existing hardware, OS and libraries to complete the migration. A reconstruction of an application instance is not necessary. In the next chapter, we will introduce *data migration*, an application-integrated migration method, which features a

low overhead and requires neither hardware nor OS support.

4.2.5. Algorithm-based Fault Tolerance

Unlike system-level fault tolerance strategies such as migration and data-level fault tolerance checkpoint-restart, *Algorithm-based Fault Tolerance (ABFT)* is an attempt to provide fault tolerance on the compute kernels of the application. First introduced by Huang and Abraham [Hua+84], ABFT is under active research in many studies [Bos+09; CD08; Ban+90; AL88; Che13], especially for HPC applications, as most of the HPC applications are somehow *iterative*. There are many kinds of ABFT, in the original work by Huang et al. [Hua+84], they introduce a checksum to verify the result from different matrix operations, which can detect faulty operation due to data corruption. A more modern idea of ABFT is, that with proper partitioning the actual error of calculation from a given number of a faulty processors (or other components) can be compensated for and kept at a small level. As *machine error* cannot be fully eliminated anyways, the approximation is considered useful if its margin is small enough. The effect of additional errors can also be kept within an acceptable range.

Since modern applications rely typically on external math libraries such as *Intel MKL* [Wan+14] or *BLAS* [Law+77] to perform mathematical operations, developing one's own ABFT kernel requires significant development effort, especially for existing code. Any ABFT based approach must target these libraries to provide sufficient support for real-world scenarios.

4.2.6. Summary of Fault Tolerance Techniques

To conclude this chapter, Table 4.1 summarizes all the fail-operational methods discussed in this chapter. While *checkpoint&restart* supports a variety of system configurations, migration-based methods have lower overhead. The highly specialized ABFT provides lightweight fault tolerance for HPC applications. However, it cannot handle all kind of errors and does not suit for most cloud systems.

With sufficient fault prediction, migration-based fault tolerance should be more efficient than *checkpoint&restart*. This is also the conclusion from many studies by Pickartz et al. [Pic+14; Pic+16a; Pic+16b; Pic+18]. However, combined with the previously mentioned high false positive rate problem, the decision for migration can be still costly in case of false positives. Nevertheless, without a prediction system, it is generally not possible to recover from a fault by using migration. This limits the use cases for migration-based fault tolerance approaches.

The overhead of fault tolerance approaches has not been quantitatively analyzed in this chapter. The reason is that depending on the use case, the overhead can vary

Table 4.1.: Overview of Fault Tolerance Techniques

Location	Name	Prediction Required?	Special Hardware Required?	Cloud/HPC
Data	Checkpoint Restart	optional	none	both
	Data Migration	yes	none	both
System	VM Migration	yes	yes	both
	Container Migration	yes	yes	both
Algorithm	Algorithm-based Fault Tolerance (ABFT)	none	hone	HPC

significantly. For example, the live migration of the main memory Database Management System (DBMS) takes a significantly higher overhead than *checkpoint&restart* a non-volatile memory based DBMS.

The next chapter deals with the concept of *data migration* in detail and present two different techniques are presented: one technique with system support and another one without system support.

5. Data Migration

Numerous fault tolerance strategies were introduced and discussed in the last chapter. Now, in this dissertation, we are going to focus on the concept of data migration, a promising concept. Data migration is a *proactive and application-integrated fault avoidance* technique from the class migration techniques. As mentioned before, it features a low overhead by design at runtime as well as at migration time because no virtualization is deployed. Instead of migrating a whole Virtual Machine (VM) or a container, the application is in charge of choosing the relevant information to be migrated. This reduces the footprint for the migration operation. It also gives the programmer the options of either reacting to an event or not. Furthermore, since classic applications usually provide support for checkpoint&restart, the selection of relevant data is a natural step. This means that low transition cost for an existing application can be expected by using this approach. However, any existing application has to be modified to support data migration as it is not application-transparent. The trade-off between the development cost and the overhead at runtime needs to be considered thoroughly.

The basic idea of data migration is simple; two examples are shown in Figure 5.1. For an application running on two nodes (nodes 0 and 1), the data of this application is distributed on these nodes. Upon (predicted) failure of node 0, the data on node 0 is then transferred to node 1. The affected node 0 is eliminated from the concurrent execution. If any spare node (node 2) is available to the application, the data on node 0 can be moved to the spare node, where another instance of the application is started. This way, the failure of the application is avoided. Fault tolerance is achieved in systems both with and without a spare node by migrating the data of a failing node.

From this example, we can see that data migration is useful if the distributed parallel application is following the Single Program Multiple Data (SPMD) model, which is also the most common model for parallel programming. Nevertheless, the data migration technique can also be derived for other programming styles and distributed applications, if the application part running on the target node can provide the same functionality.

Load balancing resulting from data migration is also an important issue to be addressed. Data in parallel applications is usually carefully partitioned into equal subsets to be

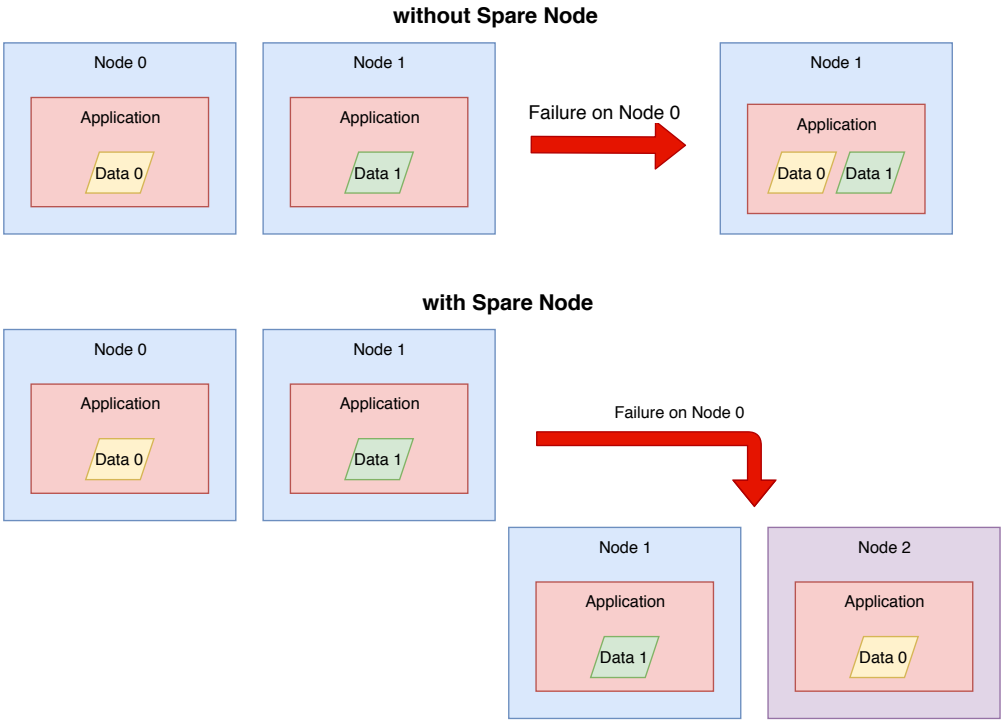

Figure 5.1.: Schematic Example of Data Migration

processed with similar time requirements. This way, waiting time can be kept at a minimum, and synchronous communication can be carried out with minimum time effort. However, the migration of application data without the spare node will break this balance. The application must react to this imbalance in order to restore efficient execution. To sufficiently support data migration, an application must be able to dynamically repartition after migration.

Another technical challenge is to expose the information on upcoming events to the application so that it can react to these events. As there are many programming models and libraries in parallel programming, the unification, and standardization of such interfaces for external information is a real challenge. One right way is to extend an existing programming model and library for this purpose.

In the remainder of this chapter, the concept of data migration and its requirements is introduced in detail. Later in this work, we provide two different library prototypes to support data migration for existing and new applications.

5.1. Basic Action Sequence for Data Migration

In the following, the steps for data migration **without** a spare node are explained in detail. An example of an application with three instances and a fault predictor without a spare node is presented in Figure 5.2.

For triggering data migration without a spare node, the running application must react to the command of a predicted upcoming event from a fault prediction system. Such a system can be an extension to a runtime system such as SLRUM, or a standalone application that has access to the monitoring infrastructure. The prediction result should be broadcast to all nodes so that this event is known by all participating instances of a running application. In the event of a fault, if the prediction system does not broadcast the information on fault, but only informs the failing node, the application instance on the failing node needs to relay this information and forward this to its other instances. Without information about any upcoming fault on all the application instances, the non-failing instances cannot initiate the action for data handover and application synchronization. It is also important to point out that the lead time of the prediction system must be long enough so that a live migration can be carried out by the application before its failure.

After all the application instances perceive a predicted fault, the failing instance must transfer its data to the other application instances. The data transfer can be done asynchronously. However, one must be sure that the data transfer process succeeds on all application instances. Otherwise, the application becomes corrupted, and data migration fails. In practice, it is essential to restore the balance of the distribution of

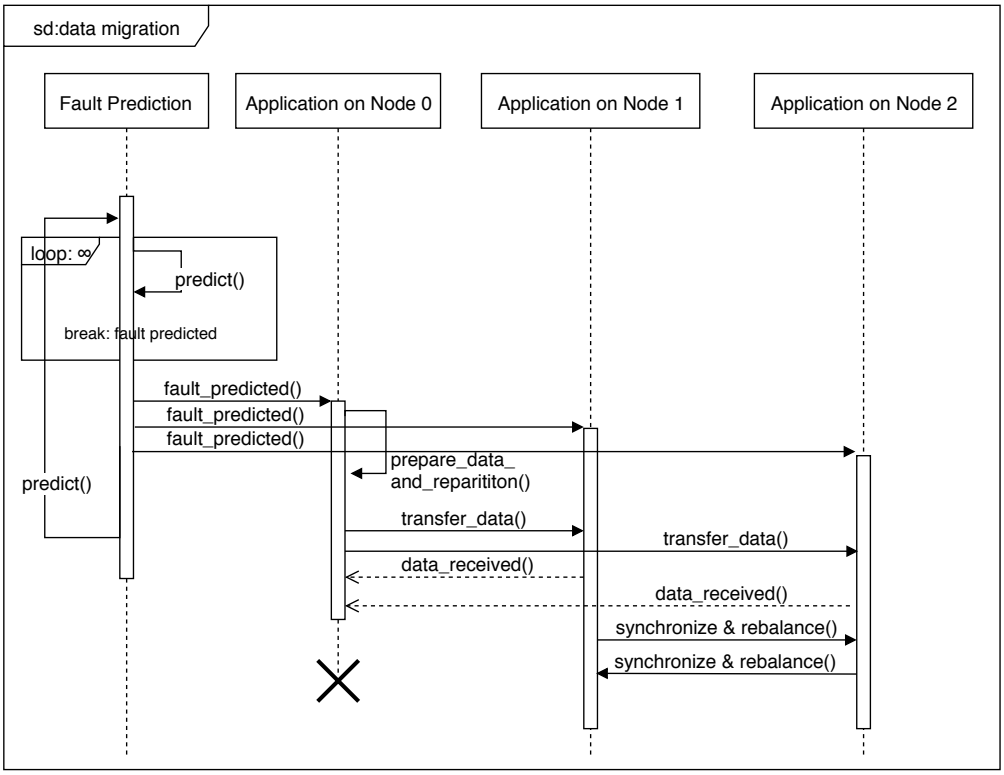

Figure 5.2.: Sequence Diagram of a Basic Data Migration

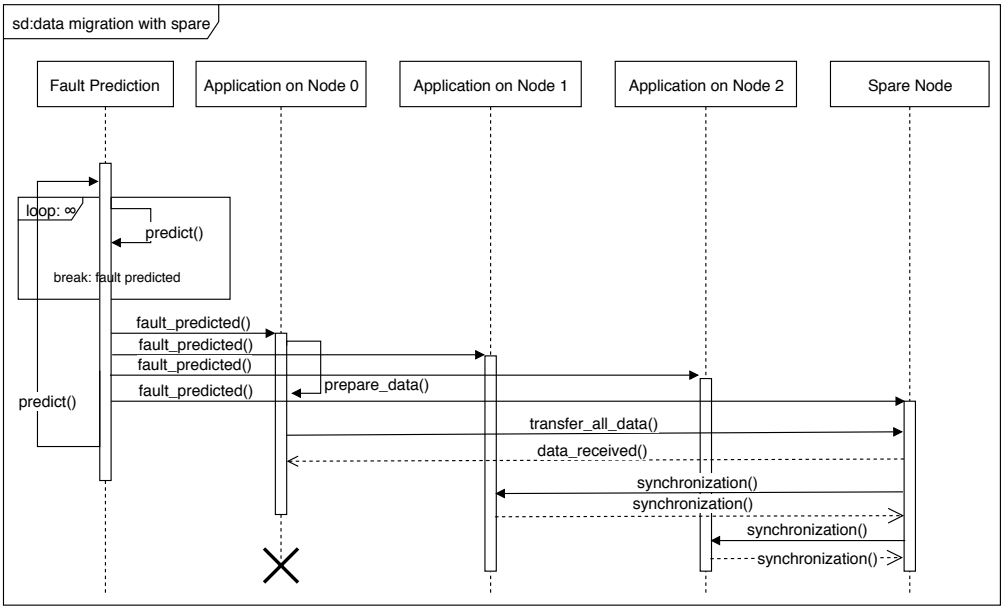

Figure 5.3.: Sequence Diagram of a Basic Data Migration with a Spare Node

data across the remaining nodes after the migration succeeds. The load balancing often limits the efficiency of a parallel application significantly. Therefore, the load balancing must be preserved or restored after failing nodes are removed from the application.

In the case where a spare node is available to the application, the action sequence is only slightly different from the previous case. Figure 5.3 shows the required steps for repartitioning with one spare node for an application with three instances and nodes. Upon prediction of an upcoming fault, an application instance must be started on the spare node. Instead of redistributing the data among the remaining nodes, data from the failing node can be transferred to the newly connected application instance. This way, the unaffected application instances can continue operation until the next global synchronization point is reached. Data migration with a spare node may also require a global redistribution of data for load balance purposes, if the number of failing nodes and the number of additional nodes are not the same.

Advantages and Disadvantages of Data Migration

As we can see from Figure 5.2 and Figure 5.3, the concept of data migration requires no system support apart from the fault predictor. The user application can fully achieve a data migration alone. This is the most significant advantage of data migration over other system based migration approaches. Moreover, as only application data is

communicated and not system-level state information, a lower overhead is expected.

The disadvantages of data migration also rely on the required application support. As the application is needed in order to react to future fault predictions, an existing application has to be adapted to be able to perform this task. However, this is an acceptable task because good parallel applications usually use modular data abstractions and communication routines. The required demand for adaptation is expected to be small.

Like any other migration-based methods, if there is no fault prediction available the effectiveness of data migration relies on the reconstructability of the data range on the failing node.

As mentioned above, load rebalancing is a crucial task enabling practical use of data migration. For clarity's sake, the key concepts in organizing the data of typical distributed parallel applications are introduced and explained in the following section.

5.2. Data Organization of Parallel Applications

In this section, we investigate the typical ways of organizing data in parallel applications. The scope of data migration in this work is limited to parallel applications; it is based on the Single Program Multiple Data (SPMD) model because it is a widely used programming model for parallel applications in a BJPCS, most prominently for HPC systems. To understand the different states of a data structure in an application, we briefly recap the basic concept by utilizing an example of a program as with four instances on two different nodes. Figure 5.4 highlights this example, in which two independent program instances run on each node. In this example, there are three different data structures: *DS0*, *DS1* and *DS2*. The distribution of data structures is called **partitioning** in this dissertation. Each part of the data (that is a subset of the total dataset) is called a **partition**. As shown in Figure 5.4, the location of data is different for all three data structures in this example application. A specific subroutine calculates the location of data in the parallel application. We call this subroutine a **partitioner** in this work. A partitioner can be *static*, where the partitioning itself cannot be changed after its creation. It can also be dynamic, where the partitioning can change as required. A typical example of changing the existing partitioning is *load balancing*. If the workload amount among the participating program instances is no longer balanced, the current partitioning has to be adapted to ensure load balancing. In this work, we call this process of adaptation or modification of an existing partitioning **repartitioning**. In addition to load balancing, a repartitioning can also be triggered by data migration, because existing data needs to be redistributed to other program instances. The relation between partition, partitioning, partitioner and repartitioning is shown in Figure 5.5.

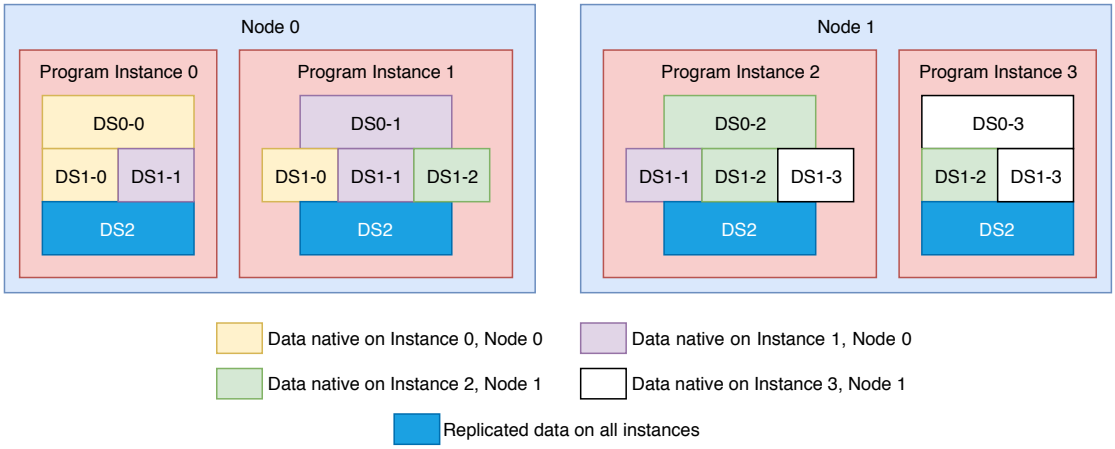

Figure 5.4.: Example for Data Distribution in a Parallel Distributed Application. DS stands for "Data Structure".

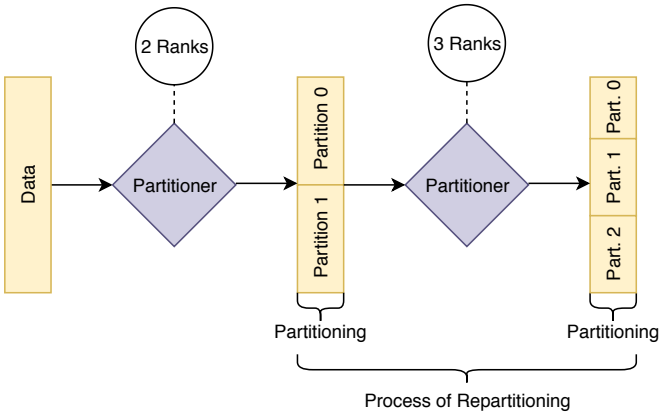

Figure 5.5.: Relation between Partitioner, Partitioning, Partition, and Repartitioning

Common Partitionings

There are three different basic partitionings in the example shown in Figure 5.4. The native location (ownership) of these data structures is visualized in different colors: yellow for program instance 0, purple for program instance 1, green for program instance 2, and white for program instance 3. The DS2 data structure (dark blue) is replicated and held by all program instances. The DS0 and DS1 data structures are distributed. DS1 is additionally replicated on various program instances locally. These three basic partitionings are widely used by parallel and distributed applications. We have summarized these partitionings as follows:

- **Exclusive Partitioning**: A data structure is in an exclusive partitioning if each program instance holds a mutually exclusive range of the data. The intersection between any partition is empty. This partitioning is usually used for kernels where the data on other program instances is not required. An example of this partitioning is found in a matrix-vector multiplication application solving $A \cdot x = y$, where A is the input matrix, and x is the input vector. If the partitioner is based on row-wise slicing of the matrix A, the resulting partitioning for A and the result vector y is in an exclusive partitioning. In Figure 5.4, DS0 is in an exclusive partitioning and divided into partitions DS0-[0...4].

- **Replicated Partitioning**: A data structure is in a replicated partitioning if each participating program instance holds a copy of the data. The data is *replicated* on all program instances. In the example of matrix-vector multiplication, the input vector x needs to be in replicated partitioning because it has to be accessed by all program instances. In Figure 5.4, the data structure DS2 is in replicated partitioning.

- **Shared Partitioning**: Shared partitioning is used for data structures, the partitions of which are required by various program instances. In parallel applications, since the calculation is executed independently on different program instances, a specific data range can be used by different program instances. A *reduction* is required to combine multiple partitions into one for data structures in shared partitioning. A simple example of shared partitioning is the *Halo* exchange pattern, in which access to the data from the immediate neighborhood instance is required. The borders of one's neighbor partitions must be known by each program instance so that the calculation can be performed. After each iteration, these borders are communicated and updated. In Figure 5.4, DS1 is in shared partitioning (DS1-[0...4]). Each program instance holds a copy of its neighbor partition.

A variety of further different partitionings can be derived from these three basic partitionings. For example, a *single* partitioning, where only a single program instance holds a specific data structure is a special case of the *exclusive* partitioning. The concept of partitioning and repartitioning is the most important foundation for data migration based fault tolerance, as the data migration is simply a (corner) use case of repartitioning.

Repartitioning Strategies

Two different repartitioning strategies can be used for restoring the balance and optimizing for the amount of data transfer:

- **Incremental Repartitioning**: The data from the failing node is equally redistributed among the remaining nodes. In the ideal case, only a minimum amount of data (that is the data on the failing node) is transferred. It results in minimum overhead at migration time. This scheme is illustrated in Figure 5.6.

  However, this approach can only be applied if the application kernels support the execution of non-consecutive data ranges. Furthermore, certain application classes will have higher communication overhead. For example, the *Jacobi* kernel [FH60] code with *Halo* exchange will certainly have higher communication demand after a data migration based on the incremental repartitioning scheme. The data on the first node is scattered across several nodes after migration.

- **Global Repartitioning**: All application data is redistributed across all non-failing nodes. A new partitioning is calculated globally at migration time, excluding the failing node. The application data is then transferred to a corresponding new

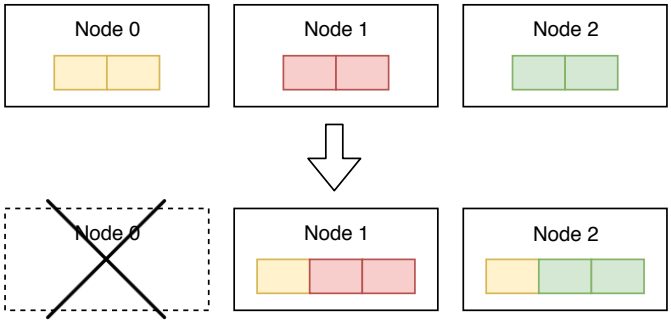

Figure 5.6.: A Incremental Repartitioning Scheme

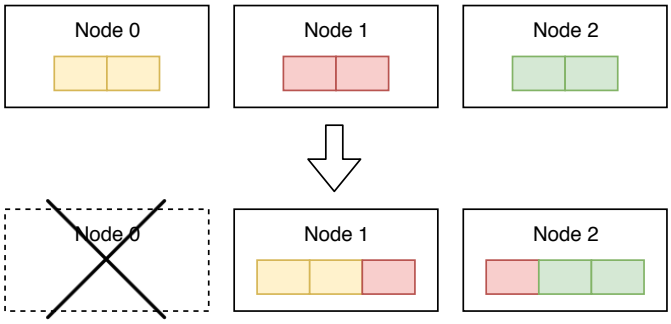

Figure 5.7.: A Global Repartitioning Scheme

destination. This scheme guarantees continuous data ranges on each remaining partition. Figure 5.7 visualizes this scheme.

The overhead at migration time is high: A large amount of data needs to be transferred across multiple nodes. The advantage of this approach is compatibility. When properly executed, the remaining, non-failing nodes can continue to operate as if the failing node had never participated in the parallel processing at any time. Furthermore, communication optimizations at runtime are easily achieved as the application can utilize its original code for communication without modification. The data migration also does not affect the performance of the previously-mentioned example of the *Jacobi* kernel either.

Selection of Repartitioning Strategy

The selection of a repartitioning strategy depends highly relies on the data partitioning, the application kernel support, the access pattern (read/write), the importance of data preservation, and the amount of data to be redistributed. It is the users' responsibility to select the most efficient and effective repartitioning strategy. Some examples are given in Table 5.1. However, some general guidelines apply: A replicated partitioning does not require repartitioning at all, because all the program instances replicate the data. A shared partitioning usually works better with global repartitioning to the reduce communication overhead at runtime. For exclusive partitioning, the repartitioning strategy depends on the ability of the application kernel to process non-continuous data ranges. If the data structure is not required to be persistent (i.e., a local working vector that is accessed and updated regularly per iteration), then repartitioning is not required.

Table 5.1.: Examples for Selecting Repartitioning Strategy

Partitioning	Access Pattern	Persistence	Kernel	Repartitioning Strategy
exclusive	any	yes	continuous data range	global repartitioning
exclusive	any	yes	any data ranges	incremental repartitioning
replicated	any	any	any	not required
shared	any	yes	any	global repartitioning

Repartitioning and Data Migration

Ultimately, repartitioning and data migration are similar processes. The target partitioning after a data migration can be calculated by re-executing the partitioner excluding the failing program instances. The only difference between repartitioning and data migration are the different abstraction levels. For a given data structure, there are two different abstraction levels: The *index space* and the *data* itself. In the simple example of an *array* with a length of n, identified by a *pointer* p, the data is the content stored in the memory at the pointer location. The index space is a collection of indices $[0...(n-1)]$ for the data structure. To access the data in a specific cell of the array, its index is used as *offset* to the pointer. This way, any data in a data structure can be addressed directly.

It is the responsibility of the partitioner in a parallel application to divide the *index space* into partitions, forming a partitioning. Following this partitioning, the actual data can be chunked into *data slices* and transferred to their designated location. Figure 5.8 illustrates the relation between partition, data slice and partitioning for the example of a 1-dimensional array.

At the abstraction level of *index space*, a repartitioning can be performed to free a failing program instance from any partition. Its corresponding *data slice* can then be transferred to its new location according to the new partitioning. While the reparti-

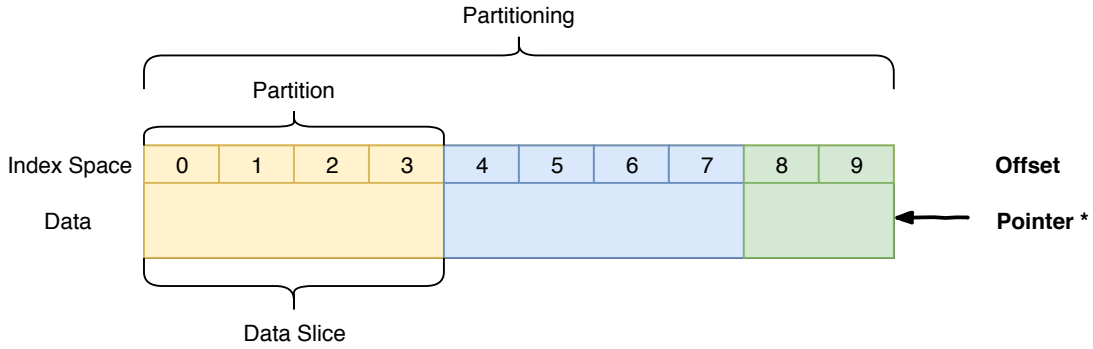

Figure 5.8.: Relation between Index Space and Data Slice

tioning process operates on the index space of a data structure, the process of data migration works on the actual data in the memory. Ultimately, a data migration process is based on the new partitioning producd from repartitioning.

In the remainder of this work, the terms *partition* and *partitioning* are exclusively used to refer to the index space of a data structure. The term *data slice* is used to refer to the actual data in memory. In order to perform data migration, data consistency must be preserved. In the next section, the data consistency model and process synchronization for data migration are discussed.

5.3. Data Consistency and Synchronization of Processes

In the previous section, we focused on the data distribution and organization for parallel applications. In this section, we focus on the data consistency within a (potentially heterogeneous) node and its effect on data migration.

Figure 5.9 shows an example setup with two heterogeneous nodes, each equipped with a single (multicore) CPU and two GPUs. These nodes feature a high-speed network interface (e.g., InfiniBand), which is connected to a high-speed interconnect. As an example of state-of-the-art technology, both the network card and the GPU are capable of performing Direct Memory Access (DMA). This allows peer-to-peer access of GPU memory with modern protocols, i.e., *GPUDirect* [RT+15]. Besides, the GPUs are connected via a high-speed near-range communication link, such as *NVLink* [FD17], further speeding up the peer-to-peer connection of GPUs within a node. All devices on the node are connected to the CPU using a Peripheral Component Interconnect Express (PCIe) [BAS04] bus.

Further, in this example, a parallel application, which supports the execution on both CPU and GPU (often known as a *hybrid application*), works with two data structures: *DS0* and *DS1*. The application data is partitioned in *exclusive* partitioning (cf. Section 5.2). It is started on the CPU and transfers part of its data to the GPUs for accelerated processing. It is written in line with the SPMD model. Moreover, it can benefit from all the hardware features and perform peer-to-peer data transfer without involving the CPU. This means that data on the GPU is only transferred back to the main memory if it is processed by the CPU. This approach is also the standard approach for *hybrid applications* nowadays. The effects of cache are not discussed, as its consistency with main memory is ensured by the hardware. For data migration, data stored in a persistent storage – the globally distributed file system – does not require migration.

The activity diagram of this example hybrid application is illustrated in Figure 5.10. In this application, one GPU-only synchronization is performed for data communi-

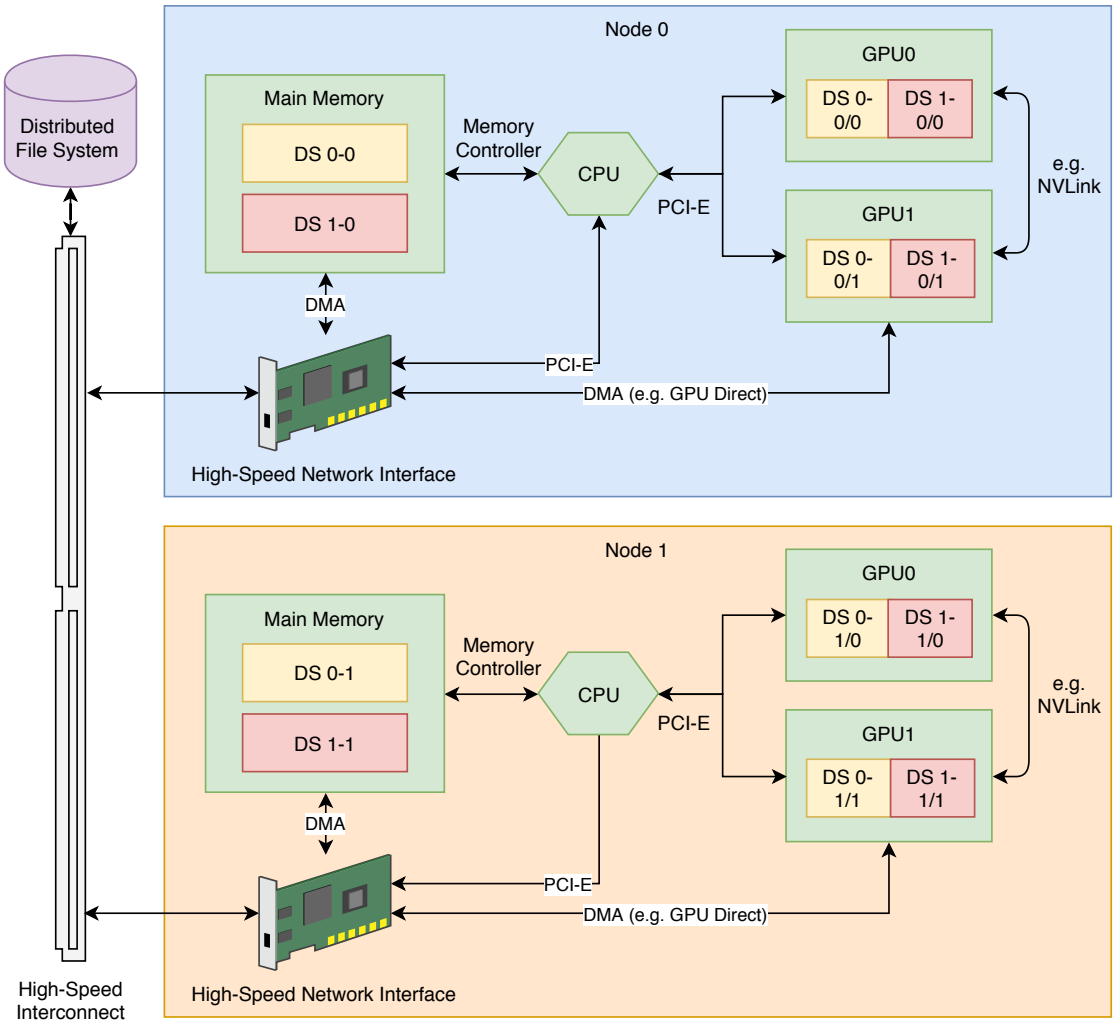

Figure 5.9.: Scheme of Data Distribution in a Dual-node System

cation. Furthermore, part of the data is calculated on the CPU in parallel. In the activity diagram in Figure 5.10, the yellow and blue colors indicate activities performed in parallel on *Node 0* and *Node 1*. The grey area indicates a non-parallel region that requires synchronization between *Nodes 0* and *1*. The red boxed region stands for activities on the GPUs without any CPU involved.

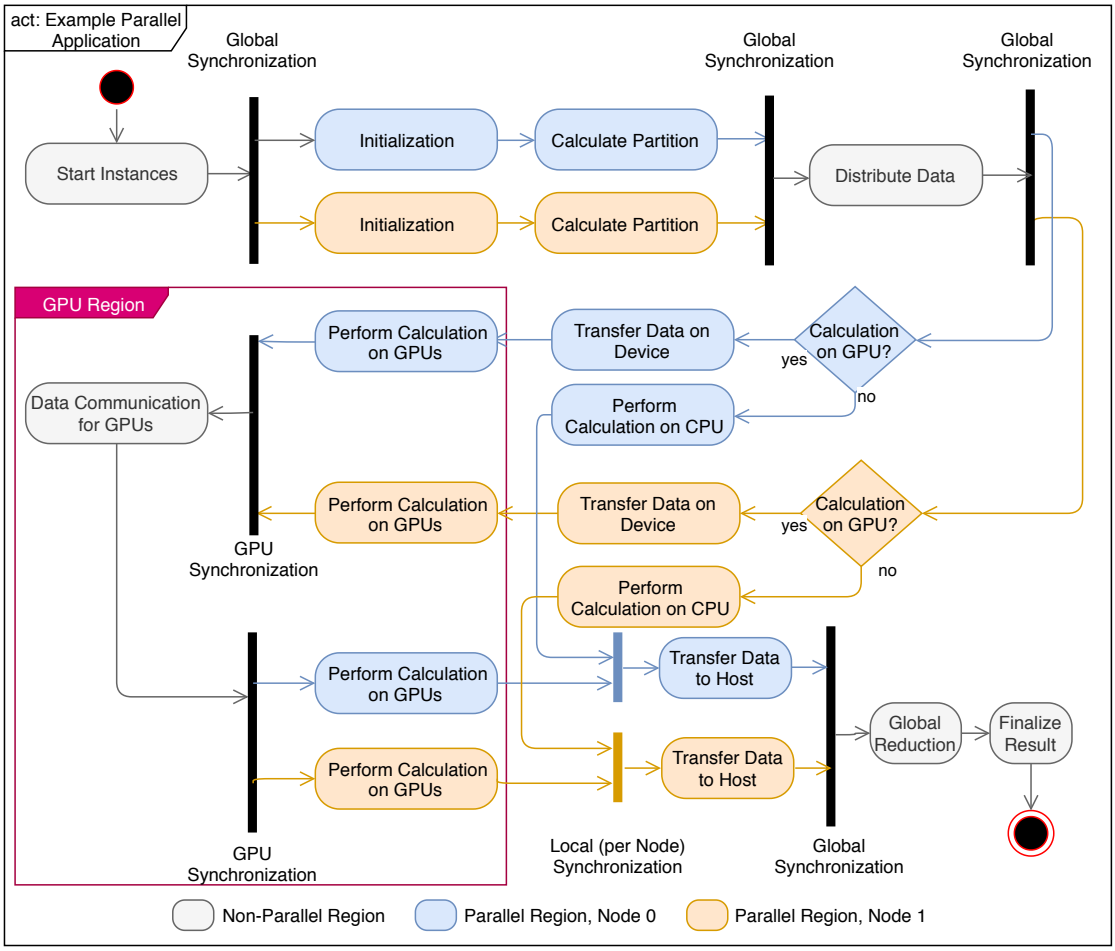

Figure 5.10.: Activity Diagram of an Example Hybrid Application

From this example, we can see that the data is not consistent in main memory after the data transfer to the device (GPU) memory. It remains inconsistent until the last global synchronization occurs. However, in order to perform a data migration (e.g., from *Node 0* to *Node 1*), a global data consistent state in the main memory of all nodes must be reached. Recapping Section 5.2, in order to perform data migration,

repartitioning has to be done in order to obtain a new valid partitioning that excludes all failing application instances. The new partitioning often requires moving large data blocks globally. If the data is inconsistent (i.e., if different application instances are in different iterations), it cannot be moved safely. Furthermore, as movement of data is often initiated by the CPU, data on the device must be copied back to the main memory. Although peer-to-peer data migration in a device is possible, the implementation of such migration kernels is very complicated and inefficient, as a global redistribution of data may change the workload assignment between the CPU and GPU. Therefore, data consistency is a requirement prior to any data migration.

In our application, a migration can only be performed at the very beginning or at the very end. This is not the desired use of data migration. Fortunately, most real-world applications require global synchronization more often. The above-mentioned condition regarding data consistency for data migration exists in more phases throughout the application.

In a standard Batch Job Processing Clustering System (BJPCS), the operating system does not have sufficient information regarding the location and distribution of the application data. Consequently, an automatic transfer from device memory to host memory is not possible without the support of the application. Furthermore, most GPU kernels cannot be interrupted. For this reason, data migration can only be achieved with proper support from the application. The application must be modified so that it can to react to the outside information regarding upcoming failures. Moreover, it has to perform the required data migration at a suitable time point.

5.4. Summary: The Concept of Data Migration

In this chapter, we have introduced the basic concept of *data migration*, which is a fail-operational fault tolerance strategy based on proactively migrating the data from a failing node to another location. *Data migration* is also an application-integrated fault tolerance strategy because it requires the application to react to system-level information regarding an upcoming event. It can be performed with or without a spare node.

The main advantages of *data migration* over other migration methods is the reduced overhead. It does not request a full migration of all process or system-level information. Since the application controls the migration process, the overhead of data migration can be further reduced by selectively migrating only the critical data. As current failure prediction systems can cover a large variety of upcoming hardware failures and also have a high or undeterminable false alarm rate, the overhead of migration significantly

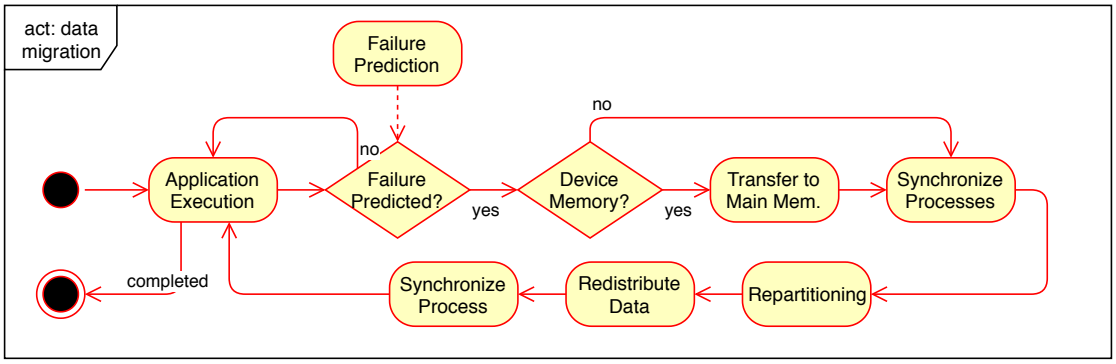

Figure 5.11.: Activity Diagram of the Data Migration Process

impacts the usefulness of migration-based fault tolerance methods. With *data migration*, the lower overhead can help reduce the overhead created by a false positive prediction. Redundant data migration can be reverted with less impact on the program execution. Moreover, a data migration can be achieved by repartitioning the data structures and redistributing the data. This means that it can be seen as a corner case of dynamic load balancing operation. The latter is supported by many existing parallel applications. Finally, data migration is a user-level only action, which does not rely on hardware, the OS or system-level libraries.

However, some drawbacks of *data migration* apply. Due to the nature of being application-integrated, an existing application has to be adapted to support data migration. Data migration can only be performed in a state in which data consistency is ensured by the application. This means that additional synchronization might be required prior to any data migration. Finally, like any other migration based fault tolerance strategy, data migration alone cannot handle any random error.

Although the scope of our concept currently includes SPMD-based parallel applications, the concept of data migration can also be used in other kinds of distributed and parallel applications. For example, a master-slave application can achieve data migration by selecting a new master or redistributing the workload of the slaves.

The conceptual process of *data migration* is modeled in Figure 5.11. The required steps and conditions are summarized as follows:

1. Assume a (parallel) application is running in a normal state. It periodically checks on external information as to whether a failure has been predicted.

2. On a predicted upcoming failure, the application chooses the next possible timepoint to react to the prediction.

3. If kernel execution is performed on an accelerator (such as a GPU), the data is copied back to the main memory.
4. All the process instances are synchronized. The application ensures a consistent data state.
5. A new partitioning of all the data is calculated by repartitioning that excludes the failing nodes.
6. All the data is redistributed according to the new partitioning.
7. The process is synchronized again to ensure data consistency after repartitioning.
8. The data migration has now been done. The application continues normal execution.

In this chapter, we have identified several important design quality factors and also its limitations. These are:

- Data migration can be seen as a corner case for any load balancing operation.
- The impact of data migration on performance depends heavily on load balancing after migration.
- Data migration can only be performed in a consistent data state.
- Data migration cannot be performed easily on device memory on a peer-to-peer basis.

In summary, data migration is a versatile tool for achieving proactive fault tolerance on an application-integrated basis. In order to demonstrate the usefulness of data migration, a user-level library, called LAIK, has been developed to assist parallel-application programmers in supporting data migration more easily. We are going to introduce and evaluate the LAIK library in the next chapter.

6. LAIK: An Application-integrated Index-space Based Abstraction Library

In the last chapter, we discussed the concept of data migration as a fault-tolerant strategy, along with its advantages and disadvantages. The requirements and constraints of data migration were also introduced and explained. Furthermore, we identified that data migration can be treated as a corner case of dynamic load balancing by excluding the failing nodes from the parallel application.

In this chapter, based on the previous knowledge, we focus on a possible library prototype for assisting data migration in parallel applications called *Lightweight Application-Integrated Fault-Tolerant Data Container (LAIK)*. LAIK stands for "Leichtgewichtige AnwendungsIntegrierte DatenhaltungsKomponente" (translation: Lightweight Application-integrated Data Container). The basic idea of LAIK is to provide a lightweight library that helps HPC programmers to achieve (proactive) fault tolerance based on the concept of data migration. The library includes a set of Application Programming Interface (API)s to assist programmers to dynamically partition and repartition data. By handing over the responsibility for data partitioning, LAIK can calculate and recalculate the best partitioning option according to the current workload and hardware situation. If a prediction regarding an upcoming failure is made, LAIK can take that information and repartition. The user can then adapt the data distribution and free the failing node from any application data.

The concepts and structure of LAIK are discussed in detail in the following. Furthermore, the performance evaluation of LAIK confirms the usefulness of LAIK for fault tolerance. In the end, the performance impact is given in order to show the low overhead introduced by LAIK.

6.1. The LAIK Library

As introduced in Section 2.3.1, Single Program Multiple Data (SPMD) is a widely used programming model to exploit data parallelism in HPC systems. The basic idea of SPMD is to apply the same kernels (calculations) to a portion of data multiple times in parallel on different Processing Unit (PU)s. For an SPMD-based application, the index space of the user data is usually *partitioned* into a specific, user-defined *partitioning*.

The data slices are distributed across different program instances according to this partitioning. Each instance of the parallel application computes on its part of data – this rule is commonly known as "owner computes". The user-defined partitionings should support the efficient execution of the parallel application. It is the programmer's responsibility to provide a proper implementation of a partitioner, by utilizing his and her knowledge in the communication pattern, the compute kernel, and the importance of load balancing.

The LAIK library (in the following: LAIK) makes use of this model and assists applications written in the SPMD model with data migration by taking over control of partitioning from the user application. This way, LAIK can expose system-level information regarding a predicted upcoming fault to the user application by adapting the partitioning and excluding failing nodes. Accordingly, an application can react proactively on the prediction by adopting the new partitioning.

Furthermore, LAIK can take over the responsibility of repartitioning and redistributing the data if the programmer transfers the responsibility for application data to LAIK. The programmer specifies a partitioner to tell LAIK *how* data is distributed across different program instances. This way, the programmer only needs to specify *when* data migration is allowed. The actual process of repartitioning and data migration is automatically completed by LAIK. For different purposes and different layers of abstraction (index space or data space), LAIK provides different APIs. The layered design of LAIK will be introduced and presented later in this chapter.

By transferring the responsibility of data distribution to LAIK, implicit communication can be achieved. The programmer only needs to specify *which* part of the data is required and *when*. With LAIK, this is achieved by providing multiple partitionings that are bound to the same data structure. By changing the currently active partitioning, which is a process called "switch", data can be redistributed across available program instances, resulting in implicit communication.

Another advantage of automatic data distribution and redistribution is that load balancing can be ensured throughout program execution. With HPC systems emerging in the future, load balancing becomes a crucial task to ensure scalability. For traditional HPC applications, adaptive load balancing is usually achieved explicitly. This results in high application-specific engineering and implementation, leading to high code complexity. This type of high code complexity eventually leads to maintainability issues. With LAIK, this responsibility can also be fully transferred to LAIK to ensure high modularity and low complexity of user code.

This section of our work focuses on the design and interfaces of LAIK in detail. Besides the basic feature of providing fault tolerance based on data migration, other highlights of LAIK include the following:

- LAIK is modularized. Every component layer and its corresponding APIs can be changed, without affecting other LAIK components.
- LAIK is incremental. Different layers of APIs can be used at different levels of abstraction, resulting in adapting applications incrementally. Different features can be achieved with different layers of APIs. This means that an application programmer can transform his and her existing application on a step-by-step basis.
- LAIK is lightweight. Unlike related programming models such as Charm++ [KK93] and Legion [Bau+12], and task based programming model such as OmpSs [Dur+11] and StarPU [Aug+11], LAIK neither requires a specialized compiler nor a runtime system, thus resulting in low performance overhead per design and low system complexity.

LAIK is provided as open source software and can be obtained from *Github*[1].

6.1.1. Basic Concept of LAIK

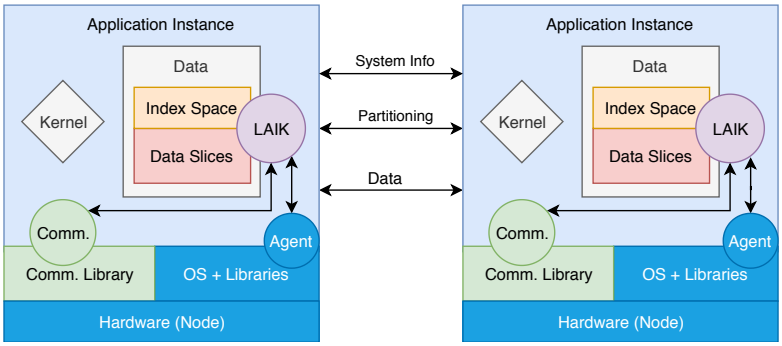

Figure 6.1.: Basic Components of the LAIK Library

Figure 6.1 schematically illustrates the role of LAIK and its interaction with the user application in a schematic view. Technically, LAIK is a C library which runs completely in user space within the application process. Each application (process) instance contains also a LAIK instance, as shown in Figure 6.2.

[1] `https://github.com/envelope-project/laik`, accessed in July 2019

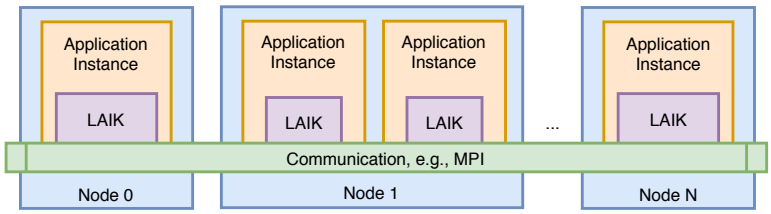

Figure 6.2.: Location of LAIK Library on a Multinode Setup

LAIK provides different API layers for controlling either the process groups only , or the index space (partitioning), or both the index space and the actual data slices (cf. Section 5.2). The LAIK library is designed to use existing communication backends, such as the Message Passing Interface (MPI) or the Transport Control Protocol (TCP). It is neither designed to become a runtime system, nor a communication library. It is designed to be a dedicated library which helps programmers control the partitioning of data at different timepoints during program execution. This enables LAIK to be lightweight with little to no performance overhead. In a basic setup, LAIK neither changes an application's kernel execution, nor its communication. However, by transferring the responsibility of data slices to LAIK, communication can then be fully abstracted in LAIK. This way, implicit communication is executed instead of explicit programming required by communication routines. An advantage of implicit communication with LAIK is its ease of transition to other communication backends.

Information from the system environment, such as a predicted upcoming error is inserted into LAIK by utilizing a so-called "agent", which is a type of adapter that intercepts information from the operating system or from the runtime system. This way, fault tolerance through data migration can be achieved.

In order to support different communication backends as well as different agents, APIs for *backend* and *external* information are created. Furthermore, two layers for user level APIs, the *index space* APIs and *data container* APIs are included to allow operations at different abstraction levels. The detailed architecture of LAIK and its APIs are explained in the following subsection.

6.1.2. Architecture of LAIK

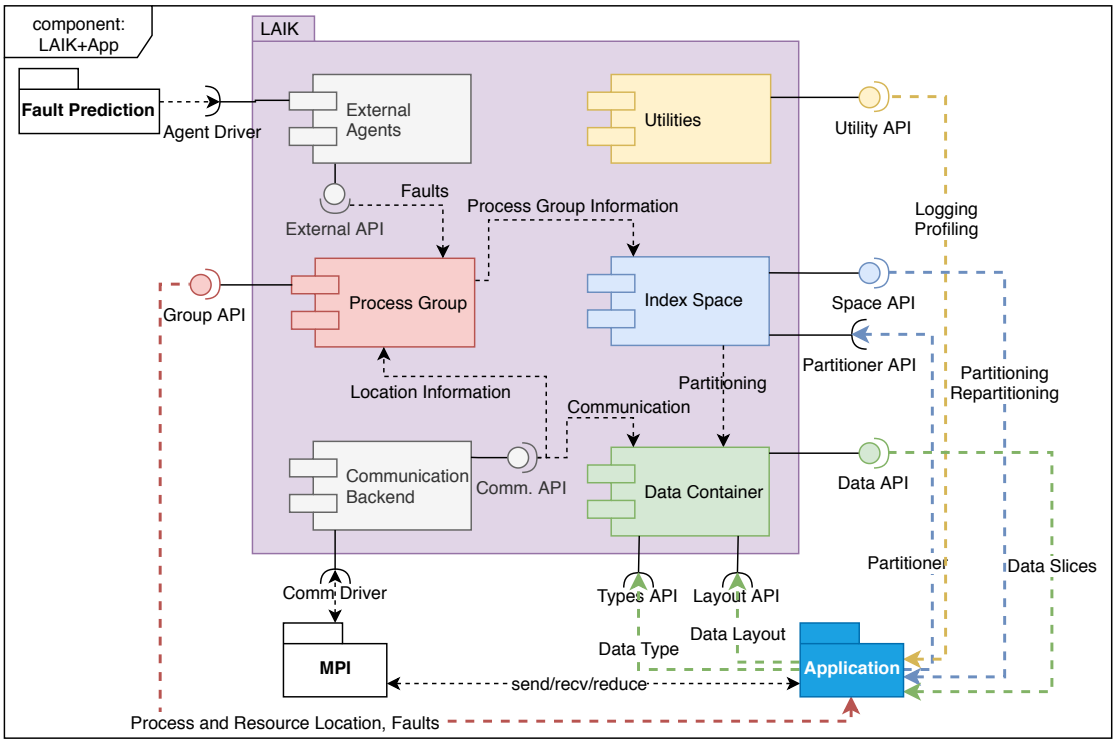

Figure 6.3.: Component Diagram of LAIK

As already mentioned, LAIK is designed to be modular with different components providing different sets of functionalities. There are six major components in LAIK. Their relations and interfaces are presented in Figure 6.3 above.

An overview of the components showcased in Figure 6.3 is listed in Table 6.1 on the following page. Detailed descriptions of the functionalities of these components are given below.

- Process Group Component (Figure 6.3 light red): The process group component of LAIK takes care of the size and assignment of process numbers, as well as their mappings to the PUs (usually processor cores). LAIK may also make use of the other components, such as the communication backend and external agents to optimally calculate group information. This information is stored in a `Laik_Group` object and provided to the application and other LAIK components using the *Group API*.

Table 6.1.: Overview of LAIK Components and Their APIs

Component	API	Usage
Process Group	Group API	Creating and managing process groups in LAIK.
Index Space	Space API	Create and manage index space partitioning in LAIK.
	Partitioner API	Provide callback for partitioners to LAIK.
Data Container	Data API	Creating and managing data structure (and storage) in LAIK.
	Layout API	Providing callback function for packing and unpacking data to LAIK.
	Types API	Creating and managing user-defined data types in LAIK.
Communication Backend	Communication Driver	Providing communication interface for LAIK, such as MPI.
External Agents	Agent Driver	Providing runtime and system information provider (agent) to LAIK.
Utilities	Utility API	Providing debugging and profiling functionalities.

In our current prototype, LAIK groups are *static*, which means a `Laik_Group` object is immutable. Any changes to a group (such as information regarding a failing node) create a new group object. Programmers are advised to explicitly free group handles that are no longer in use.

- Index Space Component (Figure 6.3 light blue): The index space component of LAIK is responsible for handling partitioning and repartitioning over abstract index space. As explained in Section 5.2, the index space is used to identify a specific data offset within a data structure. In SPMD-based applications, the entire index space is usually split into partitions according to a given partitioning scheme. In LAIK, the calculation of partitionings is done by the index space component, which can be based on either a generic or a user-provided partitioner. The index space component provides the *Space API* to the user application so that it can operate on partitioning and its related information.

- Data Container Component (Figure 6.3 light green): The data container is a component of LAIK which operates on the data slices of the application data structures by providing memory management functionalities. It features the *Data API* which is ultimately based on an allocator interface that utilizes the partitionings calculated from the index space module. By using the data API, application programmers can get managed memory space for storing the data slices. Furthermore, if the application data is managed by LAIK, communication is triggered implicitly when switching a data container from one partitioning to another, if the application data is managed by LAIK.

  The data container component includes two user-provided callback interfaces: *Layout API* and *Types API*. The layout API is used to gather user provided dense representation of given data structures, thus increasing communication efficiency by packing data slices into this dense format. The types API is used to register user-specified data types in addition to the primitive data types.

- Communication Backend (Figure 6.3 grey, bottom): The communication backend component is responsible for handling the communications resulting from any LAIK operations, such as switching between different partitionings for a data container. Furthermore, it provides LAIK with environmental information from the communication library, such as the location of process instances. The communication component is an internal-only component; therefore its API - the *Communication API* cannot be accessed by user application. A communication backend includes a communication driver for a specific communication technology, such as the MPI, the Transport Control Protocol (TCP), or Shared Memory (cf. Figure 6.1 and Figure 6.2).

- External Agents (Figure 6.3 grey, top): The external agents component is responsible for handling information that comes from "external" locations, such as the runtime system, the Operating System (OS) or an external operator. For fault tolerance purposes, a fault predictor can be attached to LAIK as an external agent. External agents are loaded as dynamic libraries at runtime.

  Each LAIK instance can load multiple external agents. There is no predefined rule on what an external agent can do. Consequently, it is the user's responsibility to provide the implementation and desired functionality.

- Utilities (Figure 6.3 light yellow): The utilities component provides a *Utility API* with basic functionalities such as comprehensive logging, profiling, and recording of program execution information (e.g., current iteration and phase). It also feeds the process group API with crucial information regarding fault prediction. Furthermore, the utilities API provides most basic functionalities for the LAIK, such as initialization and deinitialization.

Among all these components, the utility, process group, index space, and data container are used to provide functionalities to the application programmer. The external agents and communication backend components are used by LAIK to access system resource and runtime information. The utility and process group components together provide the most basic functionalities of LAIK. For this reason, we also call them the *LAIK Core* component together.

6.1.3. Overview of LAIK APIs

In the last section, we discussed the major components of LAIK. With these components, LAIK provides three different types of APIs (see also Table 6.2):

1. User APIs: The user APIs are used to provide functionalities from LAIK that are required application programmers. There are four sets of user APIs: *Utility*, *Process Group*, *Index Space*, and *Data Container*. Except for the Utility API, which is used to provide the most basic functionalities, the other three APIs are designed to assist data migration at different abstraction levels.

2. Callback APIs: The callback APIs are used by LAIK to gather information and functionality from the programmer or from a third-party library. Callback APIs in LAIK include *Agent Driver*, *Communication Driver*, *Partitioner*, *Data Types*, and *Data Layout*.

3. Internal APIs: The internal APIs are currently designed to be used by neither the application programmers, nor by any third-party library. They are designed to

support LAIK's modularity and the ease of interchanging components in future development.

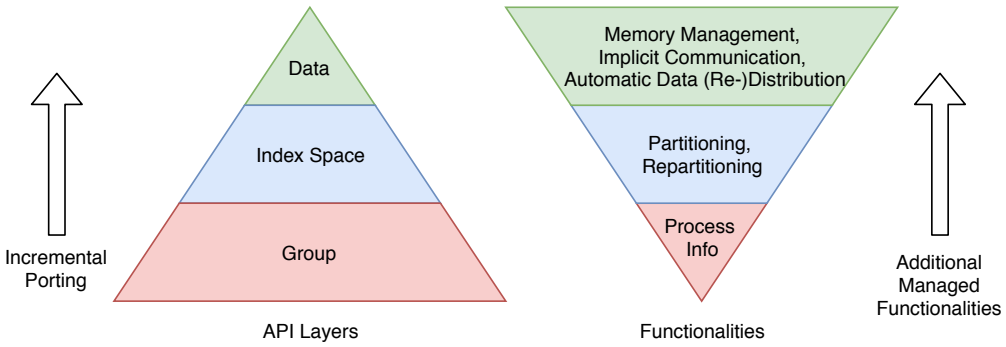

Figure 6.4.: API Layers of LAIK for Incremental Porting

With regard to the user APIs, three sets of user APIs can be used by the application: Group, Space, and Data. This design is intended to support incremental porting of existing applications, which can make use of different API layers, in order to gain increasing functionality. This way, the user can adapt different functionalities in a step-by-step manner, reducing the overhead and turn-around time for porting existing applications. The functionalities of each layer of API are listed in Table 6.2.

Table 6.2.: Overview of LAIK API Layers

Abbreviation	API Layer	Operand	Functionality
Group	Process Group	Process Instances	Provides basic information on current process instances. Information about the process location and its identification, the size (total number of participating processes) of the parallel application, and upcoming failures is provided by this interface.
Space	Index Space	Index Space of Data Structures	Provides operations and information on index space (partitioning). Calculate a valid partitioning and provide repartitioning functionality according to a change in a process group, such as in failure cases. Calculate the difference of two partitionings (a so-called Action Sequence) to advise the user to communicate efficiently.
Data	Data Container	Data Slices (Memory) of Data Structures	Provides operations and information on data slices (memory) with an allocator interface to allocate data storage space in memory for a given partitioning. Provides automatic data and communication management functionalities. Provides an allocator interface, perform the action sequences (communication) on the switch between different partitionings for a given data structure.

In the following, we will introduce these different layers of APIs by looking at the example of porting a matrix-vector multiplication application to LAIK using the different API layers. For this example, let A be the matrix, which is stored in a 2D array, and let x be the vector, which is stored in a 1D array. The calculation for the matrix-vector multiplication is shown in Equation 6.1. The primitive implementation of an MPI-based matrix-vector multiplication is given in Algorithm 1.

$$A \cdot \overrightarrow{x} = \overrightarrow{y} \tag{6.1}$$

Algorithm 1: MPI-based Matrix Vector Multiplication

```
A : The Input Matrix
x : The Input Vector
y : Result of Matrix-Vector-Multiplication

MPI_init();
(myStart, myEnd) ← partitionMatrix(A, MPI_size(COMM_WORLD),
 MPI_rank(COMM_WORLD));

for i ← myStart to myEnd do
  | y[i] ← calculateDotProduct(A[i], x);
end

MPI_Allreduce(y);
if MPI_rank(COMM_WORLD) == 0 then
  | returnResult(y);
end
MPI_Finalize();
```

6.1.4. User API: The Process Group API Layer

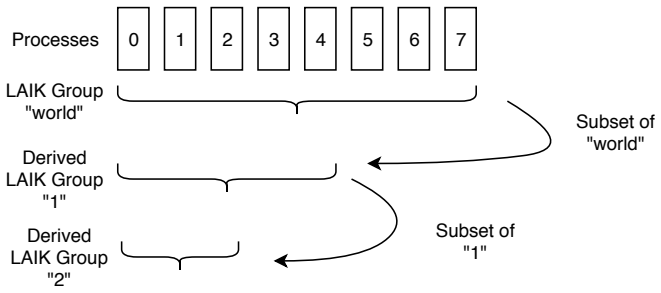

Figure 6.5.: Schematic of Different LAIK Groups

Similar to MPI Groups [WD96], LAIK process groups (object type `LAIK_Group`, also called LAIK group) are collections of process instances. Figure 6.6 illustrates the potential distribution of process groups of an example parallel application. The LAIK group *world* is a default process group that contains all the process instances of a LAIK application. Further LAIK groups can be derived from existing LAIK groups by subsetting them. Different LAIK groups are showcased in Figure 6.5.

In the prototype we have implemented for LAIK, the LAIK process groups are static,

which means they cannot be changed after creation. The advantage of static process groups lies in their simple semantics. Furthermore, this concept is very similar to the concept of *MPI groups*, allowing an easier transition of MPI applications to LAIK. However, static groups come with the downside of having many group handles, which may become orphaned and cause high memory overhead.

For fault tolerance, LAIK provides the function `Laik_get_failed()` to enable the application to get a list of failing process instances within the *world* group. This information can be used by the application to create a subgroup of the initial *world* group, thus excluding the failing instances in order to achieve fault tolerance.

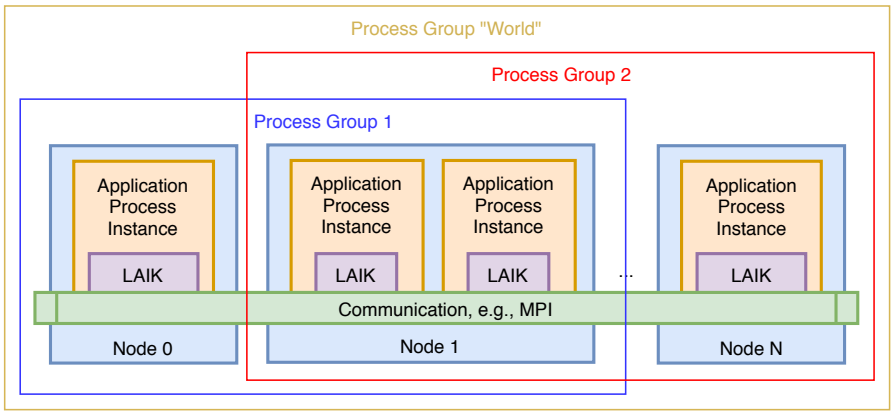

Figure 6.6.: Schematic Overview of LAIK Process Group Concept (cf. Figure 6.2)

For the example of a matrix-vector multiplication application only using LAIK's Group API Layer, the application can be handed to LAIK as presented in Algorithm 2. The information regarding the process instances is transited to LAIK's responsibility instead of to MPI's. With LAIK's MPI backend, LAIK inherits backend size and rank information to reduce porting effort, which then further reduces the effort for porting existing MPI application to LAIK.

As a result, almost no changes need to be done in the application code. Note that the partitioning and communication are still the responsibility of the user code with the Group API.

Algorithm 2: LAIK-based Matrix Vector Multiplication with Group API Layer

```
A : The Input Matrix
x : The Input Vector
y : Result of Matrix-Vector-Multiplication
LAIK_init();
(myStart, myEnd) ← partitionMatrix(A, LAIK_size(LAIK_WORLD),
 LAIK_id(LAIK_WORLD));
for i ← myStart to myEnd do
    y[i] ← calculateDotProduct(A[i], x);
end
MPI_Allreduce(y);
//The communication still stays in MPI.
//finalization
if LAIK_rank(LAIK_WORLD) == 0 then
    returnResult(y);
end
LAIK_Finalize();
```

6.1.5. User API: The Index Space API Layer

With the next API layer, the Index Space API, an application specifies a partitioner it intends to use and does not need to care about the partitioning anymore. For each data structure, multiple partitionings may be required in different phases for the same data structure. An example is shown in Figure 6.7.

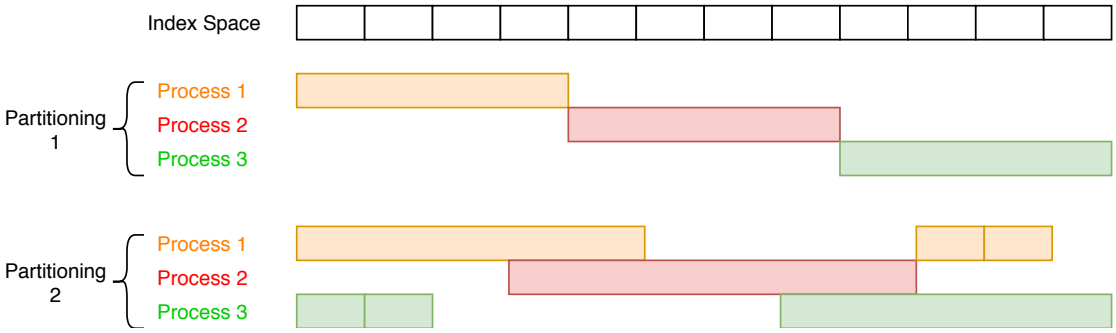

Figure 6.7.: Schematic Overview of Two Example Partitionings for a Single Index Space

LAIK's partitioning interface is bound to the previously introduced group interface introduced. It uses the process group information as an input to calculate the best

partitioning. The subroutine that performs the actual partitioning calculation is called a *partitioner*. Partitioners for LAIK are either directly provided by LAIK or specified by the application programmer by utilizing the *Partitioner Callback API*.

Out of the box, LAIK provides default partitioners for some regular data structures, such as 1D/2D/3D arrays. LAIK provides a set of default partitioning algorithms, which are illustrated in Figure 6.8:

- Master: Only one process holds the data structure completely.
- Block: The data is distributed equally among all the instances.
- Halo: The data is distributed equally among all the instances, with each instance holding a read-only part of its neighbors.

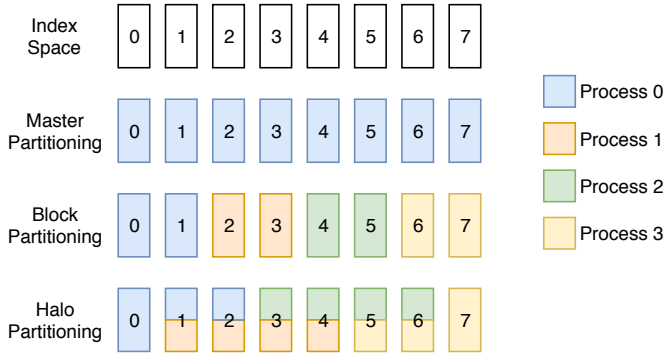

Figure 6.8.: Overview of Different Built-In Partitioners of LAIK

If different partitionings are applied to the same data structure, the application can specify *which* data is needed and *when*. The change from one partitioning to another is called a *transition* in LAIK. When there is a transition, a communication sequence that is required to switch between one partitioning to another can be calculated. This sequence of communication is called *Action Sequence* in the LAIK terminology. Potential actions can be send/receive, broadcast, or reduction. An example of a transition and its corresponding action sequence is given in Figure 6.9. A calculated action sequence can be accessed by the application and executed by either the application or (with the Data APIs) by LAIK.

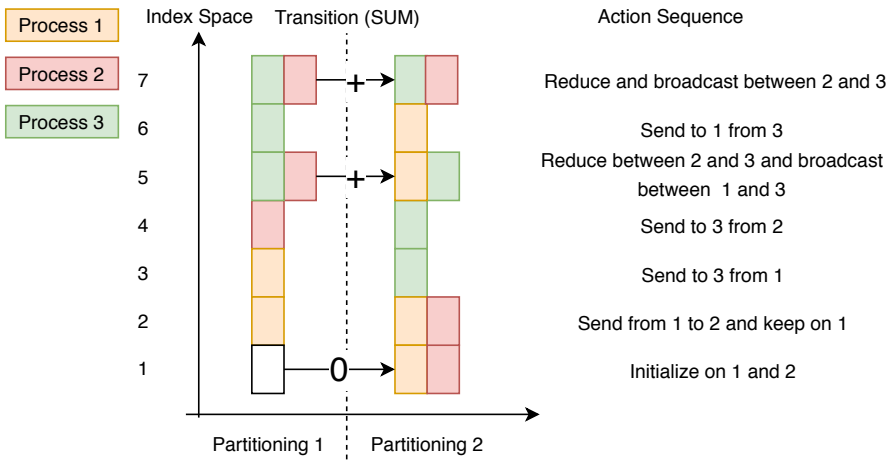

Figure 6.9.: Schematic Overview of a Transition

For our Matrix-Vector multiplication example, two partitionings are created:

- *Sub-Partitioning*(`pSub`) for the matrix and the resultant vector, which is a distributed partitioning where each process instance holds a portion of the data (default block partitioning), and
- *Total Partitioning*(`pTotal`), where the data is only held by one process instance (default master partitioning).

The computation kernel is adapted so that each instance asks LAIK for its partition. LAIK does not necessarily ensure the continuity of partitions for any given process instance. As a result, an additional `while`-loop is added to enable the application to work on multiple non-continuous subpartitions. Finally, the application can obtain a list of communication operations that need to be done. Our algorithm is transformed

as shown in Algorithm 3 on the next page.

Algorithm 3: LAIK-based Matrix Vector Multiplication with Space API Layer

```
A : The Input Matrix
x : The Input Vector
y : Result of Matrix-Vector-Multiplication
LAIK_init();
//subMatrix and wholeMatrix are partitioners
//Create partitioning for data
pSub ← LAIK_new_partitioning(A, subMatrix);
pTotal ← LAIK_new_partitioning(A, wholeMatrix);
(myStart, myEnd) ← LAIK_my_partition(pSub);
while (myStart, myEnd)! = NULL do
    for i ← myStart to myEnd do
        y[i] ← calculateDotProduct(A[i], x);
    end
    (myStart, myEnd) ← LAIK_my_next_partition();
end
transition ← LAIK_calc_transition(pSub, pTotal);
foreach t in transition do
    executeTransition(t);
    //This is still responsibility of the programmer!
end
if LAIK_rank(LAIK_WORLD) == 0 then
    returnResult(y);
end
LAIK_Finalize();
```

6.1.6. User API: The Data Container API Layer

The *LAIK Data API* layer is the most advanced API layer in LAIK, operating on the application data itself and providing memory and communication management for data structures. It utilizes both the information from process groups, as well as the partitionings calculated from the space API (cf. Section 6.1.5). The user specifies the data types and layouts of its data structures, and LAIK performs memory allocation and deallocation for these data structures. LAIK provides primitive data types – such as integers and floating point numbers – and default data layouts – such as 1D/2D/3D arrays. Customized data type and layout can be added by utilizing the *Data Type* and the *Data Layout* callback APIs (cf. Section 6.1.7).

To access a data structure, the application calls the `Laik_map()` function to get the underlying pointer and range for its data slice. Each time after a transition between different partitionings is executed, the data pointer may become obsolete and will then need to be updated by calling the `Laik_map()` function again.

To reduce the overhead from frequently mapping and invalidating the pointers to the data structure, LAIK provides a feature called *pointer reservation*. By requesting pointer reservation, the application can enforce LAIK to keep a pointer persistent. The pointer to the data slice remains valid until a change is allowed at a future timepoint. A schematic view of this concept is given below in Figure 6.10.

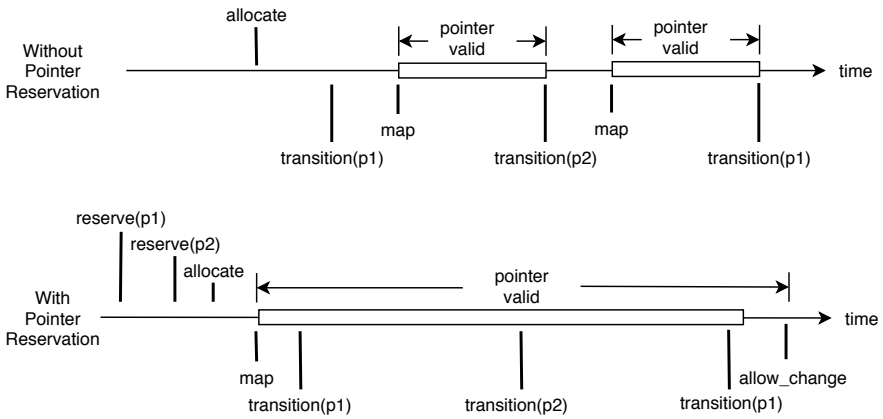

Figure 6.10.: Schematic Overview of Pointer Reservation

However, by utilizing the pointer reservation, higher memory overhead can be produced. Figure 6.11 showcases an example. To ensure pointer validity, the data structure is not kept dense on the memory. Instead, sufficient memory space is allocated for storing data that is used in all the required partitionings.

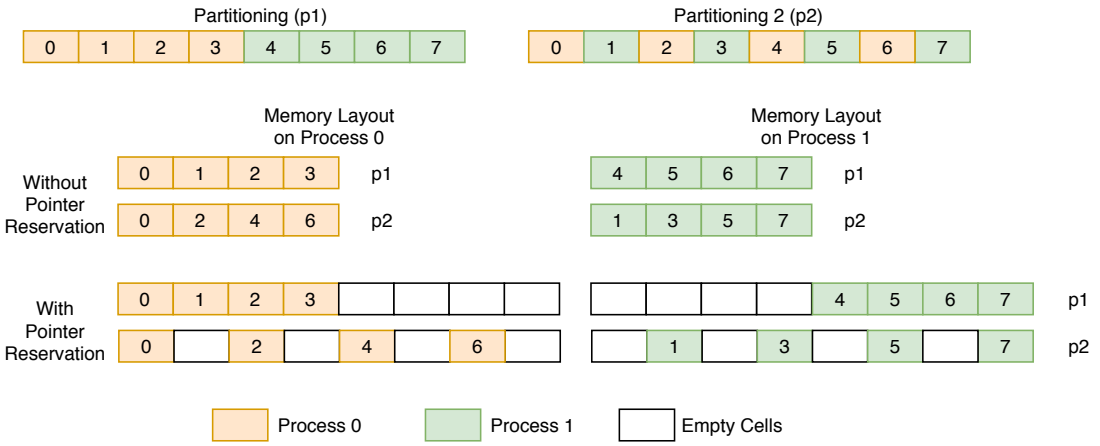

Figure 6.11.: Example Memory Layout with and without Pointer Reservation

In addition to the location of data, LAIK also keeps track of the access pattern to data structures, such as read-only, read/write. Furthermore, LAIK stores information regarding the usefulness of the data at a given timepoint (these timepoints are called *access phases*), such as preserve data, initialize data, or useless data. This way, LAIK can optimally calculate and minimize communication overhead. For example, data that is not required at in a subsequent access phase does not need to be transferred, thus reducing communication demand. An example of the relation between transition and different access patterns is given in Figure 6.12.

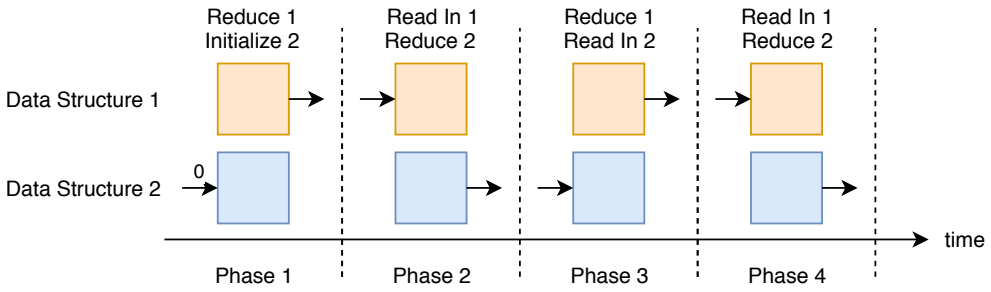

Figure 6.12.: Schematic Overview of Access Pattern and Transition

The action sequences calculated in the *Space API* can be executed in the data API by calling the `Laik_exec_transision()` function, triggering communication and data exchange. This way, LAIK performs implicit communication for the application. With LAIK's Data API, the communication can be purely achieved by describing the required location of data (partitioning) and the transition between different partitionings.

To further reduce the overhead caused by the calculation of action sequences, programmers can ask LAIK for a pre-calculated action sequence between different parititonings. Combined with the pointer reservation functionality, these preserved action sequences also remain valid until the memory layout is changed. This way, the application can keep executing the same action sequence for the same transition between two particular partitionings, unless a change in a given partitioning occurs.

For fault tolerance, LAIK's Data API uses `Laik_repartition_and_repart()`, which obtains node failures from the external interface (cf. Section 6.1.8. This function also calculates a new derived partitioning by using the derived, shrunk group excluding the failing nodes. Finally, it triggers all the necessary communication operations for data migration by executing a transition from the original to the derived partitioning.

In our Matrix-Vector-Multiplication example, with the Data API Layer, all the communication and memory management is done by LAIK. The resulting program is given in Algorithm 4. The resultant vector y becomes a LAIK data container, as well as the matrix A, because they are distributed across all nodes. Initially, both y and A are assigned with `pSub` partitioning for parallel computation. After the kernel execution, the resultant vector is switched to `pTotal` partitioning, which gathers the data to instance with the ID 0. This transition between the partitioning `pSub` and `pTotal` also triggers *gather* on instance ID 0 automatically.

In summary, the three different levels of user APIs allow the programmer to port an existing application in SPMD model to LAIK in an incremental, step-by-step manner. Data migration is possible at all API layers (cf. Table 6.2): With *Group API*, the user is informed about current process instance and group information. With *Space API*, data partitioning and repartitioning is calculated by LAIK. The user only needs to carry out the specific data operations corresponding to the change in data partitioning. With *Data API*, all the operation required for data migration is covered by LAIK. The programmer only needs to tell LAIK about *when* an operation is allowed.

Algorithm 4: LAIK-based Matrix Vector Multiplication with Data API Layer

A : The Input Matrix
x : The Input Vector
y : Result of Matrix-Vector-Multiplication

```
LAIK_init();
//subMatrix and wholeMatrix are partitioners
//Create partitioning for data
pSub ← LAIK_new_partitioning(1D, subMatrix);
pTotal ← LAIK_new_partitioning(1D, wholeMatrix);

//create data containers
matrix ← LAIK_new_data( 1d, A, pSub); y ← LAIK_new_data (1d, y, pSub);
(myStart, myEnd) ← LAIK_my_partition(pSub);
while (myStart, myEnd)! = NULL do
    LAIK_get_data_pointers(matrix, y);
    for i ← myStart to myEnd do
        result[i] ← calculateDotProduct(matrix[i], x);
    end
    (myStart, myEnd) ← LAIK_my_next_partition();
end
LAIK_switch_to_partitioning(result, pTotal);
if LAIK_rank(LAIK_WORLD) == 0 then
    returnResult(y);
end
LAIK_Finalize();
```

6.1.7. Callback APIs

The callback APIs are interfaces that enable the user to configure LAIK in order to support customized partitionings, data types, and data layouts. There are three callback APIs:

- *The partitioner API* is used to allow the user to create a LAIK-compatible partitioner for data partitioning. This way, the index space component can handle arbitrary data structures that come from the user application. An example of a partitioner will be given later in this chapter.
- *The data types API* is used to enable the user to configure customized data types and their operations in the data container component. This is crucial for LAIK to be able to carry out automatic communication on arbitrary data structures. To register a data type, simply specify its name and size, as well as two functions: One for setting the values of a referenced array of elements to the neutral element of the reduction; and another for doing element-wise custom reduction on two arrays of values.
- *The data layout API* is used to allow LAIK to efficiently manage memory as well as communication. As user-defined data structures are usually driven by the logic of user application, these data structures can have complex data layouts. In order to handle such layouts, the programmer has to supply LAIK with this layout information. The layout callback API is utilized by specifying a *pack* and an *unpack* function to allow LAIK to serialize and deserialize a data structure.

A partitioner for LAIK is structured as following: It should provide LAIK with the information about which index range belongs to which partitioning. Figure 6.13 illustrates an example of partitioning to showcase the partitioner API. In this example, a 2D index space is equally distributed row-wise in two tasks. Such partitioning is useful for applications such as matrix multiplication. For each row, 4 elements are added to partition 1 and partition 2, respectively. The example partitioner for LAIK is given in Algorithm 5. Our example partitioner loops over the rows in this matrix, and adds 4 elements (called *slices*) to the given partition. This way, the index space is partitioned into two partitions.

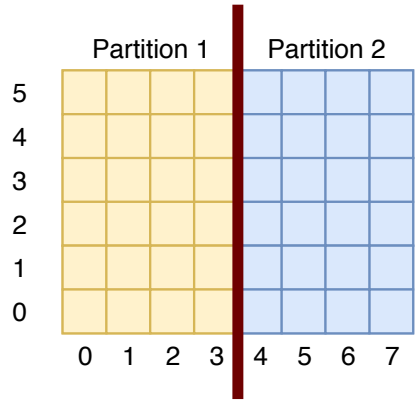

Figure 6.13.: Example Partitioning for LAIK Partitioner API Demonstration

Algorithm 5: Example Partitioner of the Example Partitioning in Figure 6.13

```
t0, t1 : LAIK Tasks
p      : LAIK Partitioning
for i ← 0 to 5 do
    LAIK_append_slice(p, t0, i, 4);
    LAIK_append_slice(p, t1, i+4, 4);
end
return p;
```

6.1.8. The External Interface

LAIK's external interface is designed to communicate with external programs or routines – called *Agents*. The agents are provided by the user, according to his and her needs. Each agent is a *dynamic library*, which is loaded demand at initialization time. The agent is executed synchronously within LAIK instances' process. Therefore, agent programmers have to implement their own asynchronous communication if required. Each LAIK agent must provide the implementation of `agent_init()` function, stating its name, version, and capabilities. This function is also the main entry point for each LAIK agent.

LAIK supports the loading of multiple agents at the same time. However, loading a large amount of agents may cause significant memory and performance overhead because they run within the same scope of the LAIK instance. Currently, LAIK only supports agents for fault tolerance purposes which feature an `agent_get_failed()` function, reporting the identifiers of the upcoming failing nodes. The LAIK agent component provides a synchronous call `Laik_get_failed()`, which can be used by the

application of LAIK internals to query information on either upcoming faults or on current faults from all loaded agents. This way, LAIK exposes system and runtime level information to the application.

The schematics for the LAIK external interfaces are given in Figure 6.14 below.

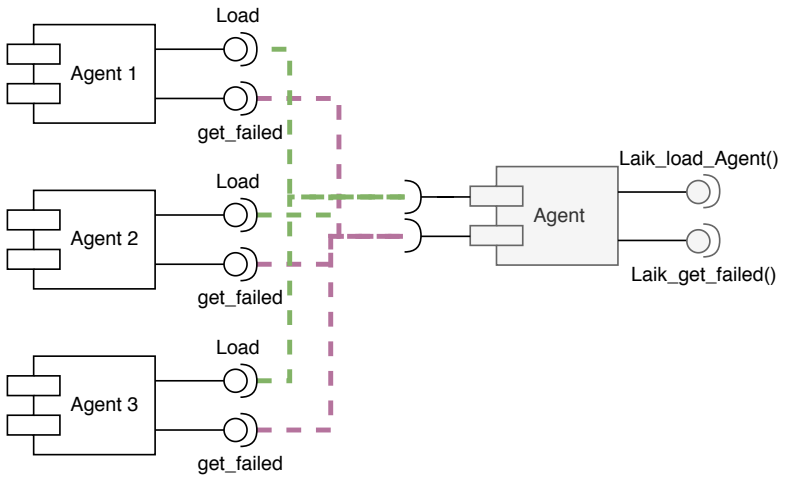

Figure 6.14.: Schematics for the LAIK External Interface

6.1.9. The Communication Backend Driver Interface

LAIK supports arbitrary communication backends to help its operation. Out of the box, LAIK currently provides two different backends:

- MPI backend, and
- TCP backend.

However, application programmers can provide additional LAIK backends by implementing a set of callback functions. The most important callback functions are:

- `init()`: This is the main entry point for initializing the backend. It is called before any application code is called and create a LAIK instance with the given backend. It also allows the backend driver to initialize the communication library.
- `finalize()`: This function is used to advise the backend to clean up its resource and the communication library. It is also used to destroy the corresponding LAIK instance.

- `prepare()`: The `prepare()` function is used by the backend driver to analyze the action sequence and translate the LAIK action sequence to the communication sequence actual required. Furthermore, the backend driver can use this function to allocate the resources for communication operations. As the same transition is expected to occur frequently, the backend driver can preallocate these resources (such as `send` and `recv` buffers) and preserve them to improve the performance.
- `cleanup()`: The `cleanup()` function enables the backend to clean up resources allocated by the `prepare()` function, such as such as `send` and `recv` buffers.
- `exec()`: This function triggers the actual communication in the backend for a given action sequence.
- `updateGroup()`: As LAIK is designed to be fault-tolerant, the underlying backend may need to react on changes to the process group (such as the removal of a process instance in a group). The adaptation of the group-specific data in the backend is achieved by calling this function.

The application programmer is required to call the initialization function of his and her desired backend directly to obtain a LAIK instance with that given backend (e.g., `LAIK_Init_mpi`). LAIK can run with multiple backends at the same time. However, each LAIK backend will create an independent LAIK instance, albeit these instances are running in the same application. Any operation on a specific LAIK instance has no effect on other LAIK instances.

6.1.10. Utilities

LAIK's utility interface is mainly responsible for providing basic profiling operations, debugging printouts, and exposing the program information to LAIK. The profiling functions provide basic functionalities such as starting/stopping a timer, printing out timing information and writing such information to file. The program control functions enable any program to set identifiers for an application's current status information, such as iteration (`LAIK_set_iteration`) and program phase information (`LAIK_set_phase`). Finally, the debugging functions provide the capability of printing information to standard output tagged with detailed information about LAIK groups and sizes at different debugging levels.

6.1.11. Limitations and Assumptions in Our Prototype Implementation

Our current LAIK implementation is provided as open source software on *Github*[2]. As LAIK is still a prototype, some limitations apply:

- At the time of this writing, our LAIK implementation comes with two communication backends: *TCP backend* and *MPI backend*.
- The MPI backend is used for evaluating the performance of LAIK, as well as its overhead on runtime and memory. However, as MPI does not provide fault tolerance capabilities such as shutting down some of the existing ranks gracefully, an empty LAIK process without active data to calculate cannot be shut down. In our evaluations, such processes call `laik_finalize` and wait until the other processes also finish.
- The TCP backend is a lightweight implementation that provides a minimum set of communication functions (send/receive) to allow LAIK to operate. This backend is not used for performance benchmarking. However, it overcomes the drawback of not allowing empty processes to shutdown and can be used to demonstrate the fault-tolerant capability of LAIK.
- Currently, LAIK only supports the "shrinking" of process groups because the external interface only allows the transmission regarding failure information (`get_failed`). This means that data migration with spare nodes is currently not supported.

6.2. Basic Example of a LAIK Program

Before we get started with our evaluation of LAIK, let us understand LAIK with the help of a simple example application called *vsum*. *vsum* is an application which calculates the sum of a given vector A which is stored as a simple C array. For the sake of simplicity, we assume that the length of the vector is divisible by the number of processes. This simple application is illustrated as pseudocode in Algorithm 6 on the next page.

To implement this algorithm in LAIK, we select the MPI backend of LAIK. This way, the application can be started through `mpirun` and controlled as an MPI application. Figure 6.15 on the following page illustrates the necessary steps for the *vsum* example. After the initialization of LAIK, a `Laik_Space` object `s` is created for the index space of

[2] `https://github.com/envelope-project/laik/commit/a2641901de9a0fe3c3f236be165cd5a94430b70a`, accessed in December 2018

Algorithm 6: Example Application of *vsum* using MPI

```
A : The Input Vector
y : Result of Vector Sum

MPI_Init();
n ← A.length() / num_processes;
myStart ← n*my_rank;
myEnd ← myStart+n;

for i ← myStart to myEnd do
    y += A[i];
end

MPI_Allreduce(SUM, y, WORLD);
MPI_Finalize();
```

vector *A*. A `Laik_Partitioning` object `p` is created using the built-in *block* partitioner, where each process instance has an equal range of the index space assigned. Afterward, a *Laik_Data* object `a` is created for A with the previously created `Laik_Space` object `s`. The data is switched to the partitioning `p`, triggering redistribution of data, if required.

Before the calculation can be started, each process instance must call `laik_my_slice` to identify its range for processing. Finally, the calculation is executed. After the calculation has been done, we create another partitioning with the built-in *all* partitioner, where each process instance holds the entire vector. By switching the currently active partitioning to the new partitioning, LAIK triggers the specified *all* reduction automatically, after which every process instance holds the result of the vector sum.

In the end, `laik_finalize` is called, indicating the end of the program.

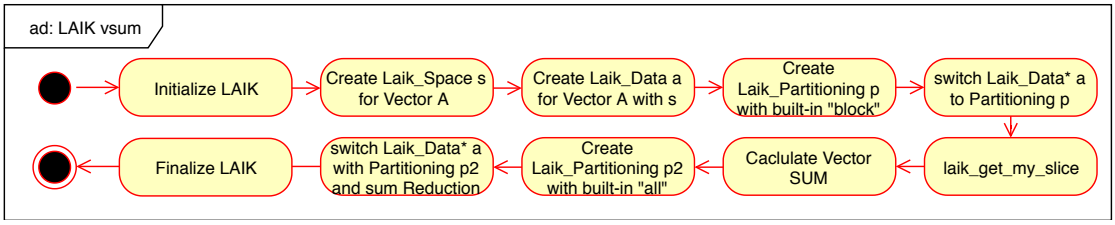

Figure 6.15.: Activity Diagram of LAIK-based *vsum*

6.2.1. Extended Example of Automatic Data Migration with LAIK

Given the previous example of *vsum*, we extend this application with data migration functionality. The adapted activity diagram is shown in Figure 6.16.

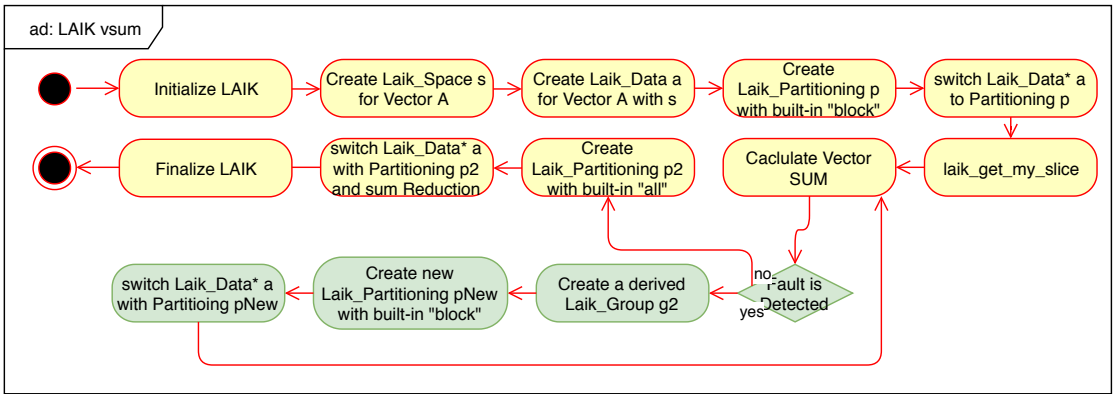

Figure 6.16.: Activity Diagram of LAIK-based *vsum* with Data Migration

Upon a predicted fault, the application can ask LAIK to create a derived new `Laik_Group` object `g2`, thus excluding all failing process instances. Based on this new group, a new `Laik_Partitioning` object `pNew` is created with the built-in partitioner *block*. Finally, by executing a transition to the new partitioning, data transfer (and migration) is executed automatically. This way, LAIK ensures that no workload remains on the failing process instances.

6.3. Evaluation of the LAIK Library with Real-world Applications

6.3.1. Application Example 1: Image Reconstruction with the Maximum-Likelihood Expectation-Maximization (MLEM) Algorithm

In order to evaluate the performance impact and the effectiveness of data migration, two applications have been ported to LAIK. The first application we evaluated is the Maximum-Likelihood Expectation-Maximization (MLEM) algorithm for Positron Emission Tomography (PET) image reconstruction [SV82]. A PET is a medical imaging technique to observe the metabolism process in order to detect, stage and monitor a range of diseases. For this purpose, a radioactive tracer has to be injected into the observational object. Such tracers emit positrons during their β-decay. Two 511keV gamma photons, which travel in opposite directions, are created through annihilation

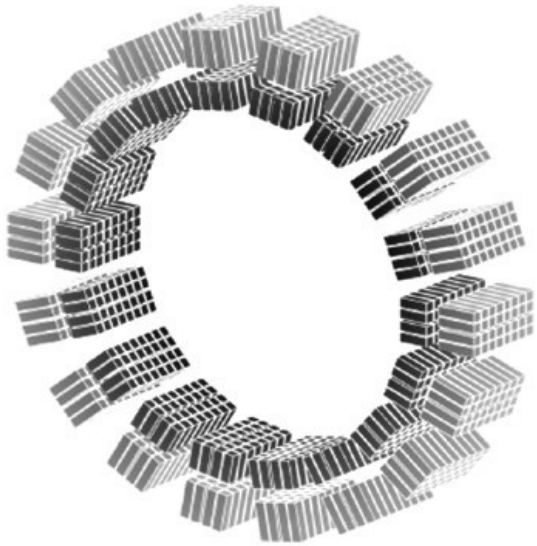

Figure 6.17.: Schematic View of the *MADPET-II* PET Scanner [Küs+09]

of the positrons with electrons [Küs+09]. To detect these photons, a PET scanner equipped with a ring of detectors based on scintillator crystals and photodiodes is used. If two detectors record a photon within a certain time window, an event is assumed somewhere along the line connecting the detectors. Such a line is called a Line of Response (LOR). In 3D space, the two detectors can detect events not only from a line, but rather a larger polyhedral 3D space inside the scanner, called the Field of View (FOV). The number of detected events influences the quality of measurements, and the coverage of the FOV by LORs affects the resolution. The FOV is generally divided into a 3D grid, where each grid cell is called a voxel [Küs+09].

In this dissertation, the example scanner used is called *MADPET-II*, which was developed at *Klinikum rechts der Isar der TU München*. It is designed for high-resolution 3D imaging of small animals. *MADPET-II* features a unique design with two concentric rings of detectors. This way, the sensitivity increases without loss of resolution. A schematic view of the *MADPET-II* scanner is given in Figure 6.17.

The output from the detector - called list-mode *sinogram* - must be reconstructed using the system matrix. The latter describes the geometrical and physical properties of the scanner.

The Maximum Likelihood Expectation Maximization (MLEM) Algorithm

The algorithm used to reconstruct the image for our *MADPET-II* scanner is MLEM, developed originally by Shepp and Vardi [SV82] in 1982.

$$f_j^{(q+1)} = \frac{f_j^q}{\sum_{l=1}^{N} a_{lj}} \sum_{i=1}^{N} a_{ij} \frac{g_i}{\sum_{k=1}^{M} a_{ik} f_k^q} \tag{6.2}$$

The algorithm uses the iteration scheme shown in Equation 6.2 above, where

- N is the number of voxels,
- M is the number of detector pairs (LORs),
- f is the 3D image that is reconstructed,
- A is the system matrix of size $M \times N$, which describes the geometrical and physical properties of the scanner,
- g is the measured list-mode sinogram of size M, and
- q is the iteration number.

The algorithm is based on the probability matrix $A = a_{ij}$, where each element represents the probability of a gamma photon discharge from a voxel j being recorded by a given pair of detectors i [Küs+09].

In particular, this algorithm can be divided into four major steps:

1. **Forward Projection**: $h = Af$. In this step, the current approximation of the image is projected into the detector space.
2. **Correlation**: $c_i = \frac{g_i}{h_i}$. The projection from step 1 is correlated with the actual measurement.
3. **Backward projection**: $u = A^T c$. The correlation factor is projected back into image space by multiplying with the transposed system matrix.
4. **Update image**: $f_j^{q+1} = \frac{u_j}{n_j} f_j^q$. An update for the image with the back-projectiong correlation factor is calculated and a normalization n is applied.

The algorithm assumes an initial estimated gray image, which is calculated by summing up the elements of g, which is the measured data, and then dividing this by the sum of the elements of A, which is the geometry matrix. This process is given in Equation 6.3 [Küs+09] on the following pages.

$$f^0 = \frac{\sum_{i=1}^{N} g_i}{\sum_{j=1}^{M} n_j}$$
$$where, n_j = \sum_{l=1}^{N} a_{jl} \tag{6.3}$$

The runtime of the algorithm is dominated by the *forward* and *backward projection*, both being sparse matrix-vector operation. This algorithm is known to be memory (bandwidth) bound [Küs+09].

The System Matrix of MADPET-II

The system matrix A of MADPET-II contains its geometrical and physical description. The FOV is divided into a grid of $140 \times 140 \times 140$ voxels in (x, y, z) direction. There is a total of 1152 detectors in MADPET-II, which result in 664128 unique LORs. The content of the system matrix is generated by the detector response function model as described in [Küs+10; Str+03]. The matrix is stored in Compressed Sparse Row (CSR) [Saa03] format to save space. The elements are stored in single precision floating point number format [Küs+10].

A more detailed description of this system matrix and a density plot for visualization are given in Appendix B of this dissertation. Although the transposed system matrix is required in the backward project, we do not store the transposed matrix. This is because the backward projection can be written as $u^T = c^T A$, which only requires the original CSR matrix [Küs+09].

6.3.2. MPI Parallelization

For parallelization, the matrix is partitioned into blocks of rows with approximately the same number of non-zero elements per block. This results in good, albeit not perfect load balancing. The reference implementation of MLEM is written using MPI. It features a checkpointing possibility after each iteration and a command-line parameter controls this option. The implementation of the MPI-based MLEM is shown in Algorithm 7. The implementation of the reference MLEM is available as open source software on *Github*[3].

[3] `https://github.com/envelope-project/mlem`, accessed in December 2018

Algorithm 7: MPI Based MLEM Implementation

```
matrix : Input: The System Matrix
lmsino: Input: The Listmode Sinogram
nIter  : Input: The Number of Total Iterations
chkpt  : Input: Flag for Checkpointing
image  : Output: The Reconstructed Image

(myStart, myEnd) ← splitMatrix (rank, size);
//myStart = starting row, myEnd = ending row
calcColumnSums(ranges, matrix, norm);
if needToRestore then
    restore(checkpoint, image, iter);
else
    initImage(norm, image, lmsino);
end
for i ← 1 to nIter do
    calcFwProj(myStart, myEnd, matrix, image, fwproj);
    MPI_Allreduce(fwproj);
    calcCorrel(fwproj, lmsino, correlation);
    calcBkProj(myStart, myEnd, matrix, correlation, update);
    MPI_Allreduce(update); calcUpdate(update, norm, image);
    if chkpt then
        checkpoint(image, i);
    end
end

if rank==0 then
    writeImage(image);
end
```

Porting MLEM to LAIK

In order to port MLEM to LAIK, the code is transformed in several steps [Yan+18]:

1. Matrix Loading Routine: The sparse matrix class file reading the data from the `csr` file[4] is modified to support the reading of non-contiguous data ranges. This is required in order to support multiple non-contiguous partitions (and data slices), which is created with an *incremental partitioner* (cf. Section 5.2).

2. Data Partitioning: Create the index space object (`Laik_Space`) over the rows of the sparse matrix, as well as a corresponding partitioning object (`Laik_Partitioning`). The built-in *block* partitioner with a weighted index is chosen to support the unbalanced rows of the sparse matrix. Furthermore, a shared partitioning (cf. Section 5.2) is created with the built-in *all* partitioner for the replacement of *allreduce* operations.

3. Data Container: Create data storage space (`Laik_Data`) for all working vectors, including `norm`, `correlation`, `fwproj` and `image`.

4. Kernel Wrapping: The calculation kernels for the forward projection and the backward projection are encapsulated with an additional loop, in order to process non-contiguous data slices.

5. Communication Transformation: All the MPI communication routines are replaced by switching the partitioning of the respective LAIK data container to the *all* partitioning previously created. This triggers an implicit *allreduce* operation.

The workload of porting the application (excluding the preparation work) was approximately half a day without any previous experience with LAIK [Yan+18]. The LAIK version of MLEM is available as open source software on *Github*[5].

6.3.3. Evaluation of Application Example 1: MLEM

The evaluation of our LAIK-based MLEM implementation (in the following: *laik-mlem*) in comparison with the original MPI-based MLEM code (in the following: *reference mlem*) was carried out on the CoolMUC-2 system at the Leibniz Rechenzentrum der Bayerischen Akademie der Wissenschaft (LRZ) (the Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities) [6]. Each node on CoolMUC-2 is equipped with two processors with Intel Haswell architecture and 14 cores. The RAM

[4] Compressed Sparse Row [Saa03]

[5] `https://github.com/envelope-project/mlem`

[6] `https://www.lrz.de`, accessed in June 2019

available per node is 64 GB. More detailed information of CoolMUC-2 is provided in the Appendix A.2.

Both *laik-mlem* and *reference mlem* are compiled using *gcc* version 5 and Intel MPI version 2017. All binaries are compiled with the `-march=nativ -O3` flags to ensure maximum optimization. We ran *laik-mlem* and *reference-mlem* with 1 to 28 MPI processes. Furthermore, we pinned these processes equally among the four NUMA domains of a node (with the environmental variable for Intel MPI `I_MPI_PIN_DOMAIN=numa`). For both programs, we executed ten MLEM iterations and captured four independent measurements for each task/binary combination. All the experiments were carried out the same physical node. To eliminate delays from the file system, the initial time of loading the sparse matrix from the file system was not considered in the scalability evaluation. All evaluation is completed on a single node because the sparse matrix is small enough to fit into the system's main memory. Furthermore, the single-node setup allows us to understand the minimum overhead of LAIK. The original evaluation results were published in [Yan+18]. Figures 6.18 – 6.22 represent the evaluation results.

As shown in Figure 6.18, LAIK does not produce much overhead over the *reference mlem*. This is also reflected in Figure 6.19. As a result, a total speedup of up to 12x can be achieved. The same speedup is achieved with the *reference mlem*. Furthermore, Figure 6.19 shows that the speedup flattens out starting from approximately 16 MPI processes, and oscillates with a higher number of MPI processes. This is expected behavior, as MLEM is memory (bandwidth) bound [Küs+09]. The architecture of our testbed features a dual-socket design with two NUMA domains on each processor chip (also known as cluster-on-die), resulting in a total of four NUMA domains. This architecture negatively affects performance if the number of total MPI processes is not divisible by four.

To better understand the overhead produced by LAIK, Figure 6.20 shows the decomposition of runtime per iteration produced by different operations as a relative percentage. As expected, time spent in the backend (communication library, i.e., MPI) increases with the number of processes, as MLEM requires *allreduce* operations. Potential overhead produced by LAIK is very low and does not scale with the number of processes. This conclusion is also confirmed by Figure 6.21. Therefore, we conclude that the overhead of LAIK for MLEM is low, and it does not change the scaling behavior of MLEM.

We now compare different repartitioning algorithms. Since we implemented *laik-mlem* in a way so that it can handle a non-contiguous range of sparse matrix rows, we use two different (re-)partitioners to create two different partitionings for data migration. The first one is the *incremental* (re-)partitioner (cf. Section 5.2), which equally divides

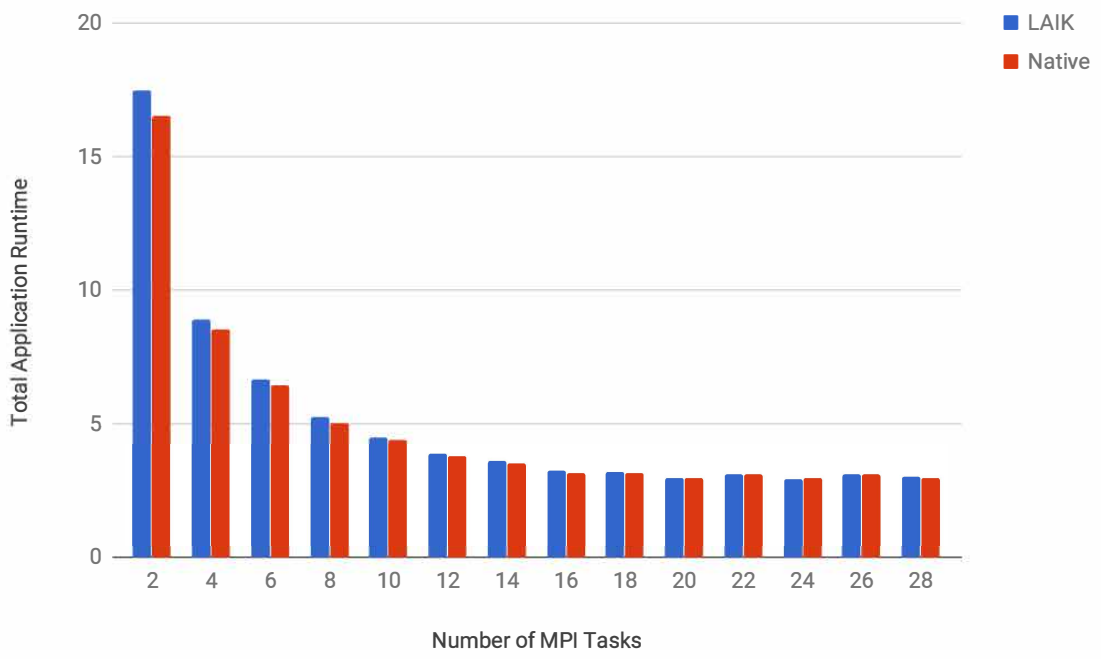

Figure 6.18.: Average Runtime per Iteration *laik-mlem* vs. *reference mlem* [Yan+18]

the workload of MPI instances leaving, and appends those to the remaining instances. This partitioner is implemented by the application programmer of MLEM. The second one is the *continuous* (re-)partitioner, which is the default *block* partitioning provided by LAIK. It calculates a new partitioning from scratch with the modified group, excluding those MPI instances leaving. The implementation for this partitioner is provided by LAIK.

For our experiment, we explicitly removed one of the working MPI instances after the sixth iteration. This way, we ensured that the time we measured was for repartitioning only, and not for the synchronizing the processes. The result is shown in Figure 6.22. One can see that the incremental (re-)partitioner outperforms the continuous one by a factor of two. The reason is that the MLEM code has to reload new parts of the sparse matrix from the file system in order to execute the computation. With the incremental partitioner, a significantly lower amount of data has to be reloaded and transferred because only the rows on the nodes leaving need to be loaded by a remaining process. With the continuous (re-)partitioner, a higher amount of data must be loaded from the file system because the workload ranges are shifted globally. This causes a higher time consumption for the first iteration after repartitioning. Further, we can see on Figure 6.22 that the execution time both before and after repartitioning remains constant, while the

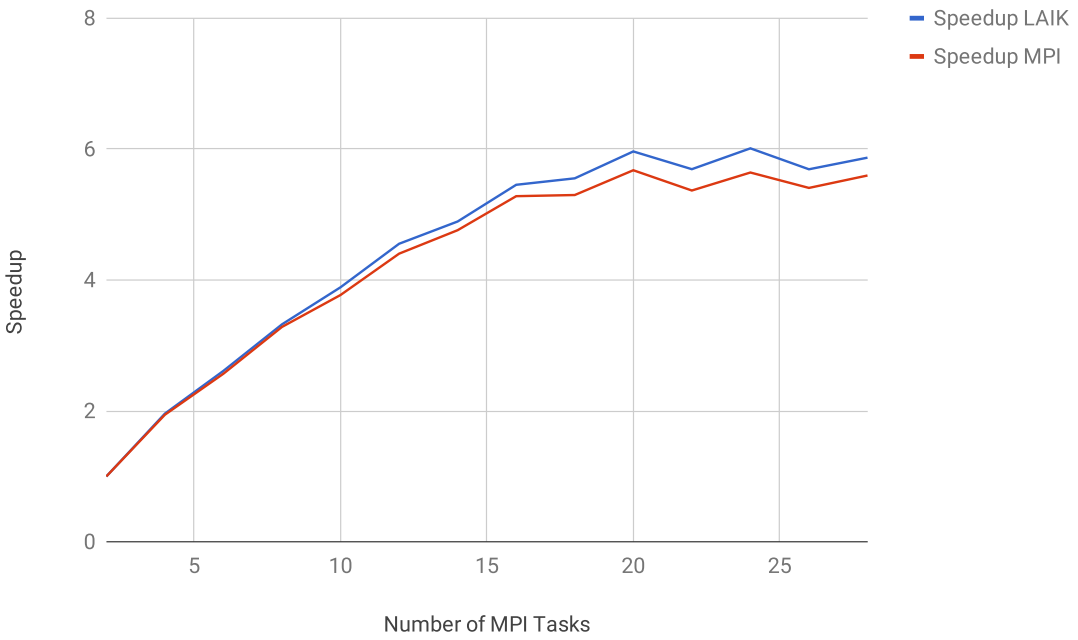

Figure 6.19.: Speedup Curve for *laik-mlem* vs. *reference mlem* [Yan+18]

time for the first iteration after repartitioning differs. Moreover, it is observable that the runtime per iteration increases by only a minimum after repartitioning, indicating that LAIK has restored the load balance across all the remaining MPI tasks. This observation demonstrates that data migration is successful and does not change the scaling behavior of MLEM. Therefore, we conclude that with LAIK, fault tolerance can be achieved by data migration.

Our experiments with different partitioners confirm our theory introduced in Chapter 5: Different repartitioning strategies are useful in different cases. For some applications, which are more rigid and do not support multiple slices with their kernels, the continuous (global *block*) partitioner works best. For other applications, which can be adapted to support non-contiguous data slices (such as MLEM), their performance benefits from incremental repartitioning.

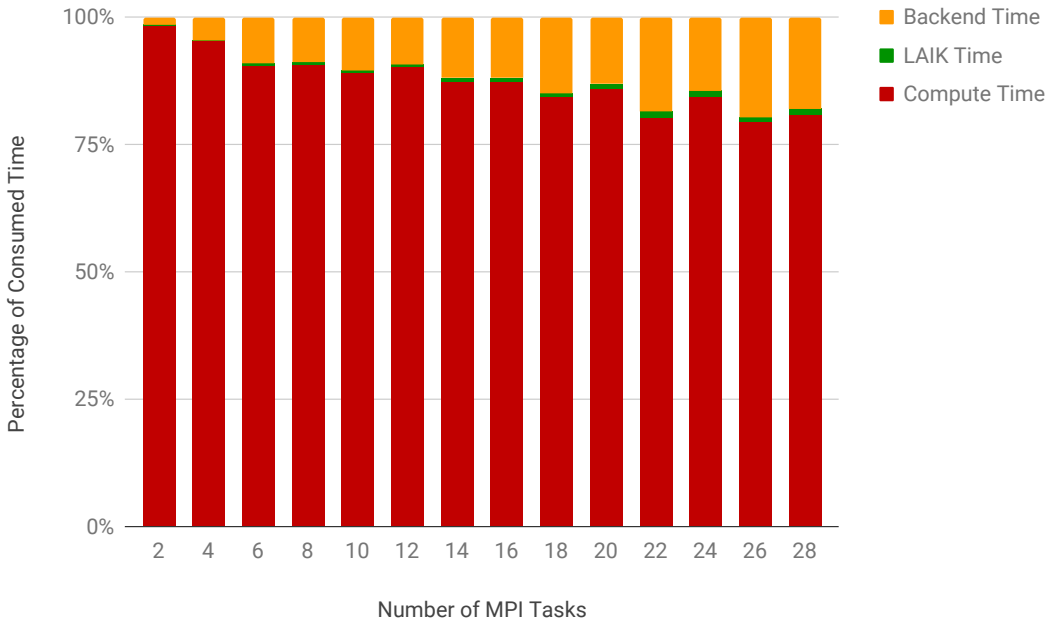

Figure 6.20.: Overhead Analysis for *laik-mlem* [Yan+18]

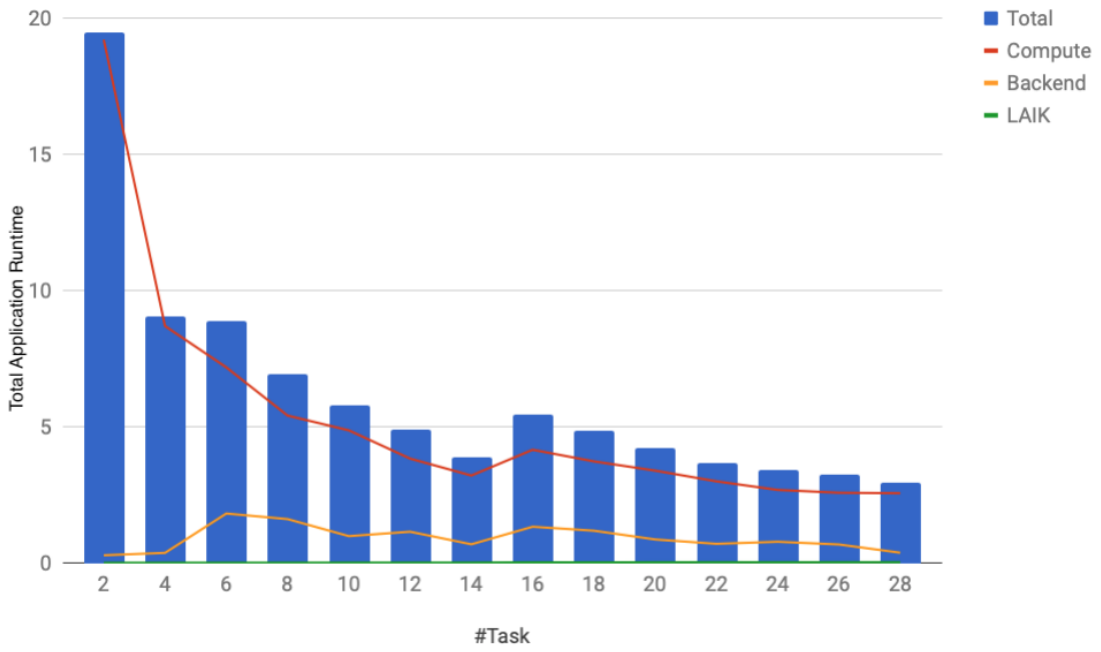

Figure 6.21.: Overhead Scalability Analysis for *laik-mlem* [Yan+18]

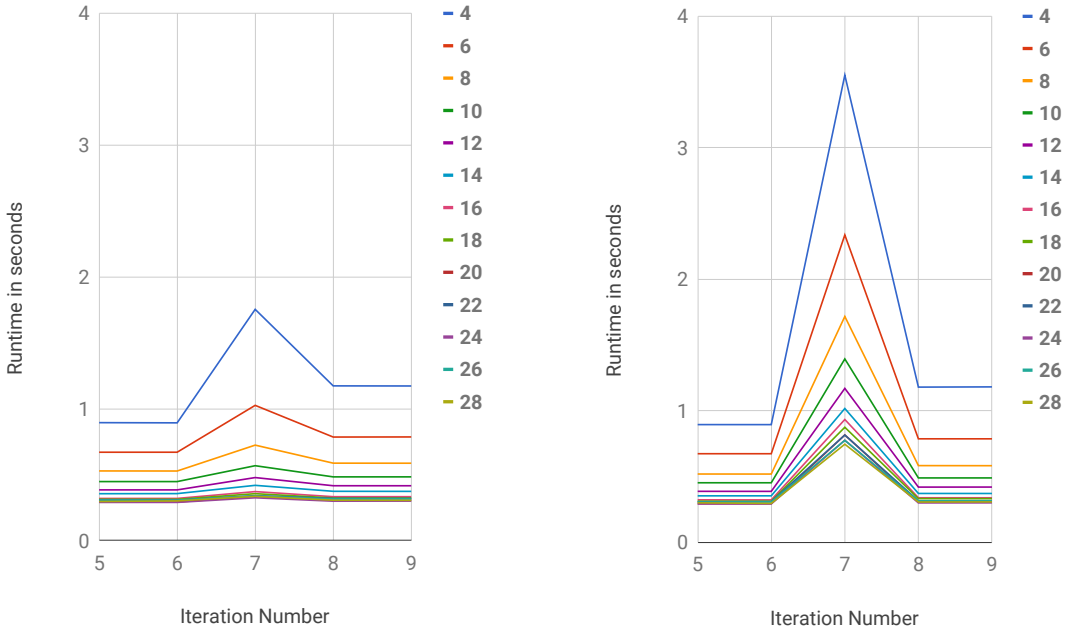

Figure 6.22.: Comparison of Different Repartitioning Strategies for *laik-mlem* [Yan+18]
Left: Repartitioning using the *Incremental* Partitioner
Right: Repartitioning using the Default *Block* Partitioner

6.3.4. Application Example 2: The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) Benchmark

With *laik-mlem*, we show the feasibility of achieving data migration to rescue data from a failing node. Furthermore, we have shown low overhead of LAIK for an application running on a single node. However, this experiment is not sufficient to prove the effectiveness of LAIK for large-scale (parallel) applications designed for cluster systems. Consequently, we have also ported another application – the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) benchmark – in order to understand the efficiency of LAIK for data migration.

LULESH is a benchmark for targeting the solution to the Sedov Blast Problem (SBP) [Sed46] using Lagrangian hydrodynamics [HKG]. The SBP represents a class of classical HPC applications. According to Hornung et al. [HKG], the reference LULESH implementation provided on *Github*[7] is drawn from a production hydrodynamics code from the Lawrence Livermore National Laboratory (LLNL).

The basic idea of achieving a solution to SBP lies in solving the Euler equations as shown in Equation 6.4, where $\overrightarrow{U}$ is the velocity, ρ is the density, e is the internal energy, p is the isotropic pressure, q is the artifical viscosity, and a_{HG} and $\dot{e}_{HG}$ are the acceleration and heating terms due to the hourglass filter [HKG].

$$\begin{aligned} \rho \frac{D\overrightarrow{U}}{Dt} &= -\overrightarrow{\Delta} \cdot (p+q) + \rho a_{HG} \\ \frac{De}{Dt} &= -(p+q) \cdot \frac{DV_{spec}}{Dt} + \dot{e}_{HG} \end{aligned} \tag{6.4}$$

The exact physics involved is beyond the scope of this work because we focus on porting the existing LULESH implementation to LAIK. While detailed information on physics and implementation of LULESH can be found in [HKG], here is a brief summary:

- The reference code simulates the SBP in three spatial dimensions x, y, z. A point source of energy is deposited at the origin as the initial condition of the simulation.
- The initial mesh is a uniform Cartesian mesh. It is divided into logically-rectangular collections of elements, which are called *domains*. Each domain – which is implemented as a C++ class – represents a context for data locality. It holds the data for all the elements as well as the nodes surrounding those elements. The schematic of LULESH mesh decomposition is depicted in Figure 6.23. Boundary data is replicated if the domains are mapped to different processors.

[7] `https://github.com/LLNL/LULESH`, accessed in December 2018

- There are two different types of data stored in the *domain* class: the initially cube-shaped finite elements (in the following: *elemental* data), and the related vertices of the finite elements (in the following: *nodal* data). There are a total of 13 nodal variables and 13 elemental variables in each domain. A detailed list is given in Tables C.1 and C.2.

- Two different operations are carried out in each iteration (timestep):

  `CalcTimeIncrement()`, which calculates the time increment Δt^n for the current timestep.

  And `LagrangeLeapFrog()`, which advances the solution of the elements from t^n to t^{n+1}.

  The pseudocode for the MPI-based implementation is given in Algorithm 8.

- For our work, we use the MPI/OpenMP hybrid implementation of LULESH 2.0. There are two main types of communication: (1) Halo exchange at borders of domains for stencil-wise updates of data structure (e.g., volume gradient). (2) Reduction (aggregation) of calculations from the element quantities to the surrounding nodes (e.g., force vector). MPI communication is explicitly coded, and communication routines are pre-initialized (e.g., finding out communication partners and setup buffers) at mesh decomposition time.

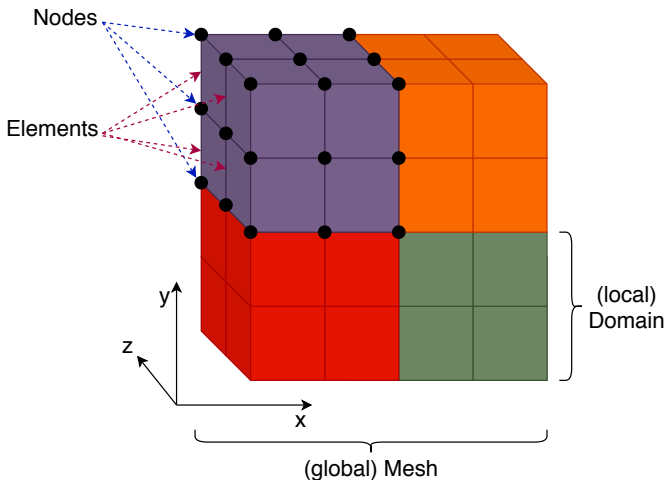

Figure 6.23.: LULESH Mesh Decomposition Schematic [HKG]

Algorithm 8: MPI-based Implementation of LULESH [HKG]

```
sz    : Input: Size of the Mesh
nIter : Input: The Number of Total Iterations

MPI_Init();
Domain locDom ← InitMeshDecomposition(rank, size, sz);
while (!endOfSimulation && nIter>0) do
    CalcTimeIncrement();
    LagrangeLeapFrog();
    nIter–;
end

if rank==0 then
    printStatistics();
end
MPI_Finalize();

---

Subroutine LagrangeLeapFrog()
    LagrangeNodal();
    //Advance Node Quantities
    CalcForceForNodes();
    //Calculate Node Forces
    LagrangeElements();
    //Advance Element Quantities
    CalcTimeConstrantsForElems();
    //Calculate Time Constrains for Elements
```

Porting LULESH to LAIK

After having identified the basic program structure, data structures, and communication patterns, we ported LULESH to LAIK. In the remainder of this work, *reference lulesh* is used to reference the original MPI-based LULESH implementation provided by the Lawrence Livermore National Laboratory (LLNL), and *laik lulesh* is used to reference our ported version. A detailed documentation for the porting was published in our previous publication [Rao+19]. The main goal of porting LULESH to LAIK is to keep the number of changes as small as possible and achieve fault tolerance by allowing LULESH to migrate data from any potentially failing node. Therefore, several rules were introduced to keep the porting progress aligned with our goals:

- Code accessing data structures and computational kernels must not be changed.

- Explicit MPI communication code should be replaced by transitions between different partitionings in LAIK in order to achieve implicit communication.
- By porting data structures from *reference lulesh* into LAIK data containers, the above mentioned implicit communication can be fully performed under LAIK's control. This also enables us to use LAIK's automatic data migration functionalities from the *Data* API Layer for fault tolerance purposes (cf. Section 6.1.6).

Accordingly, the two different types of data structures are completed in two steps:

1. We transform original communication by reimplementing all the data structures listed in Table C.1 and C.2 in Appendix C with `Laik_Data` and let LAIK to maintain these data structures, which are then updated in the regular iterations.
2. For fault tolerance, also data structures which are used purely locally are also transferred to be maintained by LAIK because it also requires migration whenever there is a data migration request. An overview of data structures is given in Table C.3 in Appendix C.

Furthermore, small modifications are required in the main iteration loop to check for repartitioning requests and trigger a data repartitioning in LAIK.

The detailed steps performed for porting LULESH to LAIK are as follows: [Rao+19]:

1. **Adaptation of Data Structures which Requires MPI Communication during Kernel Execution.**

   LULESH uses asynchronous communication for force fields ($\overrightarrow{F}$) and *nodalMass* followed by aggregation. In order to support the transitions (and implicit communication) as specified by LAIK, data must be bound to different partitionings. As LULESH kernels operate with local indexes within a domain instead of global numbering, the default partitioners in LAIK which are based on global indexes cannot be used. Therefore, we have created a customized partitioner called *overlapping* on the global nodal index space, which is illustrated in Figure 6.25. Nodal index space is partitioned among all the tasks, in which the neighboring tasks share one layer of nodes. *Laik lulesh* uses partitioners similar to the reference code with different layouts. However, some adaptation has to be made, as LULESH relies on 1D data structures which are mapped from a 3D domain in the x–y–z direction. Instead of a compact data structure as provided in *reference lulesh* (cf. Figure 6.24 right), our implementation of *laik lulesh* relies on a non-compact x–y–z layout (cf. Figure 6.24 left) based on many "pieces". The overlapping region is shared between two neighboring tasks and updated by each task independently. By transiting from the same overlapping partitioning, LAIK ensures that the

overlapping region is properly reduced among all the sharing process instances, thus achieving the same communication as provided explicitly by *reference lulesh*.

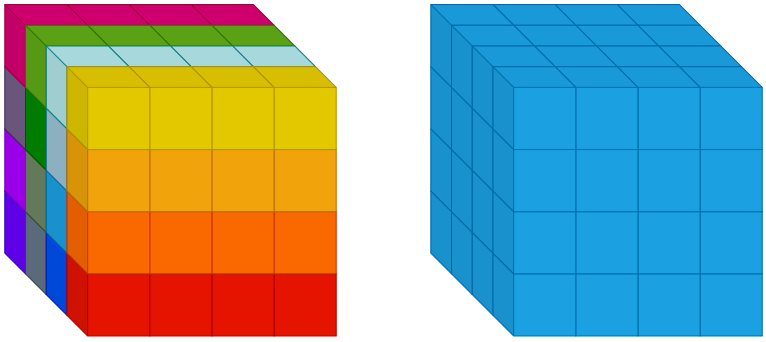

Figure 6.24.: Illustration of a Domain for Elemental Data with a Problem Size of 4x4x4 Elements [Rao+19]

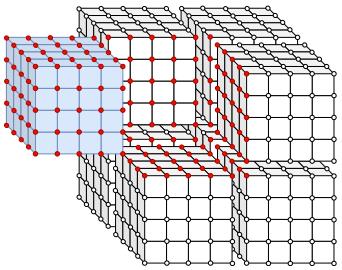

Figure 6.25.: Illustration of Nodal Data with Overlapping Partitioning [Rao+19]

LULESH also uses an asynchronous halo exchange pattern for velocity gradient fields, which is different to the overlapping communication pattern. For supporting halo exchange, we have created two partitioners – *exclusive* and *halo partitioners* – for these data. A transition between these two partitionings results in the required communications for the halo exchange. This is illustrated in Figure 6.26).

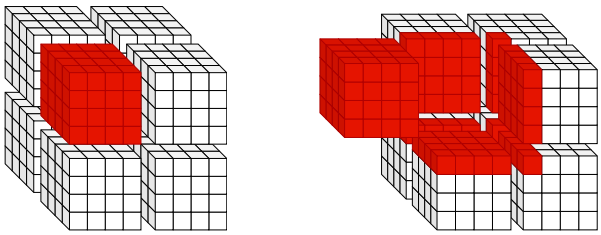

Figure 6.26.: Illustration of the Elemental Data with Exclusive Partitioning (left) and Halo (right) Partitioning [Rao+19]

2. **Adaptation of Additional Data Structures Required for Live Data Migration**

   LULESH uses a large number of additional temporary data structures. For live data migration, these data structures need to be handed over to LAIK for migration as well. Therefore, this data are fully transferred to LAIK's responsibility with built-in *block* partitioners according to data distribution before and after repartitioning at the time of the respective data migration request. After data migration, this data are restored to LULESH's original structure.

   Moreover, we modified the main `while` loop to check for a repartitioning request in each iteration. If a process is no longer a part of an active calculation after repartitioning, this process is then discarded by calling `laik_finalize()`.

3. **Additional Performance Optimizations**

   Multiple optimizations are implemented in order to enhance the performance of *laik lulesh*. We use LAIK's transition caching feature to pre-calculate the transitions between different iterations. Moreover, these transitions are used for calculating and caching their corresponding action sequences between different partitionings because these sequences are executed in each iteration. Moreover, our implementation creates many 1D "pieces" because the index regions are not contiguous for a given domain on a global view in the x–y–z domain decomposition. After each transition, pointers to these data structures typically become invalid. This causes major performance overhead. Consequently, we utilize the pointer reservation function (cf. Section 6.1.6 from LAIK to reduce this overhead.

Porting LULESH to LAIK took approximately six months for an experienced MPI programmer who was not at all familiar with LAIK and LULESH. The code is available as an open source project on *Github*[8]. It is important to mention that *reference lulesh* only supports cubic numbers of process instances (e.g., 1, 8, 27, 64, ...). This limitation is not changed in *laik lulesh* to align with the kernel design of LULESH.

[8] `https://github.com/envelope-project/laik-lulesh`, accessed in December 2018

6.3.5. Evaluation of Application Example 2: LULESH

Extensive experiments and measurements were performed on SuperMUC Phase II (in following: *SuperMUC*) in order to evaluate the performance of our ported LULESH code. SuperMUC consists of 3072 nodes, each equipped with two Intel Haswell Xeon Processors E5-2697v3 and 64 GB of main memory. A detailed description of SuperMUC is given in Appendix A.1.

Both the *LAIK Library*, *laik lulesh*, and *reference lulesh* are compiled with Intel Compiler version 2017. The MPI library used is IBM MPI version 1.4. Furthermore, all binaries are compiled with `-O3 -xCORE=AVX2` to achieve maximum optimization by the compiler.

For each experiment/binary combination, a total of 5 runs was carried out to exclude interference from the system. Weak scaling (cf. Section 2.3.3), strong scaling (cf. Section 2.3.3), and repartitioning tests were performed. The results were originally published in [Rao+19].

Weak Scaling

We selected a problem size of 16 (option `-s 30`) for both *laik lulesh* and the reference code. The upper bound of the number of iterations is set to 10000 iterations (option `-i 10000`). The result are presented in the Figures 6.27 and 6.28. The time reported here does not include initialization and finalization. For weak scaling, we ensured that the workload remained the same for each participating process involved (cf. Section 2.3.3). We calculated the runtime per iteration by dividing the reported time by the number of iterations.

Figure 6.27 shows the comparison of the normalized runtime per iteration between *laik lulesh* and *reference lulesh*, which are noted as box plots in the vertical axis. The horizontal axis represents the number of MPI instances used in each experiment. An increase in iteration runtime with an increasing number of MPI tasks from *laik lulesh* is clearly observable. In contrast, the reference code scales almost perfectly with only a slight increase [Rao+19].

To showcase the exact difference in the normalized runtime per iteration, the red line in Figure 6.28 represents this overhead. We can see that it scales up with the number of processes. Our hypothesis for the reason for this increasing overhead is the lack of support for asynchronous communication in LAIK. By analyzing the Figures 6.27 and 6.28, one can see that LAIK introduces a constant part of the overhead which does not scale with the number of processes, and a non-constant part of the overhead which scales with the increasing number of processes. The source of the constant overhead is the added layer of abstraction for data structure access, while the scaling part is most likely induced by the different MPI communication pattern [Rao+19].

Nevertheless, we conclude that the overhead is an acceptable range for a mid-size run (up to 512 parallel process instances).

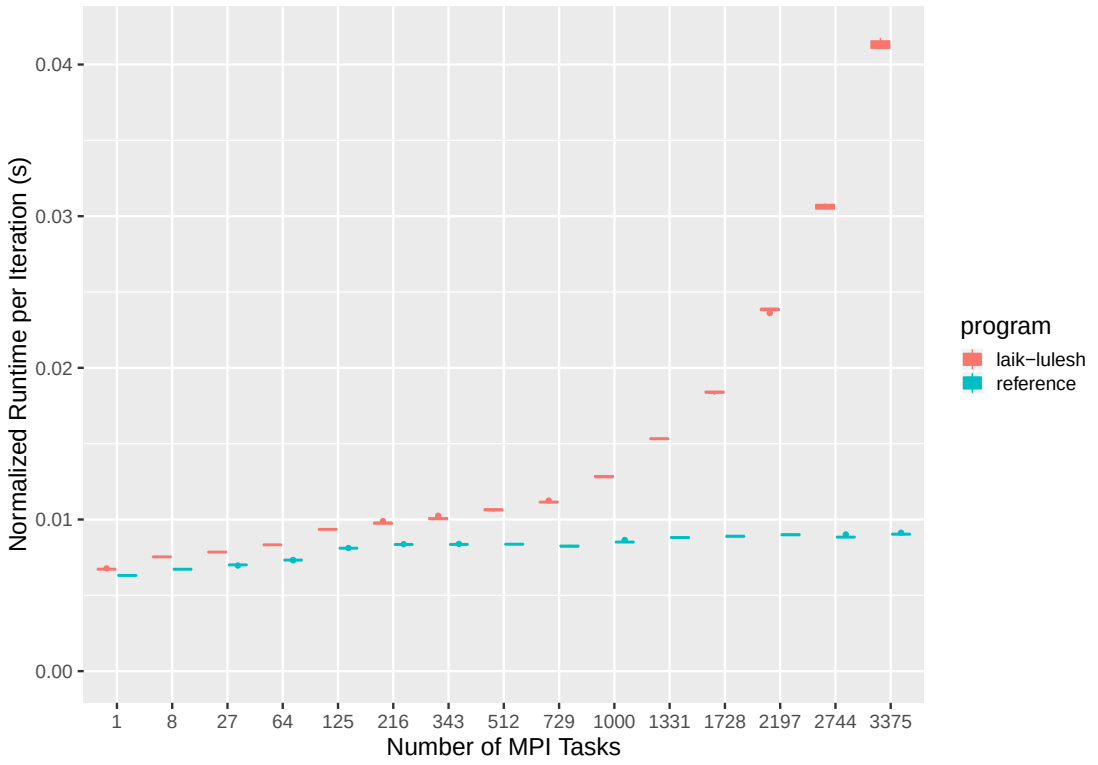

Figure 6.27.: Weak Scaling Runtime Comparison for *laik lulesh* vs. *reference lulesh* with `-s=16` [Rao+19]

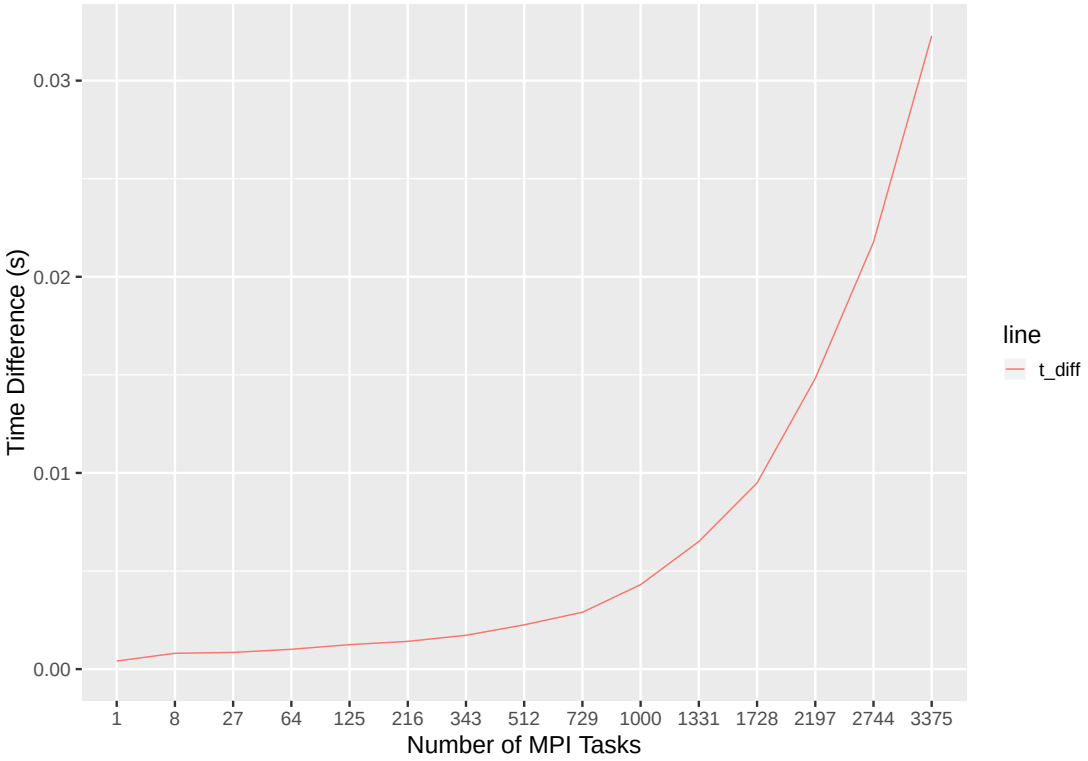

Figure 6.28.: Overhead of *laik lulesh* over *reference lulesh* with `-s=16` [Rao+19]

Strong Scaling

Since *laik lulesh* and *reference-lulesh* only support a cubic number of processes, we carefully constructed a test case for strong scaling. The design of the test case is presented below.

Let C be the global 3-dimensional problem size, which needs to be constant for all strong scaling experiments (cf. Section 2.3.3), and s be the local 1-dimensional problem size (parameter `-s`).The term $C = s^3 \times p$ applies. Furthermore, let p be the number of MPI process instances in the parallel execution, and $S = C^{\frac{1}{3}}$, then following implication applies

$$(C = s^3 \times p) \implies (S = s \times p^{\frac{1}{3}}). \tag{6.5}$$

As the limitation for the cubic number of MPI processes still applies, $p^{\frac{1}{3}}$ and s must be natural numbers. For the sake of simplicity, we set up our strong scaling experiments with $p^{\frac{1}{3}}$ being the powers of 2 and $S = 256$. The resulting corresponding tuples of (p, s) which are used in this experiment for strong scaling are therefore [Rao+19]:

$$\begin{aligned} (1^3 = 1, 256), \\ (2^3 = 8, 128), \\ (4^3 = 64, 64), \\ (8^3 = 512, 32), \\ and (16^3 = 4096, 16). \end{aligned} \tag{6.6}$$

The results from these experiments are illustrated in Figure 6.29. Note that the vertical axis is log scaled. As in the weak scaling experiment, similar scaling behavior can be observed for both LAIK and the reference version with up to 512 processes.

With 4096 processes, *laik lulesh* shows a significant overhead, where our port is at least a factor of 2x slower than the reference code. In addition, we can observe that the overhead curve first decreases, then increases with a large number of processes. Figure 6.29 further confirms the relatively constant overhead for experiments with 8, 64, and 512 processes, where the added abstraction by using 1D slices in the LAIK implementation is the dominant source of overhead. The same scaling overhead as in the weak scaling test for *laik lulesh* with a large number of processes can also be observed in Figure 6.29. This again confirms that the source of scaling overhead is the result of a lack of support for asynchronous communication in LAIK, which scales with the number of point-to-point communication (and the number of processes) [Rao+19].

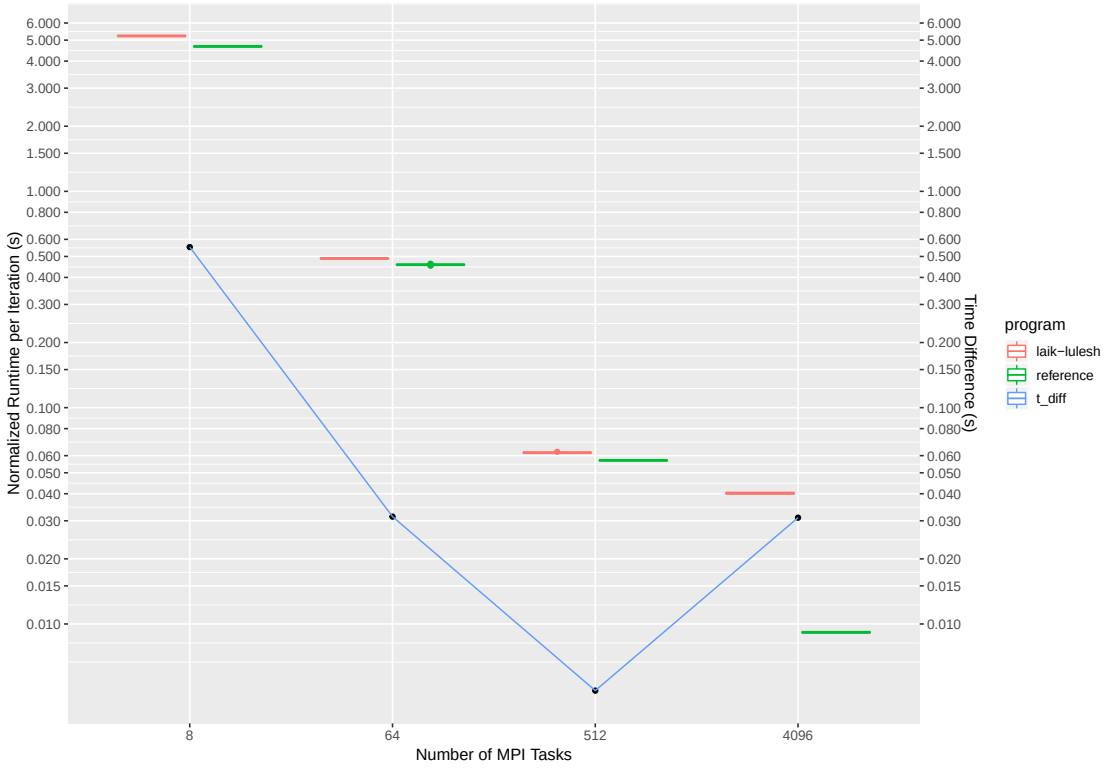

Figure 6.29.: Strong Scaling Comparison for *laik lulesh* vs. *reference lulesh* [Rao+19]

Repartitioning

Using LAIK, we can now migrate the execution of *laik lulesh* to a smaller number of MPI processes at runtime. This is the desired result for fault tolerance through data migration provided by LAIK.

To understand the impact of migration on the scaling behavior, we conducted a row of scalability experiments, which are similar to the strong scaling experiments.

Again, we choose $p^{\frac{1}{3}}$ being the powers of two, and $S = 64$. We simulate a potential fault by enforcing a repartitioning and data migration to the smaller, next supported number of MPI process instances, where $p^{\frac{1}{3}}$ remains a power of two. This results in the following repartitioning experiments: From 8 to 1, from 64 to 8, and from 512 to 64, respectively. [Rao+19]

We held the number of iterations (parameter `-i`) constant at 2000 iterations for all the experiments. We executed the kernel for 250 iterations with the initial number of MPI processes and then performed the repartitioning. Finally, the kernel was executed for another 1750 iterations until completion. This way the total runtime before and after migration should be the same. A total of 5 runs were executed on SuperMUC for each configuration. [Rao+19]

Figure 6.30 showcases the result from this experiment. On the horizontal axis, the type of migration is given. The vertical axis again represents the normalized time per iteration (note that this is log(2)-scaled). As expected, both time before and after repartitioning is linear on a log scale. In addition, the normalized runtime for a given number of MPI processes (e.g. 64) is almost the same, regardless of whether it is an initial set number of processes or the final state after repartitioning. This means that a data migration performed by LAIK does not affect the runtime behavior of *laik lulesh* [Rao+19].

The measurements reflect a good result in the effectivity of repartitioning and automatic data migration performed by LAIK. However, the limitation of only allowing a cubic number of MPI processes significantly reduces the gain of proactive migration because the failure of a single node leads to the elimination of a large number of processes [Rao+19].

Table 6.3 shows the time consumption for repartitioning and data migration in seconds. Compared to the long runtime of LULESH, the time effort for the repartitioning remains negligible.

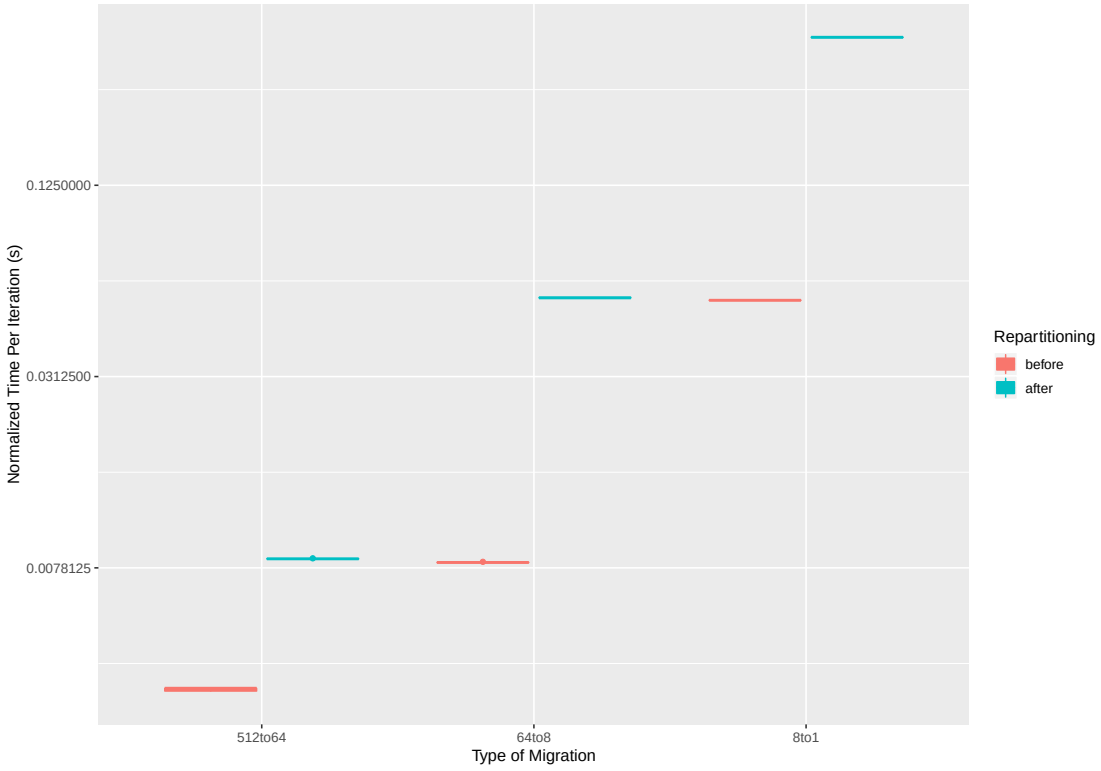

Figure 6.30.: Runtime Comparison Before and After Repartitioning for *laik lulesh* [Rao+19]

Table 6.3.: Time Consumption for Data Migration Using LAIK

Configuration	Time for Repartitioning
512 to 64	∼1.5678s
64 to 8	∼0.8803s
8 to 1	∼1.6979s

6.4. Discussion on Effectiveness of Data Migration with LAIK

In conclusion, LAIK is an approach to assist programmers in enhancing their application with data migration capabilities. It features a layered and modular design, which programmers can utilize to adapt their applications in an incremental manner.

With LAIK, an application can transfer the responsibility of data partitioning to LAIK, thus achieving implicit communication by transitions between different partitionings. Fault tolerance with data migration is triggered automatically because LAIK can simply exclude failing nodes from a partitioning and redistribute their partitions to other process instances.

We have further evaluated the effectiveness and performance of LAIK using two examples in this chapter: *MLEM* and *LULESH*. We have performed different tests for repartitioning and data migration and also conducted a scalability analysis. We can now summarize the advantages and disadvantages of LAIK as follows.

6.4.1. Advantages of LAIK

LAIK provides a list of advantages, most prominently:

- LAIK achieves a low overhead for fault tolerance per *design*. LAIK is an application-integrated approach for fault tolerance with data migration. The user has full control over the migration of application data in case of a fault, hence reducing the overhead to a desired minimum.
- LAIK ensures incremental porting. Managing existing code is a major challenge within the context of parallel computing. With LAIK, an existing application can be ported in a step-by-step manner by utilizing the different layers of the APIs.
- LAIK enables fault tolerance automatically by repartitioning and data migration, if the data responsibility has been fully transferred to LAIK. With LAIK's *Data APIs*, the user only has to specify *where* the data is needed and *when*.

- LAIK achieves communication management by using implicit communication, if *Data APIs* are used. This way, communication optimization can be fully realized in a future release of LAIK.
- Experiments with MLEM and LULESH have shown that LAIK only adds a minimum amount of runtime overhead. Although tests with LULESH have indicated some scaling-based overhead, that overhead is caused by the implementation of our LAIK prototype and not by design. The additional layer of abstraction introduced by LAIK (index space) only adds 10% more overhead and remains constant.

6.4.2. Disadvantages and Limitations of LAIK

Besides the advantages of LAIK, there are some disadvantages and limitations:

- The time demand of porting an existing complex applications to LAIK is very high. For LULESH, this was 6 person-months for an experienced parallel programmer. This was mainly due to the transition from pure MPI to LAIK. For MPI-based applications, a local view of data is typically used. However, as LAIK works on the global distribution of data, the data handling has to be adapted to a global one. For applications with complex data structures, this change requires a significant amount of adaptations. The return-on-investment of such adaptations might be too low for code owners to port their code in order to support data migration.
- LAIK's fault tolerance capability relies strongly on the backend support. If the backend (such as MPI in our prototype) does not support fault tolerance, LAIK cannot achieve true fault tolerance. In the case of MPI, since we cannot safely shut down failing MPI ranks, it is not possible to achieve fault tolerance with LAIK.
- LAIK currently only supports regular data structures, which is a huge limitation on real world applications. To make LAIK become a competitive solution for fault tolerance based on data migration, future implementations will have to support arbitrary data structures.

6.4.3. Lessons Learned

Based on the experiments and experience in porting MLEM and LULESH, we believe that LAIK is a possible candidate for achieving fault tolerance by using data migration. Furthermore, with LAIK, we have shown that data migration is a promising technique for fault tolerance in parallel applications.

However, additional implementation and optimization efforts have to be invested to make LAIK a competitive solution for application-integrated fault tolerance. The rather complex concept of LAIK can be an overkill for a fault tolerance concept only. Moreover, adequate support for fault tolerance has to be integrated into popular runtime libraries such as MPI. Without such support, application-integrated fault tolerance methods can hardly become effective.

Based on the experience with LAIK, we are going to introduce *MPI sessions* and *MPI process sets*, featuring a leaner design in order to achieve fault tolerance by using data migration.

7. Extending MPI for Data Migration: MPI Sessions and MPI Process Sets

In the last chapter, we introduced a new user-level library called LAIK, which can assist programmers with data migration for existing and new applications. However, there is still one crucial drawback in using LAIK: The actual fault-tolerant characteristics strongly rely on the support of the communication backend and the process manager. With MPI being the de facto standard for parallel applications, our LAIK prototype features a communication backend which is based on MPI. However, since MPI does not support the safe shutdown of a process instance, true fault tolerance with LAIK and MPI as a backend is not possible at the moment. Consequently, we have decided to take a closer look at the MPI standard itself and identify potential gaps and room for improvement to the MPI standard.

Many efforts in extending MPI with fault tolerance capability have been made in recent years. *User Level Failure Mitigation (ULFM)* in MPI [Bla+12] is the most prominent proposal. Bland et al. [Bla+12] have suggested the introduction of a list of new MPI constructs, which can be used to inform the application about a failed communication in MPI. Here are the key concepts [Bla+12]:

- `MPI_COMM_FAILURE_ACK` and `MPI_COMM_FAILURE_GET_ACKED` are responsible for determining which processes within an MPI communicator have failed. After acknowledging the failures, a program can resume any point-to-point communication between healthy process instances.
- `MPI_COMM_REVOKE` allows the application to cancel an existing communicator. It is required to propagate error information to those processes, which are not affected immediately by the failure.
- `MPI_COMM_SHRINK` creates a new communicator by removing failed processes from a revoked communicator. This call is collective also executes a consensus protocol to make sure that all processes agree on the same process group.
- `MPI_COMM_AGREE` is an agreement algorithm which ensures strong consistency between processes. This collective function provides an agreement, even after a

failure or in the event of a communicator revocation.

With ULFM-MPI, communication failure is exposed to the user application. The programmer can achieve user-level fault mitigation by revoking an existing communicator with failed processes. The application can then create a derived communicator without the failed processes and continue the calculation.

Another approach to making MPI more dynamic is called *Adaptive MPI (AMPI)* [HLK04]. Huang et al. utilize a virtualization called *processor virtualization*, which detaches a physical process instance from a physical processor. Instead of issuing real processes at user level, AMPI creates virtual processes that can be later mapped to a set of processors. This way, AMPI can achieve fault tolerance by utilizing techniques such as process migration (cf. Section 4.2.4) or checkpoint&restart (cf. Section 4.2.3). Although AMPI is designed to allow dynamic load balancing, it can achieve application transparent fault tolerance with only minimal code changes required. Special MPI calls such as `MPI_Migrate` and `MPI_Checkpoint` are added to support these operations. The implementation of AMPI is based on the runtime system of *Charm++* [KK93]. This requirement limits the deployability of AMPI for a wide range of real-world scenarios.

Invasive MPI (IMPI) [Ure+12] is another attempt to enhance the flexibility of MPI. IMPI enables dynamic process management functionalities by adapting the underlying process manager. Urena et al. introduced a new implementation for the Simple Linux Utility for Resource Management Workload Manager (SLURM) named *Invasive Resource Manager (IRM)*, which allows destroying and spawning new processes at program execution time. The added MPI call `MPI_Comm_invade` can be used to secure additional resources from the job scheduler. The new call `MPI_Comm_infect` enables the application to expand itself to the newly added resources by creating new process instances. `MPI_Comm_retreat` allows the application to migrate to a smaller number of process instances. Data migration can be achieved by the application by withdrawing itself to a subset of the initial process instances. However, the requirement of a modified job schedule significantly limits it from being deployed by any large-scale data center.

In this dissertation, we follow an approach similar to the ULFM MPI by proposing a standard extension to the *MPI Forum*[1]. However, unlike ULFM, which focuses on the failure mitigation after a process has failed, we focus on the proactive migration of data before any predicted upcoming faults actually occur. Using *MPI sessions*, the most recent proposal based on the MPI standard extensions, we introduce a technique for

[1]MPI Forum is the governing and standardization task force for MPI standards. More information is provided on `https://www.mpi-forum.org`.

dynamic process management for MPI sessions. Our goal is to allow changes to be made to the size of participating process instances at any time of application execution.

In this chapter, we will introduce the basic concepts of *MPI sessions*. We further present our proposed standard extension as well as an implementation prototype for our extension – the *MPI process sets*. Finally, we show the feasibility of performing data migration with MPI sessions/sets by using the previously used examples of *MLEM* and *LULESH*.

7.1. MPI Sessions

MPI sessions is a proposed extension to the MPI standard, which is likely to be integrated into the MPI standard version 3.2. The original idea of MPI sessions was introduced by Holmes et al. in 2016 [Hol+16]; active discussions and suggestions for improvement have been made and updated since then.

According to Holmes et al. [Hol+16], the original intention of MPI Sessions was to solve a range of essential issues:

- Scalability limitations due to the dense mapping of the default communicator `MPI_COMM_WORLD`.
- Lack of isolation of libraries. MPI can only be initialized once and does not work well in combination with threads.
- Conservative role of the runtime systems. MPI does not really interact with the runtime system, such as the OS or job scheduler.
- Lack of support for use cases outside the HPC field domain.
- Lack of support for **fault tolerance**, due to the impact of a failing process on the trivial `MPI_COMM_WORLD`.

The solution provided in the proposal by Holmes et al. [Hol+16] is based on the idea of making MPI less rigid. The main ideas are [Hol+16]:

1. Allowing live instantiation and deinstantiation of MPI library instances at runtime. This way, the MPI library can be instantiated many times, allowing multiple MPI instances within the same process – each identified by an *MPI session handle* at the same time.
2. Decoupling of communicator and MPI library instance. Instead of using the trivial communicator `MPI_COMM_WORLD`, *named sets* of processes should be used. By using Uniform Resource Identifiers (URIs) (i.e., `mpi://WORLD`), new dynamic

functionalities can be implemented without affecting backward compatibility. Furthermore, since the URIs are shared resources between Application, MPI library, and the runtime system, information exchange among these systems can be achieved.

3. Creation of new communicators without an existing communicator. Currently, MPI only allows the creation of new communicators from an existing communicator. With MPI sessions, creation of new communicators should be possible directly from a group, without the intermediate step of another communicator. This downplays the role of the trivial communicator `MPI_COMM_WORLD`.

4. Adding support for topology awareness. Topology-aware communicators can be created to allow efficient communication.

5. Enhance communication efficiency by removing `MPI_COMM_WORLD` and `MPI_COMM_SELF`.

6. Improve usability by implicit calls to `MPI_Init` and `MPI_Finalize`.

This proposal was eventually edited into a draft document for the MPI standard – MPI standard 3.2. Two new objects, alongside with adaptations for the existing objects, have been introduced to implement the above mentioned new functionalities [Mes18]:

- MPI Session: The MPI session is a local, immutable handle to an MPI library instance. It is created by calling `MPI_Session_init` and should be destroyed by `MPI_Session_finalize`. The MPI session object represents an MPI library instance and holds all the required information regarding configurations, groups, and communicators. With MPI sessions, the legacy `MPI_Init` and `MPI_Finalize` have become optional. These legacy calls are retrofitted to provide backward compatibility for the legacy `MPI_COMM_WORLD` and `MPI_COMM_SELF` communicators.

- Process Sets: A process set is used to hold information regarding the physical processes in which an MPI library instance is running. Detailed definition of process sets is still under discussion and has not yet been standardized in the draft [Mes18]. The process set object is used to map runtime system information to MPI instances.

- MPI Group: The semantics of MPI groups remain the same. However, it is now bound to a process set in order to provide both backward compatibility and the dynamic environment introduced by MPI sessions. A new call `MPI_Group_from_pset` has been introduced to enable a programmer to create a group from an active process set.

- MPI Communicator: Similar to the MPI group, the semantics of MPI communicators remain unchanged. However, a new call `MPI_Comm_create_group()` is introduced to allow communicator creation without any preexisting communicators. This is required to loosen the requirement on the `MPI_COMM_WORLD` and `MPI_COMM_SELF` communicators.

Figure 7.1 illustrates the relation of these objects in MPI with the sessions concept. The yellow objects have already been proposed by the MPI forum, while the gray ones have not been proposed yet. An MPI communicator is derived from an MPI group, and an MPI Group is bound to a process set. The process set takes information from the runtime system and is part of an MPI session.

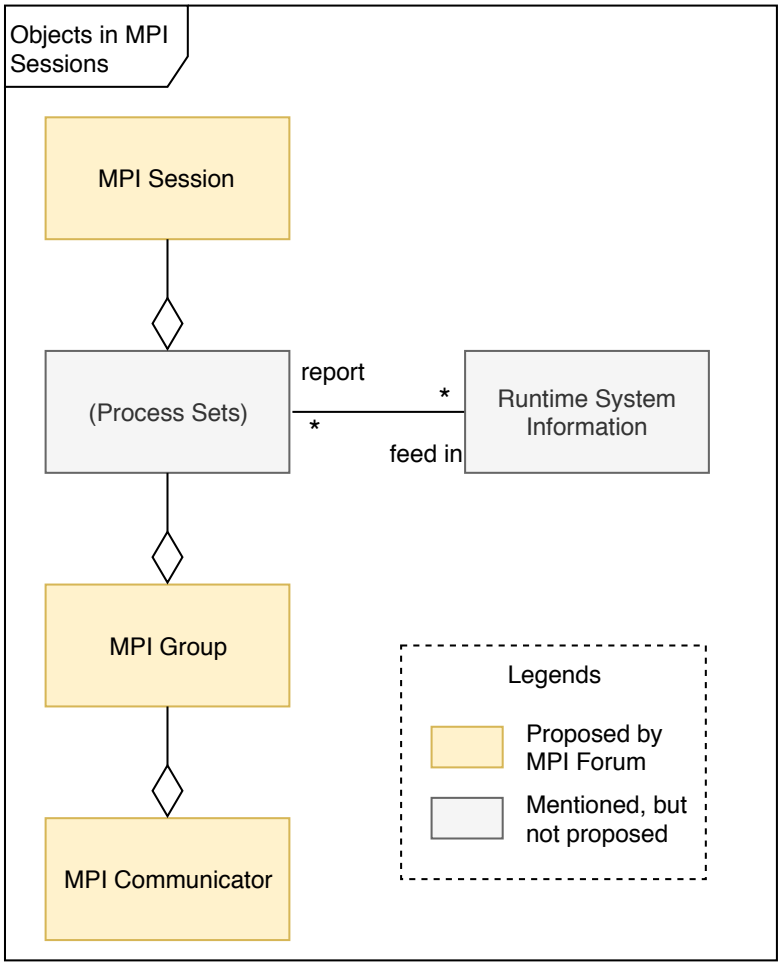

Figure 7.1.: Relations between Different Objects in MPI Sessions

7.1.1. MPI Sessions and Fault Tolerance

As Holmes et al. [Hol+16] point out, MPI sessions attempt to make the currently mandatory calls `MPI_Init` and `MPI_Finalize` as well as the global objects `MPI_COMM_WORLD` and `MPI_COMM_SELF` optional. This way, failure mitigation and fault tolerance are much easier to accomplish because a fault does not propagate to a global level automatically. This goal covers current efforts in the MPI Forum, which is considering mechanisms to determine faults in communicators and trigger a mitigation on the detection of any failure [Bla+12].

As we pointed out in the evaluation of LAIK (cf. Section 6.4), a current limitation of LAIK is that its backend based on MPI does not provide fault tolerance. Consequently, we cannot shutdown "empty" processes which are free from workload. However, with the concept in MPI Sessions, it is possible to implement a slightly adapted backend for LAIK, which is truly fault-tolerant by updating participating process sets and shutting down the "empty" processes.

7.1.2. MPI Sessions and Data Migration

Apart from data migration with LAIK, maintainers of existing applications can also achieve data migration with the proposed MPI sessions. Figure 7.2 illustrates a simple process of data migration based on the MPI sessions proposal. Unlike LAIK, since MPI sessions can exchange information with the runtime system, a failure prediction or change in an active process set can be passed to an application by MPI sessions. On receiving the information about an upcoming failure, manual repartitioning and data redistribution are performed. Afterward, the current process set can be adapted and the new MPI group and communicator can be created, respectively. Finally, communication routines can be migrated to the new communicator.

As one can easily see, the role of the process set is most crucial for the functionality of data migration and fault tolerance. As the MPI sessions are local (which means they are only known by the process which creates them) and static (which means they cannot be changed after their creation), any failure prediction at runtime have to be supported by the process sets. Therefore, in the remainder of this chapter, we will propose numerous design considerations and discussions for MPI sessions/sets, as well as our reference implementation in order to support fault tolerance based on data migration. At the end of this chapter, we will present our first evaluation results of fault tolerance based on data migration using MPI sessions/sets.

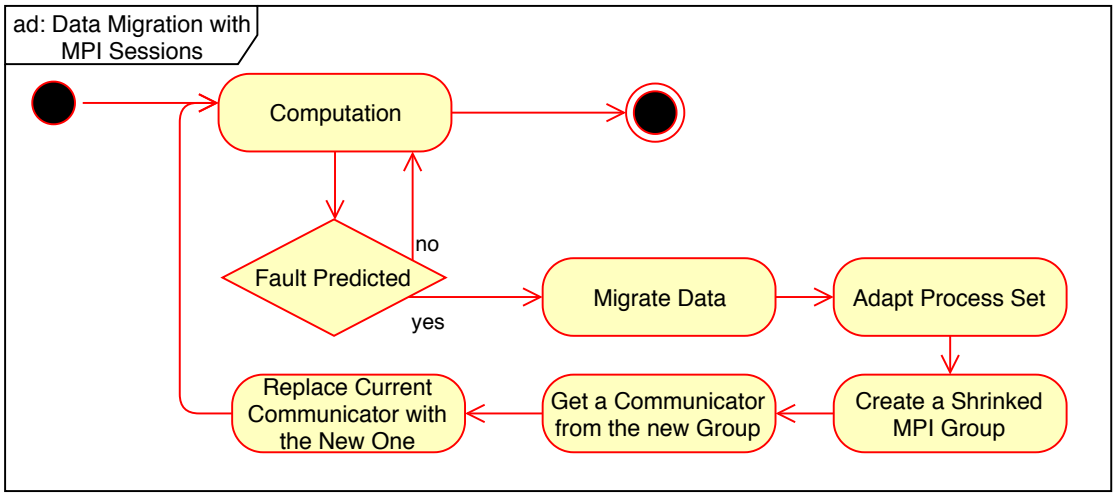

Figure 7.2.: Activity Diagram for Data Migration with MPI Sessions

7.2. Extension Proposal for MPI Sessions: MPI Process Sets

Allowing dynamic data migration in MPI sessions, we have created a comprehensive design for the MPI process sets within the context of MPI sessions. The main responsibility of the MPI process sets module is to relay information from the runtime system to both the MPI library and the application. This way, both the MPI library and the application can utilize this information and react to any *change*, such as a failure or revocation of resource. Figure 7.3 illustrates the required interfaces in our design to achieve these functionalities. The most important interfaces are the *PMIx* and *Key-value Store* interfaces in Figure 7.3, which are responsible for communication with the runtime system. In the remainder of this section, we will present all the detailed design considerations for our MPI process sets prototype. In addition to this dissertation, some basic concepts and detailed discussions on design considerations have been published by Anilkumar Suma [Sum19].

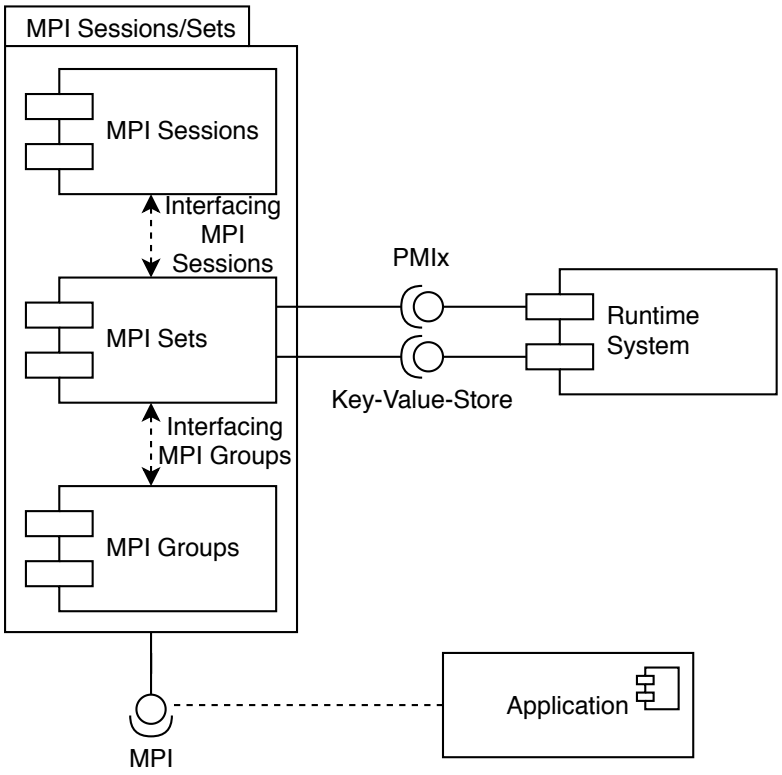

Figure 7.3.: Interfaces for MPI Process Sets

7.2.1. Components in Our MPI Sessions / MPI Process Sets Design

To illustrate the responsibility of the different components in our MPI Sessions and MPI process sets design, we have summarized all the components as illustrated in Figure 7.3 and their respective interfaces in Table 7.1. Note that although the interface to the runtime system is a crucial part of our design, we do not provide an implementation in our prototype because our prototype is designed to demonstrate the possibility of achieving data migration using the MPI sessions concept.

The modular and layered design of Our MPI sessions concept enables the possibility of exchanging key components with other implementations. For example, Anilkumar Suma [Sum19] has provided an implementation of our sessions design using the *FLUX* [Ahn+14] runtime system.

Table 7.1.: Overview of Components and Their Responsibilities within Our MPI Sessions / MPI Process Set Design

Component	Responsibility	Application Programming Interface	Required Interfaces
MPI Sessions / MPI Process Sets Library	Providing the dynamic process set management, facilitate and delivering the additional failure prediction as proposed by the MPI forum and in this work.	MPI sessions interface, which is the proposal for standardization and used as an extension to the existing MPI implementation.	MPI, Key-Value Store, Runtime System (not in our prototype)
MPI Library	Providing the functionality of the current MPI standard. In this work, MPI standard version 3 is used.	MPI (`mpi.h`), the MPI Library.	As specified in the MPI standard.
Key-value Store	Providing the required distributed storage space for MPI sessions. Maintaining consistency of information stored by our MPI Sessions Library.	Key-Value Store Interface, which provides basic getter and setter functionalities to string-based key-value pairs.	General: none. Our Prototype: Linux Shared Memory
Runtime System *(not in our prototype)*	Providing real-time and stateful information on the runtime system, such as process states, locations, topology, etc.	Suggested: PMI [Bal+10], PMIx [Cas+17]	As specified by PMI/PMIx

7.2.2. Semantics of MPI Process Sets

To understand our reasons for designing the MPI process sets, we first introduce the definition of an MPI (process) set. In our prototype, we adhere to the proposal by the MPI Forum [Mes18], in which each set is identified by a Uniform Resource Identifier (URI), starting with the identifier `mpi://` or `app://`. Examples of MPI set names are listed in Figure 7.4.

The semantics of our MPI process sets are defined in accordance with the following rules:

- An MPI set can be created either by the runtime system (noted as `mpi://`, cf. Figure 7.4(1)–(8)), or by the user application itself (noted as `app://`, cf. Figure 7.4(9)–(10)).
- An MPI set can be hierarchical, i.e., in order to represent different *failure domains*. A failure domain is an organizational entity within a cluster, such as rack, chassis, and blade. A more coarse-grain failure domain can be further divided into multiple fine-grain failure domains, which are denoted as *path* in the URI of MPI process sets. An example of different failure domains is illustrated in Figure 7.5 and the corresponding set names are given in Figure 7.4(5)–(8).
- An MPI set is *immutable*. This means that it cannot be changed nor be deleted once it has been created, as long as at least one active MPI session is still bounded to the given set.
- A change to any given, existing process set is achieved by issuing a new version of the process set. A simple integer number at the end of the *path* in the URI for a given set is used to reference a specific version of this process set (cf. Figure 7.4(2)–(3), (6), and (8)). An older version of a process set remains valid until no reference exists for that specific version. Moreover, an older version of the process set is called "inactive", and the most recent version of the process set is called "active".
- If the URI for a given set is used without its version number (e.g., Figure 7.4(1)), the currently active process set is referenced.
- Our design for the MPI process set features a *semi-global* accessibility. Each process can only see and access that particular process set in which it is a member of. We believe that this design provides the best usability while preserving scalability. Figure 7.6 illustrates this design.

```
(1) mpi://world
(2) mpi://world/1
(3) mpi://world/2
(4) mpi://self
(5) mpi://island1/rack1/
(6) mpi://island1/rack1/1
(7) mpi://island1/rack1/chassis3/blade0
(8) mpi://island1/rack1/chassis3/blade0/1
(9) app://mlem
(10) app://lulesh
```

Figure 7.4.: Examples for MPI Set Names

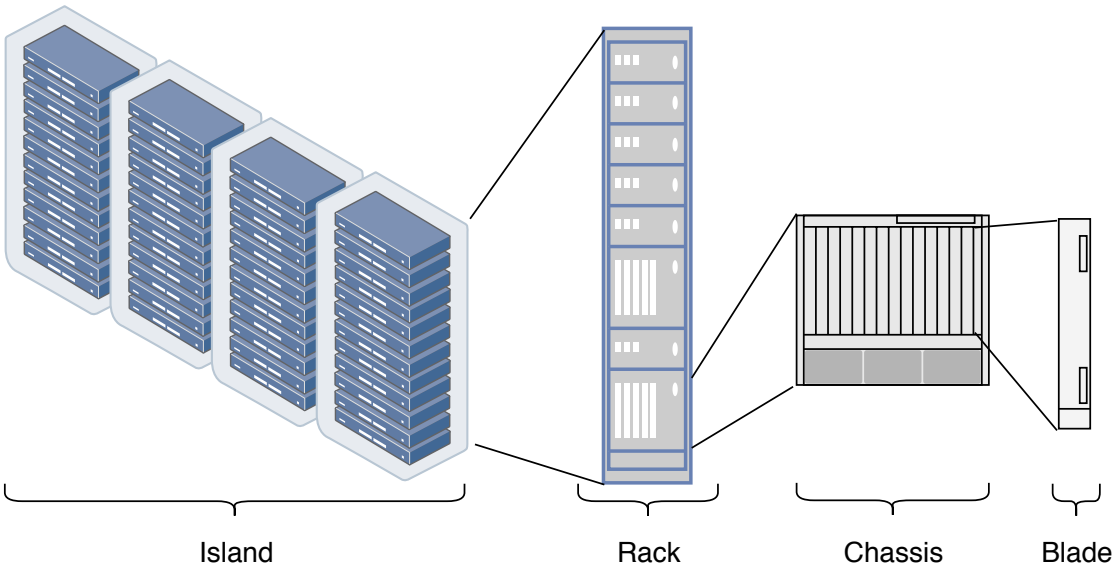

Figure 7.5.: Illustrations of Different Failure Domains

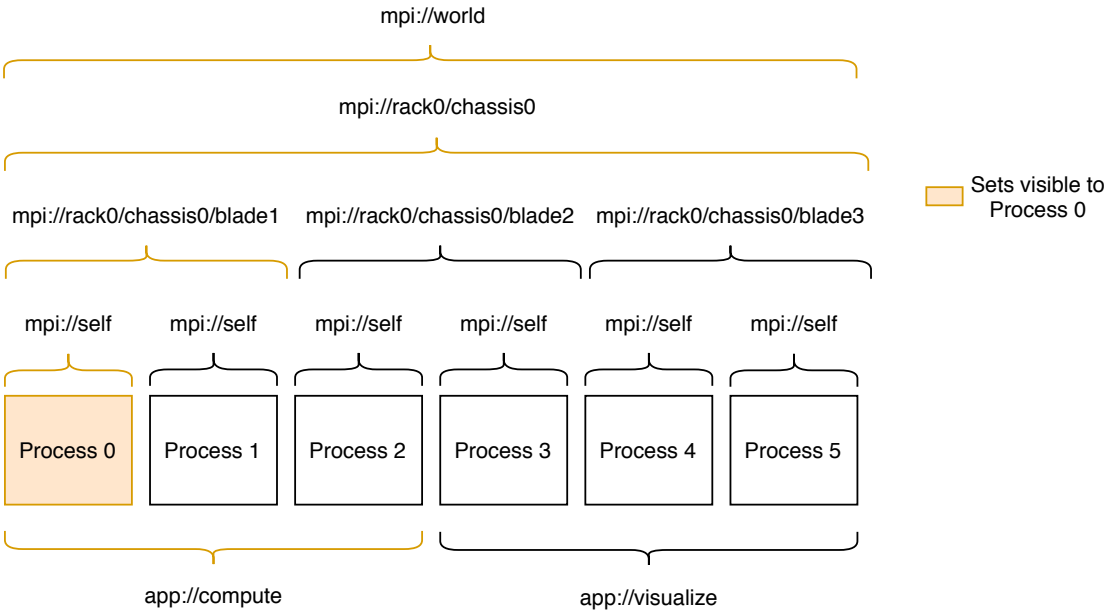

Figure 7.6.: Illustration of the *Semi-global* Accessibility of MPI Process Sets

7.2.3. Storage of Process Set Information

The semantics of the MPI process sets left one question unanswered: What is responsible for storing the set information because it needs to be consistent across all the MPI processes? The short answer is: *the runtime system*.

With our design, two different types of process sets are introduced: Architectural process sets, which are process sets provided by the MPI library or the runtime system according to the architecture or topology of the cluster system; and application process sets, which are provided by the user application. Since the architectural process sets are expected to be greater in number due to the different granularity, we believe it is best practice to transfer the process set related information to the responsibility of the runtime system. This way, the identifiers and handles to the architectural process sets can be generated dynamically on demand, thus reducing the impact on scalability. This means that all the application-related process sets have to also be stored in the runtime system, potentially using a callback API provided by the runtime system or through a distributed key-value store. Such ambitions also cope with the recent development in interface standards for runtime systems, such as the *PMIx* [Cas+17], the proposed extension of the well-known PMI standard [Bal+10].

To distinguish the MPI processes, a simple new scheme has been introduced to uniquely identify the processes. Currently, the identification of the MPI processes can be

simply done by calling the *MPI_Comm_rank()* function with the default `MPI_COMM_WORLD` communicator. However, without the world communicator, the MPI process sets need to store a mapping between the physical process instances and the abstract process sets. For our prototype, this process identifier is generated automatically by calling the `MPI_Get_processor_name`, combined with the process ID from the Operating System (OS). Here is an example: An MPI process running on the machine *node1.mysupercluster.de* with the process ID *1024* bears the globally unique identifier as shown below.

```
node1.mysupercluster.de_1024
```
(7.1)

7.2.4. Change Management of the MPI Process Set

To achieve best scalability, we provide an asynchronous design for change management in MPI process sets. The sequence diagram in Figure 7.7 illustrates this design, which can be summarized as follows:

- If an application wants to react to changes to a process set, it issues a *watch* on that process set by calling `MPI_Session_iwatch_pset()`. An `MPI_Info` object is created as a handle to the watch issued by the application.
- If a change occurs from outside (or is explicitly triggered by the application), this information is queued by the MPI runtime, which also creates a new version of the set asynchronously. This way, the latency is hidden in the MPI library, which reacts to the change event.
- At a later stage of program execution, the application can query the status of the watch by calling `MPI_Session_check_psetupdate()` with the previously created `MPI_Info` object. As the change event has already been processed by the MPI runtime, this call returns almost immediately.
- The application is now aware of this change and can obtain the latest version of the process set, perform data migration and continue its operation.

Besides reacting to changes which come from the runtime system (such as faults), our design also enables the application to delete (or add) process from (to) a process set. For this purpose, we have introduced the calls `MPI_Session_deletefrom_pset` and `MPI_Session_addto_pset`. This use case is especially interesting, if an application needs to switch to a smaller configuration with fewer processes, i.e., after the prediction of a failure.

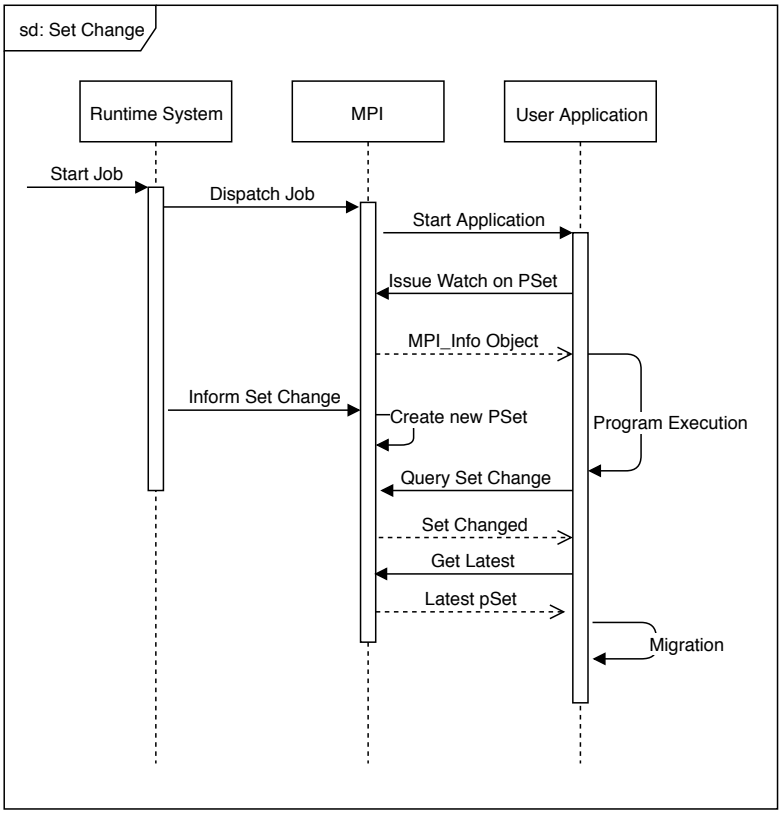

Figure 7.7.: Sequence Diagram of MPI Process Set Change Management

7.2.5. Implementation of Our MPI Process Set Module

For demonstration and evaluation purposes, we have implemented a standalone static library which implements the functionality of MPI sessions and process sets (in the following: MPI sessions library). The main reasons for this decision is the fact, that mainstream open source MPI distributions (OpenMPI, MPICH, LAM/MPI, and MVAPICH2) are hard to modify because they lack modularity and are tuned for performance. Since our aim is to demonstrate and evaluate our concept, the adaptation of an existing MPI library is not required. For this reason, our MPI sessions library is designed to work alongside any existing MPI libraries. For the development, we have chosen OpenMPI version 3.1 as a reference. The resulting architecture of our library, the OpenMPI library, and the application are illustrated in Figure 7.8.

In Figure 7.8, we can see that besides the sessions library, the required Key-value Store has also been implemented by us. This is required in order to store and distribute information related to process sets (cf. Section 7.2.3). In our prototype implementation,

we use Linux shared memory to implement this key-value store. This leads, however, to a limitation on our current prototype, which only supports shared memory systems.

Our prototype features three different type of interfaces:

- *MPI sessions interfaces* are the new calls proposed by the MPI Forum [Mes18] and us. They provide the new functionalities introduced by our proposal for the application. Detailed documentation of all the calls in this interface is provided in Appendix E.1.
- *Internal MPI_S interfaces* are connections from our library to a standard MPI library in order to provide the required MPI functionalities, which are used in our prototype. Except the `MPI_Comm_spawn`, all the MPI_S interface calls are standard MPI functions provided by the underlying MPI library *without* any modification.
- *Key-value Store interfaces* are a collection of function calls that are used to mimic the runtime system with a key-value store. This way, we simulate the interaction with the runtime system. Detailed documentation of all the calls in this interface is provided in Appendix E.2.

As mentioned above, except for `MPI_Comm_spawn`, our library does not replicate nor override any existing MPI function, therefore, it is fully compatible with any existing application without modification.

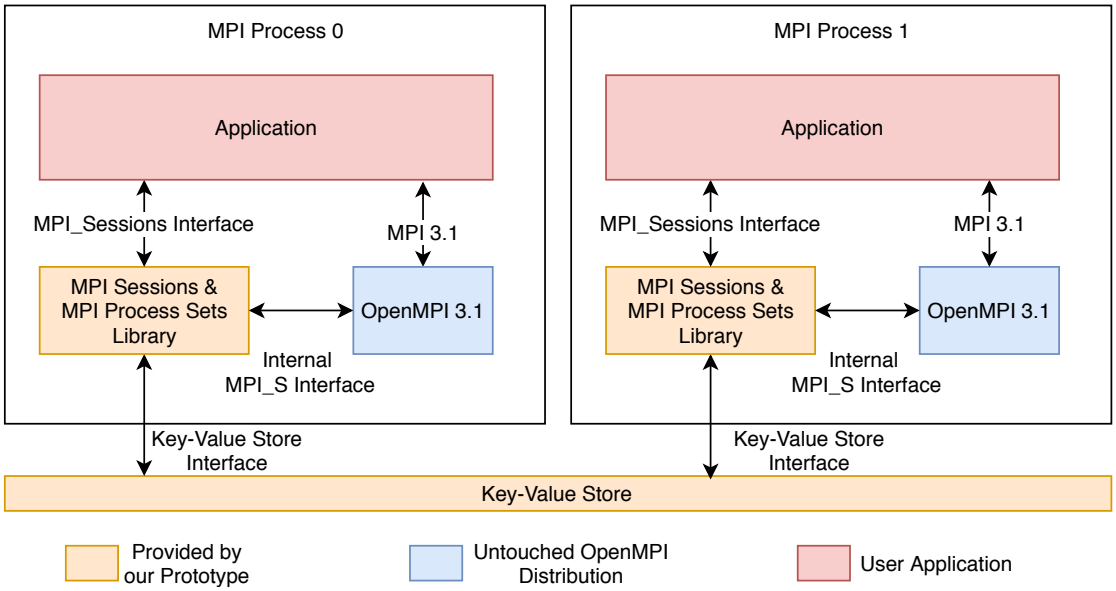

Figure 7.8.: Schematic View of Our MPI Sessions / MPI Process Set Library

7.2.6. Known Limitations of Our MPI Sessions / MPI Process Set Prototype

The goal of our prototype is to demonstrate the feasibility of achieving fault tolerance with the MPI sessions / MPI process sets concept. With sufficient progress, we can drive the related discussions into the MPI Forum and help push MPI sessions and MPI process sets into the standardization process. Consequently, we will not implement all the proposed MPI sessions function calls that have been suggested by the MPI Forum as listed in Appendix D. Instead, only a subset of functions has been implemented in order to provide the minimum required set of functionalities on top of the standard MPI 3.

Moreover, our prototype works only on a shared memory system, due to the limitation of the key-value store that we have implemented based on shared memory. This is a major limitation in functionality of our prototype. Previous attempts have been made to utilize 3rd-party key-value stores to support distributed memory. Our MPI sessions prototype has also been adapted to support the key-value store of *FLUX* [Ahn+14] by Anilkumar Suma [Sum19], which is a proposed next-generation resource manager. However, the immature nature of FLUX makes it hard to deploy for evaluation. For this reason, we have selected our shared memory key-value store for evaluation purposes.

The specification of process sets in which the application is running, is currently done by a configuration file. Our library provides support for process sets by simulating process sets by using a subset of MPI processes started by `mpirun`. Such subsets can be specified by the user by composing a configuration file with pairs of process-set names and process-rank ranges. An example is illustrated in Figure 7.9. The process set *app1*, which is used by the user application is specified with a configuration file containing the content as shown in below.

$$app1\ 0-2 \tag{7.2}$$

Finally, our current prototype has not yet been tuned for performance. Our current design heavily relies heavily on the key-value store to provide the required data and state consistency across all the MPI process instances. However, we have chosen to use *serial consistency* [Mos93] as a simple but effective consistency model. Unfortunately, this model also introduces additional serialization overhead when accessing the key-value store. Our simple implementation of the key-value store has not been optimized yet to provide best performance for this prototype.

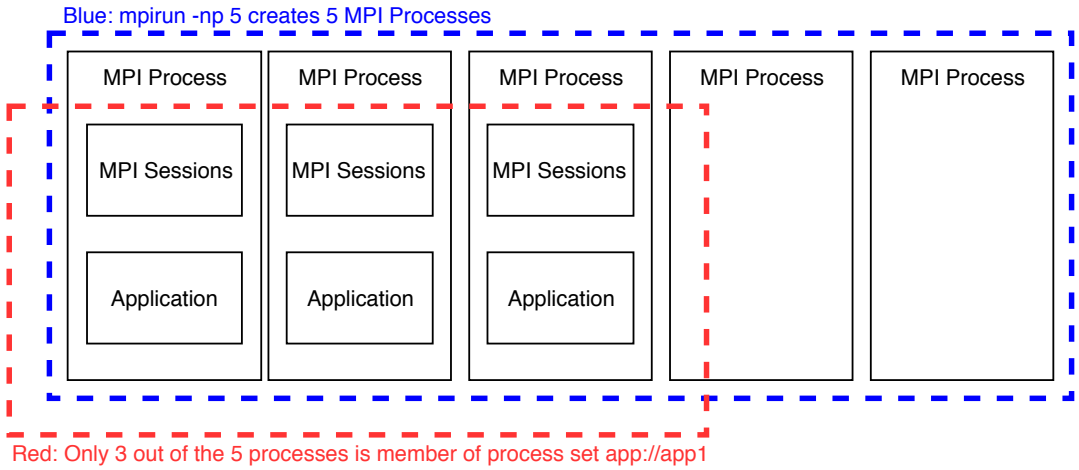

Figure 7.9.: Simulated MPI Process Set

Our prototype implementation used for evaluation is accessible as open source software on *Github*[2]. In the next section, application examples for achieving data migration with MPI sessions and MPI process sets are given and discussed. For the sake of simplicity, *MPI sessions* is used to reference the prototype implementation provided by us in the remainder of this section.

7.3. Evaluation of MPI Sessions and MPI Process Sets

7.3.1. Basic Example of an MPI Sessions Program

Before we get started with our evaluation of MPI sessions, let us implement the same example application in MPI sessions as before (cf. Section 6.2), i.e., the simple *vsum* example application. Recap that *vsum* is an application which calculates the sum of a given vector A, stored as a simple C array. For simplicity's sake, we assume that the length of the vector is divisible by the number of processes. This MPI implementation of a simple application is illustrated as pseudocode in Algorithm 9.

The primitive transition of the application using our MPI sessions is trivial because we provide backward compatibility. No modification is required at all. To enable the MPI session, simply replace `MPI_Init` with `MPI_Session_Init` and `MPI_Finalize` with `MPI_Session_Finalize`.

[2] `https://github.com/caps-tum/MPI-Sessions-Sets`, accessed in July 2019

Algorithm 9: MPI Implementation of Example Application *vsum*

A : The Input Vector
y : Result of Vector Sum

```
MPI_Init(argc, argv);
n ← A.length() / num_processes; myStart ← n*my_rank; myEnd ← myStart+n;
for i ← myStart to myEnd do
    y += A[i];
end
MPI_Comm_size(&size, MPI_COMM_WORLD);
MPI_Allreduce(SUM, y, size, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Finalize();
```

Since the `MPI_COMM_WORLD` does not exist anymore within the context of MPI sessions, a new *world* communicator has to be created. This can be done by calling the respective calls as provided in Figure 7.10. Instead of the `MPI_COMM_WORLD` communicator, the newly created *world* communicator `Comm_world` must be used.

To achieve fault tolerance with data migration, a simple asynchronous process set watch can be issued using `MPI_MPI_Session_iwatch_ps`. This way, potential changes in the currently active process set (`mpi://world`) can be detected by calling `MPI_Session_check_psupdate`. After detecting the change, a new *world* communicator can be created by calling the function sequence as indicated in Figure 7.10 again. However, unlike with LAIK, the application is responsible for preserving data that are after this change.

```
MPI_Info setInfo;
MPI_Group gWorld;
MPI_Comm Comm_world;
MPI_Session_get_setinfo(&session, "mpi://world", &setInfo);
MPI_Group_create_from_session(session, "mpi://world",
&gWorld, setInfo);
MPI_Comm_create_from_group(gWorld, NULL, &Comm_world, setInfo);
```

Figure 7.10.: Creating of the *World* Communicator with MPI Sessions

7.3.2. Application Example 1: The Maximum Likelihood Expectation Maximization (MLEM) Algorithm

In this chapter, we introduce the required changes for porting the *Maximum-Likelihood Expectation-Maximization (MLEM)* algorithm to MPI sessions.

Recap that MLEM is the algorithm used to reconstruct images from a medical PET scanner (cf. Section 6.3.1). The main kernel of MLEM consists of four major steps: forward projection, correlation, backward projection, and update. The forward and backward projection are the major computationally-intensive sub-kernels and both calculate Sparse Matrix-Vector (SpMV) multiplication. The input data are the system matrix stored in CSR format, the list-mode sinogram and the total number of iterations. A detailed description of the MLEM algorithm is presented in Section 6.3.1.

Porting MLEM to MPI Sessions

The porting process for MLEM in order to support MPI Sessions is trivial. Like many classic MPI applications, MLEM operates on the `MPI_COMM_WORLD` communicator. This has to be replaced by a communicator derived from an MPI set. For this, we have created a process set with the name `app://mlem`. Similar to the trivial example given in Section 7.3.1, a communicator is created using this set by calling the functions as indicated in Figure 7.11. Followed by a search and replace, all the occurrences of `MPI_COMM_WORLD` are replaced by the global variable `mpi.current_comm`, which relies on the process set `app://mlem`. The `MPI_finalize` is also replaced by `MPI_Session_finalize` accordingly.

```
MPI_Session_init(&mpi.session);
Info ps_info;
Session_get_set_info(&mpi.session, "app://mlem", &ps_info);
Group_create_from_session(&mpi.session, "app://mlem",
                          &mpi.current_group, ps_info);
Comm_create_from_group(mpi.current_group, NULL ,&mpi.current_comm, ps_info);
```

Figure 7.11.: Creating the *MLEM* Communicator with MPI Sessions

After this modification, the MPI sessions version of *MLEM* is already fully operational. No additional changes are required because our MPI sessions implementation ensures full compatibility with existing MPIs. We have not changed any data structure, communication patterns, or compute kernels. Consequently, no performance impact is imposed by our implementation as per design.

Enabling Fault Tolerance by Using Data Migration

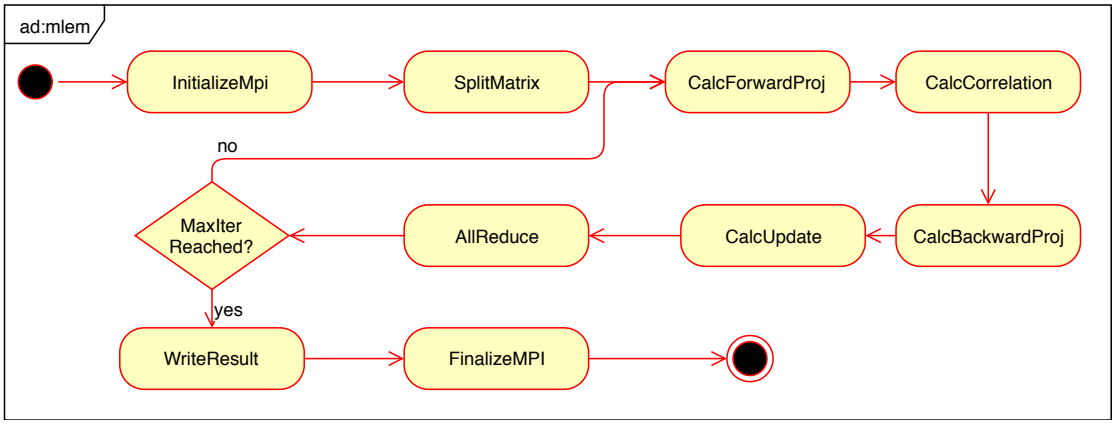

Figure 7.12.: Activity Diagram of Original MPI-based MLEM

The original MPI-based MLEM application can be summarized in the activity diagram as shown in Figure 7.12. In this iterative algorithm, the four basic kernels (`CalcForwardProj`, `CalcCorrelation`, `CalcBackwardProj`, and `CalcUpdate`) and one communication step (`AllReduce`) are executed in each iteration. This means that the current result increment – the image update – is communicated across every participating process at the end of each iteration. Therefore, no data redistribution is required for fault tolerance with data migration.

Furthermore, the system matrix does not require a migration either, albeit it is partitioned at the beginning of the application. This is because the `SplitMatrix` function only returns an abstract range of rows to be processed by each MPI process instance, and the Matrix itself is only mapped into memory using `mmap` [Joy+83]. The data of the matrix is only cached in main memory. Consequently, data can be remapped after repartitioning upon a fault. No communication of the actual matrix is required.

To obtain information on changes in an MPI set, `MPI_Session_iwatch_pset` is called to issue a watch on the current process set, as part of the modified `InitializeMpi` routine. This way, any process set changes can be determined at a later stage by calling `MPI_Session_check_psetupdate`. If a change is detected, a new group and communicator can be created from the latest version of the process set. As mentioned before, no communication or data redistribution is required at all. Repartitioning is achieved by calling the `SplitMatrix` function. Figure 7.13 illustrates the adapted application with fault tolerance capability. The above mentioned changes are highlighted in blue.

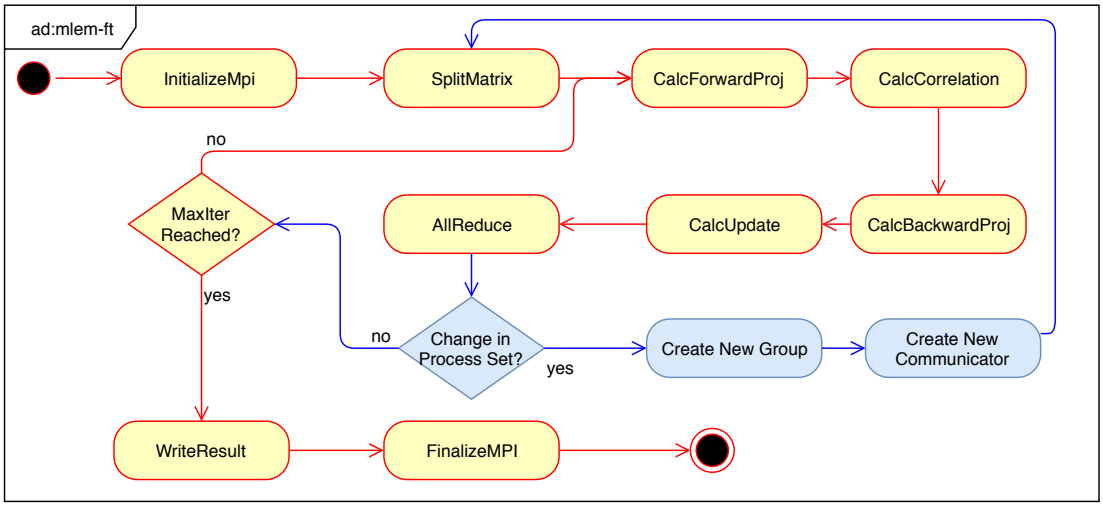

Figure 7.13.: Activity Diagram of MPI Session-based MLEM with Fault Tolerance

Evaluation

Our MPI session-based MLEM implementation is provided as open source software on *Github*[3]. We have tested the MLEM application for correctness and effectiveness by shrinking the process set of MLEM, which worked as expected.

Although our MPI sessions prototype is not designed for performance evaluation, we have run a small scale test on a single node server. The results show that no overhead at all is introduced by using our prototype for the MPI session-based MLEM implementation.

Porting of MLEM has taken approximately 2 hours for a programmer, who is experienced with MPI sessions, but not experienced with MLEM, and so had some minor assistance from the MLEM developers.

7.3.3. Application Example 2: The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)

As in the evaluations in LAIK (cf. Section 6.3.5), LULESH is the second application we have evaluated for our MPI sessions prototype. Recapping, the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is a benchmark targeting the solution to the SBP [Sed46] by using Lagrangian hydrodynamics [HKG], which represents a class of a classic HPC application. It features two types of data structures,

[3] `https://github.com/envelope-project/mlem`, accessed in June 2019

the *elemental* data and *nodal* data, which are accessed in a *halo* and *overlapping* manner, respectively. Moreover, these data are used by compute kernels, which require neighborhood communication. The iterative solver terminates either based on a predefined convergence goal, or after a specified number of iterations. Details of the LULESH application are given in Section 6.3.4.

Porting LULESH to MPI Sessions

Unlike MLEM, LULESH is a very complex application, which sets up its communication routines, targets, and buffers at the beginning within the `initMeshDecomp` call. Furthermore, as an application tuned for performance, all the data structures are stored in a struct of arrays, whose size is not designed to be modified. In order to achieve fault tolerance by using data migration, two major porting steps are required: Transform existing MPI communication to MPI communication with MPI sessions, and then change data structures to allow data migration. Unlike with LAIK, no new partitioners are required as we can use the existing partitioner provided from the reference LULESH implementation. Detailed documentation of the changes made to the LULESH code is given below.

1. **Enabling MPI Sessions by Eliminating Trivial MPI Objects in LULESH.**

   The rather simple first step is to create a communicator from the application's process set (named `app://lulesh`). This step is done by replacing the `MPI_Init` with the same sequence of MPI sessions calls as in the case of MLEM.

   This communicator is stored as a global variable for the sake of simplicity. With a search-and-replace function, we replace all access to the trivial `MPI_COMM_WORLD` communicator with access to the communicator that we have just created. The replacement of `MPI_Finalize` using `MPI_Session_finalize` concludes the first step.

   With this modification, we have created a semantically identical implementation of LULESH using the MPI sessions library. We have also tested this intermediate implementation for its correctness.

2. **Adapting All the Data Structures.** To achieve fault tolerance, the data containers held in each *Domain* (cf. Section 6.3.4) have to be elastic. Unlike porting to LAIK, there is no difference between data structures that require communication and those that do not. For simplicity, we replace all the array structures in the Domain class with `std::vectors`. This way, we utilize the build-in `resize` feature of the `std::vectors` to reallocate memory space for data containers.

3. **Creating Mapping Function for Local and Global View of Domain.**

   For the sake of simplicity and better understanding, we have selected the *global repartitioning* method (cf. Section 5.2).

   Two mandatory functions for data repartitioning are created to calculate the new mesh decomposition and data location. Figure 7.14 illustrates these two functions.

   The `pack` function takes the current mesh decomposition and calculates the indexing of the respective data structure in a global view within the whole mesh. It then copies the data from the local domain into a vector that holds data from the whole mesh for further processing. For entries in the elements (or nodes) data structure that are not owned by current process, `MAX_DOUBLE` is used for their initialization.

   The `unpack` function is the exact opposite of the `pack` function, which takes a data structure in global view and copies the relevant data for my process into a local domain object.

4. **Creating Function for Data Migration and Replacement of Communicator.**

   To enable fault tolerance, we have added data migration routines into the main loop of LULESH. Since we aim for demonstration purposes and not for best performance, our data migration is achieved by calling `MPI_Allreduce` using the `MPI_MIN` reduction on the packed global vector. All data structures given in Appendix C have to be communicated and transferred.

   After the data migration, we have created a new MPI group and communicator from the latest version of the MPI process set. Similarly as in the first step, this new communicator replaces the existing communicator created in step one. The `InitMeshDecomposition` function is then called again to create new local domains for further processing.

   Finally, the globally distributed data is unpacked into the new local domains using the `unpack` function created in the previous step. All the temporary global vectors are then freed as they are no longer required.

5. **Enabling Fault Tolerance by Adding Watch to the MPI Process Set.**

   The last step in our porting effort is to add a process set watch at MPI session initialization time by calling `MPI_Session_iwatch_ps`. Similar to the MLEM example, the status of this watch is queried after each iteration. Unlike MLEM, LULESH does not perform global synchronization after each iteration by calling `MPI_Session_check_psupdate`. For this reason, we have inserted an `MPI_Barrier` to ensure synchronization before the check for process set change. Any processes

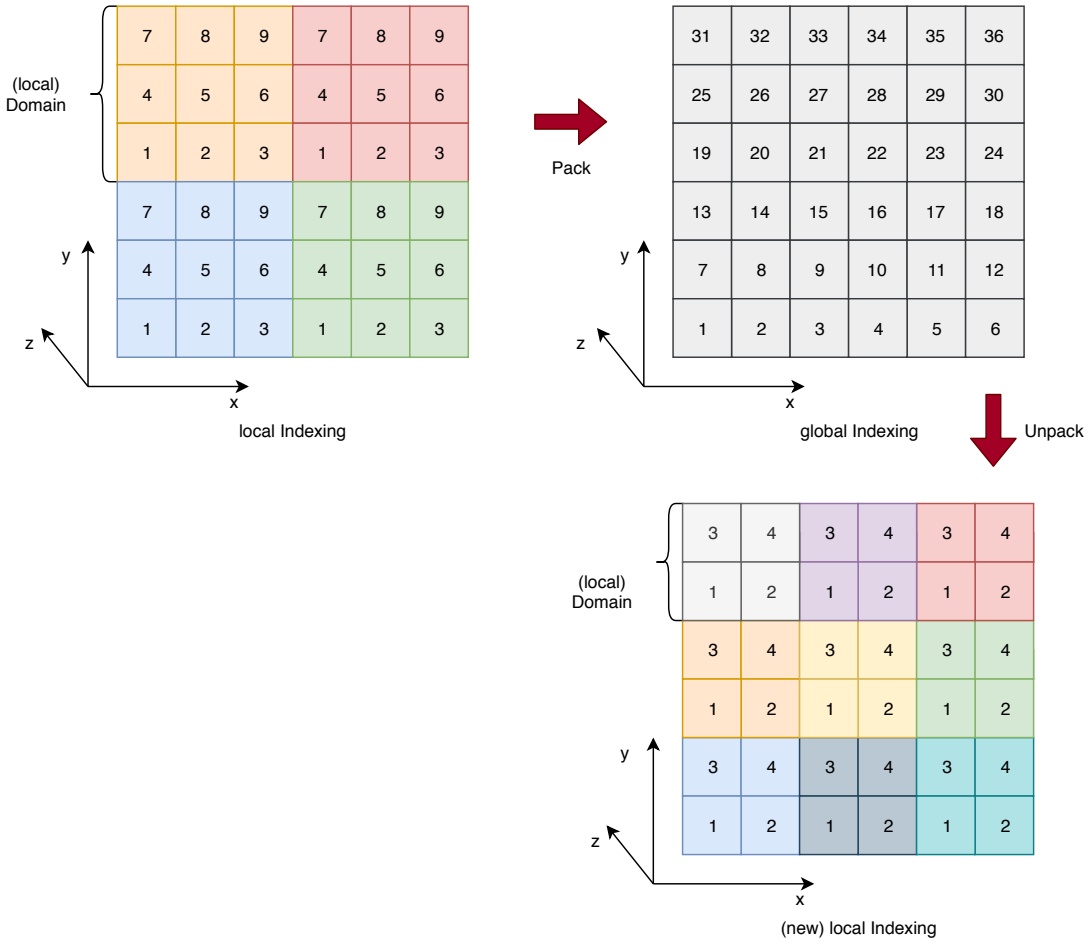

Figure 7.14.: Schematics of Local Numbering vs. Global Numbering for LULESH Elements

that are no longer part of the process set call `MPI_Session_finalize` and go to idle.

The implementation of our adapted LULESH is available as open source software on *Github*[4]. The porting effort is approximately ten hours for two programmers: one with in-depth knowledge of MPI sessions, and the other with in-depth knowledge of LULESH. It is important to mention that similar to the LAIK-based LULESH implementation, we do not change the precondition imposed by LULESH of requiring a cubic number of processes for execution.

Evaluation

We conducted a series of experiments similar to the *laik lulesh* as described in Section 6.3.5. In all experiments, we compared the reference LULESH code provided by LLNL on their *Github*[5] (in the following, *reference lulesh*), our *laik lulesh* code as introduced in Section 6.3.4, and the MPI sessions based implementation (in the following, *sessions lulesh*).

As *sessions lulesh* only support shared memory systems, we selected the *CoolMUC-III* cluster system at Leibniz Rechenzentrum der Bayerischen Akademie der Wissenschaft (LRZ) for our experiments. Each node of the CoolMUC-III system features an Intel Xeon Phi 7210F processor with 64 physical cores. Each core provides up to four hyper-threads, providing a total of 256 logical processors on each node. A detailed description of the CoolMUC-III system is given in Appendix A.3. For our experiments, no specific configuration for the High Bandwidth Memory (HBM) on the Xeon Phi chip was selected. The default configuration is all-to-all associativity with HBM working in cache mode [Sod15].

All the binaries, including the MPI sessions library and LAIK library, were compiled with an Intel Compiler version 17. OpenMP was disabled as our prototype is not thread safe. All experiments were repeated ten times to minimize the impact of other disturbances such as OS and runtime system. The results from all the experiments are documented in Figures 7.15 – 7.19. Detailed results are given in the following.

Weak Scaling

Weak scaling experiments were conducted with the settings `-s 15 -i 1000` for all three applications (i.e., *sessions lulesh*, *laik lulesh*, and *reference lulesh*). These applications

[4] `https://github.com/caps-tum/mpi-sessions-lulesh`, accessed in June 2019

[5] `https://github.com/LLNL/LULESH`, accessed in June 2019

were run with $n = 1, 8, 27, 64, 125, 216$ MPI processes. The results for weak scaling are presented in Figure 7.15. The number of MPI processes is indicated on the horizontal axis. The normalized runtime per iteration in seconds is shown on the vertical axis. All the values noted in the charts represent the arithmetic mean of the ten independent runs.

As expected, all three applications performed equally well without hyperthreading. There is no significant difference or overhead in terms of runtime scaling behavior between these versions. This indicates a low overhead introduced by our sessions library. However, a very small overhead can be observed for *sessions lulesh* with an increasing number of processes. This is most likely due to the additional synchronization overhead introduced by the added `MPI_Barrier` in order to check the change status of MPI process set.

The results from the hyperthreaded part is not useful for performance analysis. However, the significant increase in runtime in the hyperthreaded experiments also supports our theory of synchronization overhead introduced by the `MPI_Barrier` because the all processes must synchronize at this point.

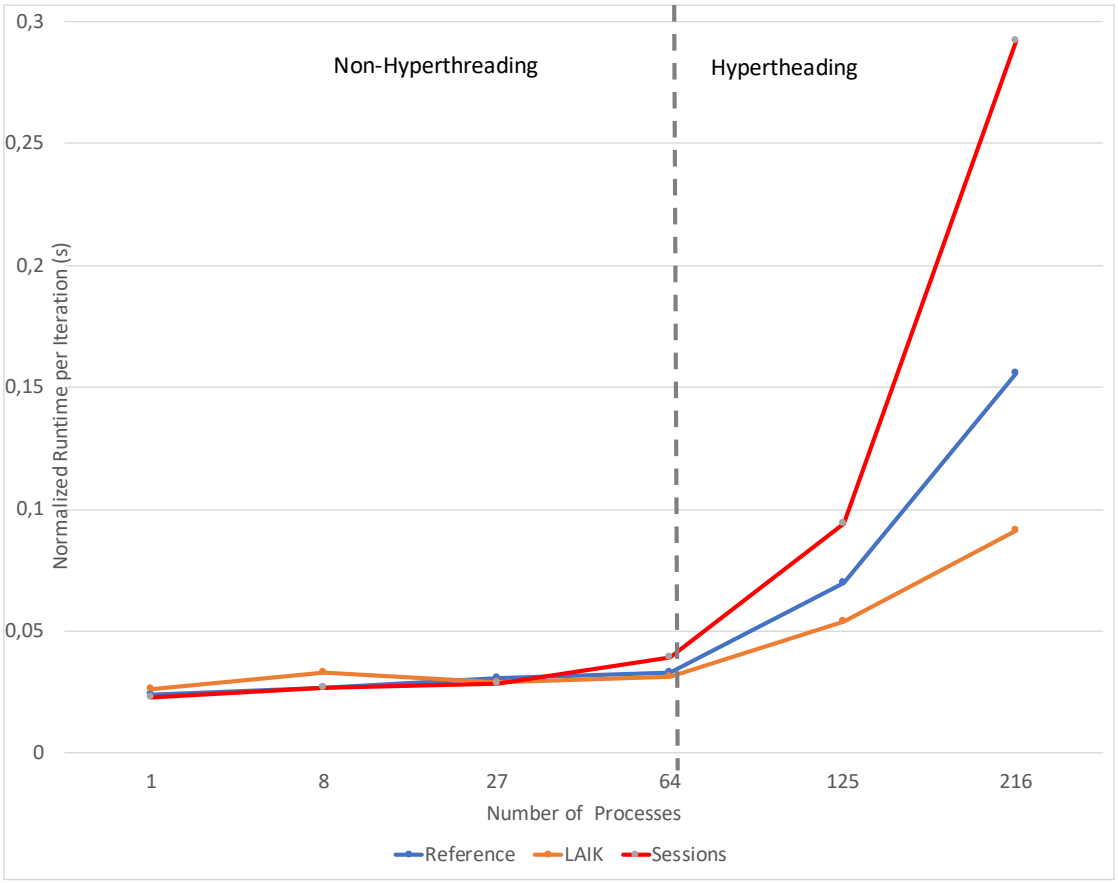

Figure 7.15.: Weak Scaling Comparison of *reference LULESH*, *sessions LULESH*, and *LAIK lulesh* on *CoolMUC-III*

Strong Scaling

Strong scaling experiments were conducted by fixing the global mesh size C (cf. Section 6.3.5). In this case, we have selected $C = 216000$. This resulted in the following configuration (n, s) for the experiment with $n = 1, 8, 27, 64, 125, 216$ processes:

$$\begin{aligned} (1^3 = 1, 60), \\ (2^3 = 8, 30), \\ (3^3 = 27, 20), \\ (4^3 = 64, 15), \\ (5^3 = 125, 12), \\ and (6^3 = 216, 10). \end{aligned} \tag{7.3}$$

We then fixed the maximum number of iterations (`-i`) at 1000. The results for strong scaling are provided in Figure 7.16, where the horizontal axis shows the number of processes, and the vertical axis shows the normalized runtime per iteration in seconds on a $log(3)$ scale. The corresponding speedup achieved for strong scaling is presented in Figure 7.17, where the vertical axis shows the achieved speedup. All the values shown on the charts represent the arithmetic mean of the ten independent runs.

Similar to weak scaling, no significant difference between *reference lulesh*, *laik lulesh*, and *sessions lulesh* is observable. The effect of the additional `MPI_Barrier` is also existent in the strong scaling experiments, most clearly seen on the speedup chart in Figure 7.17. All three applications show the same scaling tendency. Our MPI sessions library does not significantly change the performance of LULESH.

Remarkably, the *laik lulesh* even outperforms the reference implementation in some configurations. Our explanation for this phenomenon is an interference between the adapted data structure in LAIK (data is stored as "slices" of independent data regions rather than as a continuous block of data), and the high bandwidth memory used in cache mode. However, since performance engineering is not the goal of this dissertation, we have not conducted any further analysis of this phenomenon.

Again, the scaling curve for the hyperthreaded configurations confirms our theory that the overhead of *sessions lulesh* is mainly due to the additional `MPI_Barrier`.

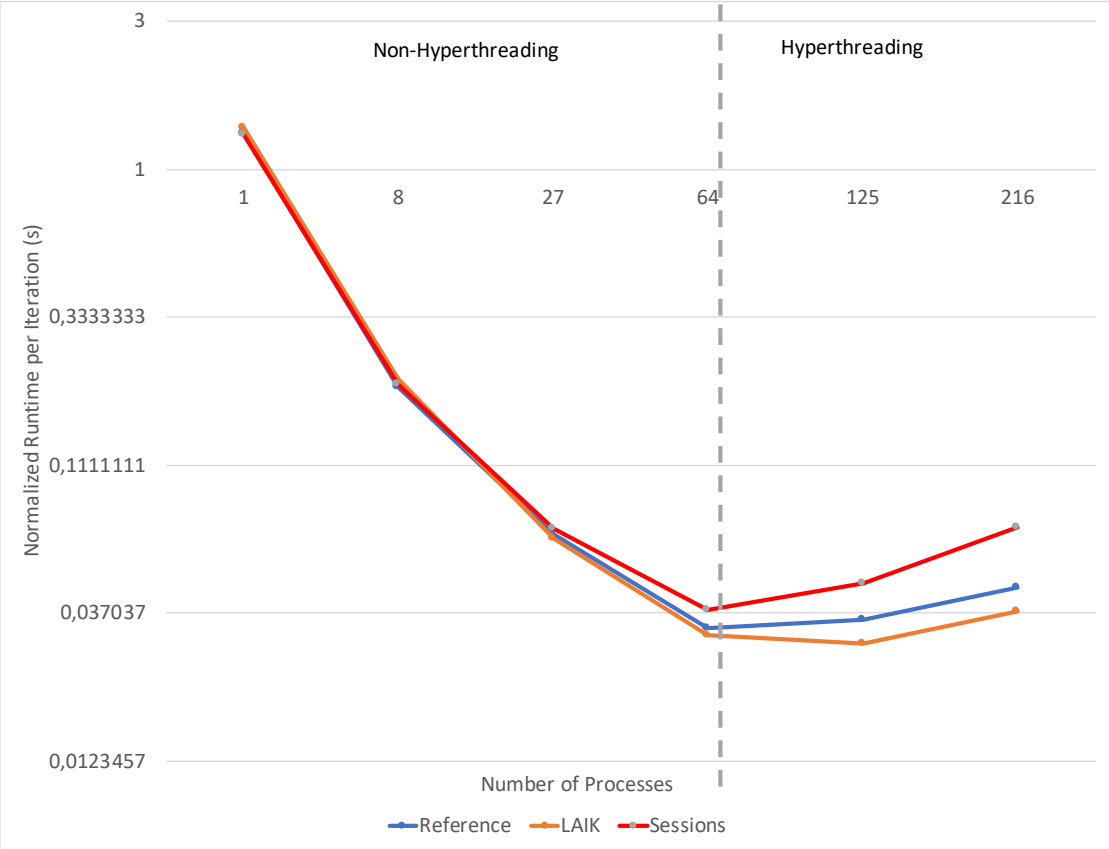

Figure 7.16.: Strong Scaling Comparison of *reference LULESH*, *sessions LULESH*, and *LAIK lulesh* on *CoolMUC-III*

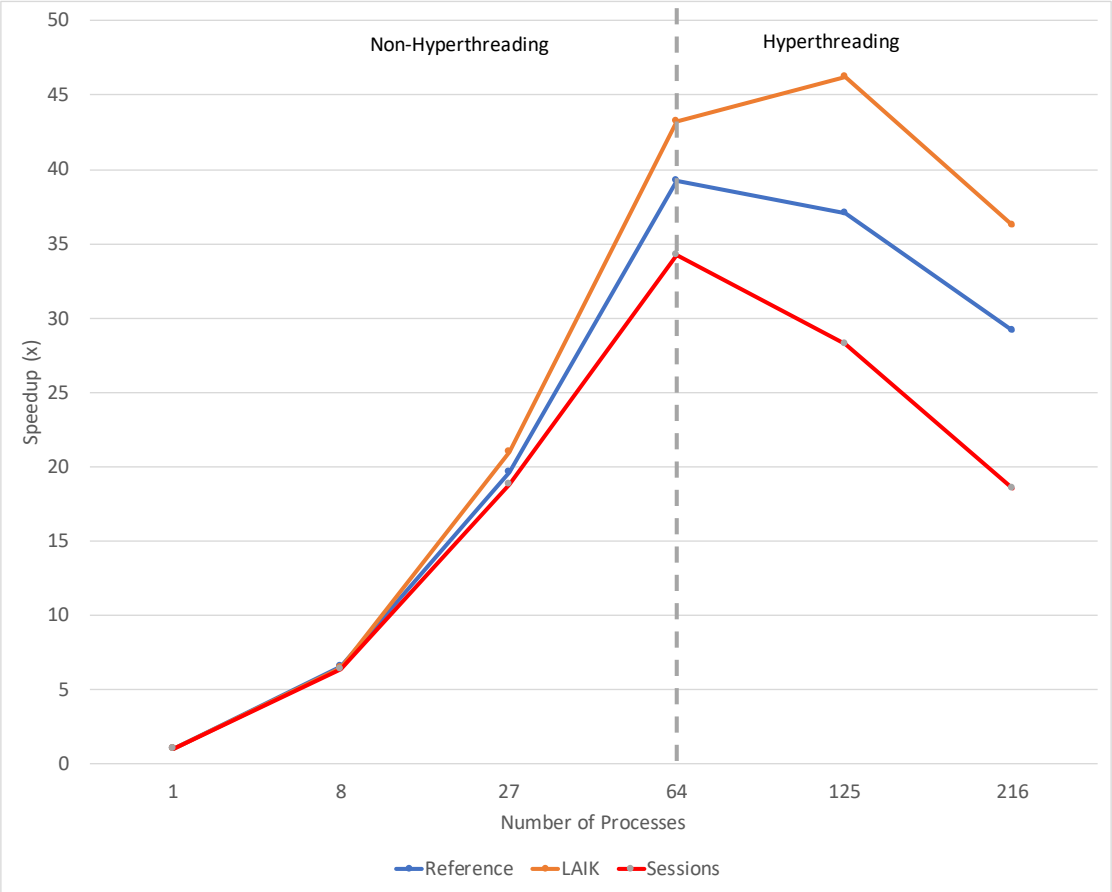

Figure 7.17.: Strong Scaling Speedups of *reference LULESH*, *sessions LULESH*, and *LAIK lulesh* on *CoolMUC-III*

Repartitioning and Fault Tolerance

The last experiment we did was intended to show the capability of achieving fault tolerance using repartitioning and data migration. A similar experiment as described in Section 6.3.5 was introduced here. Using the same setup as in the strong scaling test (cf. Listing 7.3), we migrated the the application to the next supported configuration with a smaller number of processes, i.e., from 64 processes to 27 processes ($4^3 \rightarrow 3^3$). We documented the normalized runtime per iteration before and after repartitioning for both *sessions lulesh* and *laik lulesh* for comparison. Ideally, the normalized runtime per iteration should be the same for an execution before and after repartitioning with the same number of processes. For example, for one test we started the experiment using 64 processes and then reduced it to 27 after 500 iterations. The normalized runtime for this configuration after repartitioning (27 processes) should be the same as for an experiment using 27 processes before repartitioning.

The results of our experiments are shown in Figure 7.18. The configuration used for a specific experiment is shown on the horizontal axis. The normalized runtime per iteration is shown on the vertical axis. All the values noted in the charts represent the arithmetic mean of the ten independent runs. Blue-colored bars indicate results from *laik lulesh*, and red-colored bars stand for results from *sessions lulesh*. The darker-colored bars denote normalized runtime per iteration *before* repartitioning, while light-colored bars denote normalized runtime per iteration *after* repartitioning.

From Figure 7.18, we can see the expected ideal behavior as described above. Each dark-colored bar in a given color (blue or red) matches the light-colored bar on the next configuration to the right of it. This means that both *sessions lulesh* and *laik lulesh* are equally efficient for data migration. The data migration is successful and does not change the runtime behavior of the application after the data migration. This is the best case scenario we aimed to achieve with our implementations.

As a final test, we evaluated the time required for executing repartitioning and the data migration processes. This result is presented in Figure 7.19. The configuration is depicted on the horizontal axis. The time needed for repartitioning is shown on the vertical axis. It can be clearly seen that the MPI sessions implementation outperforms the *laik lulesh*. This is because that during the sessions implementation we used *MPI_Allreduce* rather than complex point-to-point communication. For data migration, *laik lulesh* calculates a transition plan, which is then tuned for minimum communication demand. However, the communication overhead in a shared memory system is minimal, whereas overhead of the transition plan calculation becomes dominant. Our *sessions lulesh* benefits from the simple communication pattern, which leads to a smaller time requirement. Nevertheless, both approaches are fairly fast for data migration and are suitable for achieving fault tolerance with data migration.

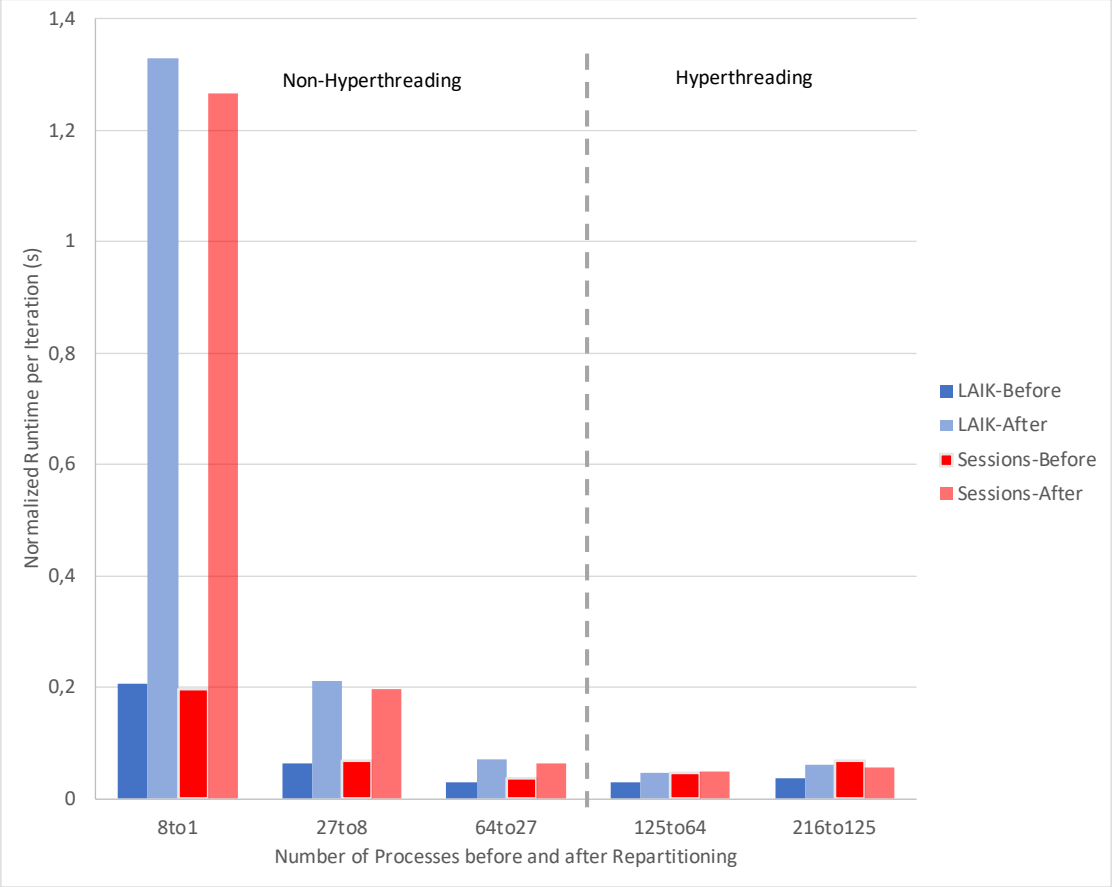

Figure 7.18.: Repartitioning Effect of *sessions lulesh* and *laik lulesh* on *CoolMUC-III*

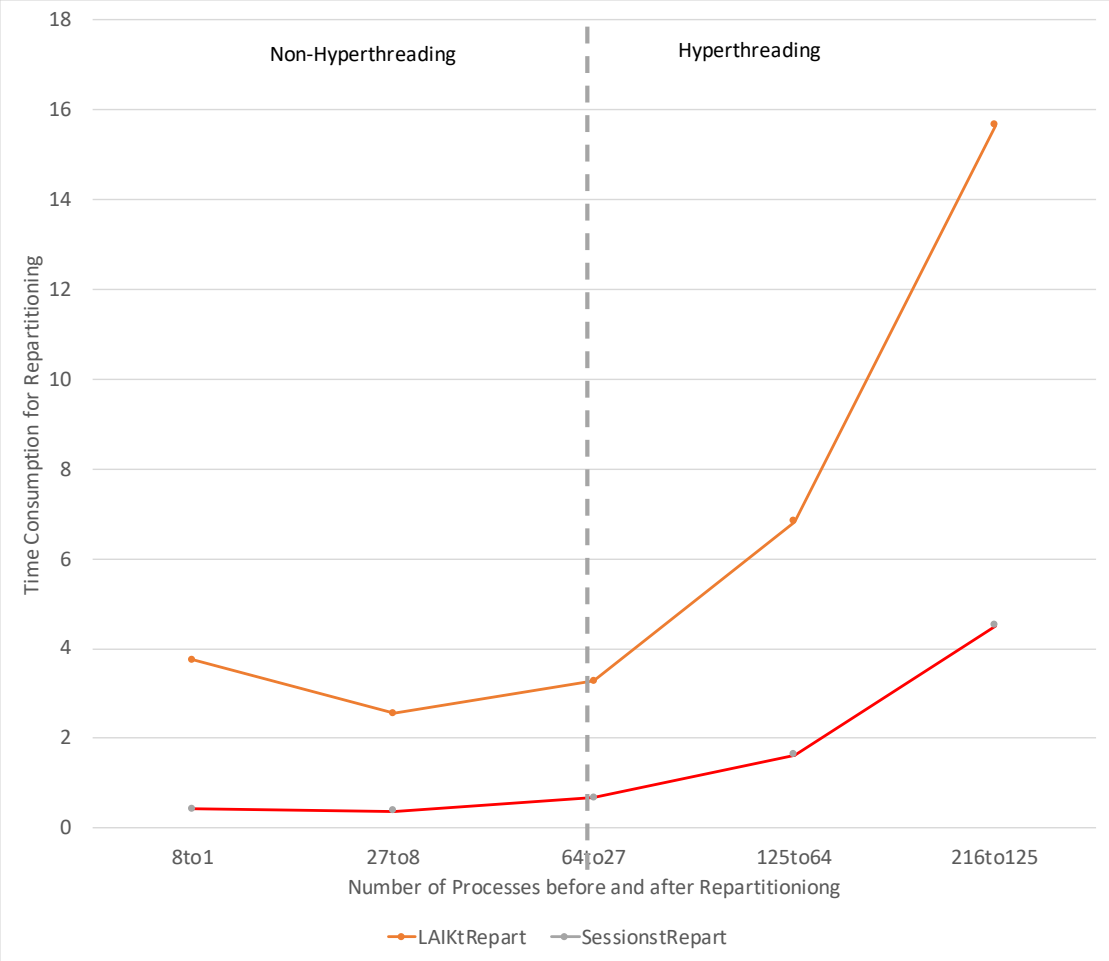

Figure 7.19.: Time for Repartitioning for *sessions lulesh* and *laik lulesh* on *CoolMUC-III*

7.4. Discussion on the Effectiveness of MPI Sessions and MPI Process Sets

In this chapter, we have introduced and considered the concept of the MPI sessions, the MPI Forum's proposal, as well as our own extension to the MPI Forum's proposal – the MPI process sets. We have implemented a prototype of our extension for a minimum set of MPI sessions and MPI process sets functionalities in order to verify the effectiveness of our prototype in achieving fault tolerance by using data migration.

With two real world applications, MLEM and LULESH, we have demonstrated the functionalities of MPI sessions and MPI process sets. Although our prototype has not been optimized for performance evaluation, we have proven that the scalability of LULESH is not affected.

We finish this chapter in the following section, consider what we have discussed, and present our conclusions on the MPI sessions and MPI process sets interfaces.

7.4.1. Functionality of MPI Sessions and MPI Process Sets

The concept of MPI sessions is an active topic in the MPI Forum. Many efforts and discussions on important topics related to MPI sessions have been made within the MPI Forum, including but not limited to:

- Dynamic Session vs. Static Session: Discussions on whether an MPI session can be changed after it has been created. recent discussions from 2019 prefer the static-session approach.
- Local Session vs. Global Session: Discussions on whether an MPI session is a local handle to a local instance of MPI library, which is only accessible and known to the process which creates it. Currently, the MPI Forum prefers the local instance.

However, little effort has been put into the discussions on the role of the MPI process sets. In this work, we provide a potential design for the MPI process-set management part in the MPI sessions proposal, which has the following characteristics:

- MPI process sets are static and versioned: A process set is an immutable object that cannot be changed. Any changes to a process set will automatically trigger the creation of a new version of that given set. This approach matches the current static-sessions concept of the MPI Forum.
- MPI process sets are named using URIs, and they are local objects: Although the MPI process set information is shared with the runtime system, the MPI process set itself is designed to be a local object. A process can only see sets that it belongs

to. This matches the current efforts of the MPI Forum on the sessions concept and helps provide better scalability.

- MPI process sets can be created by the application or the runtime system: In our current design and implementation, the MPI process set is used to share information between the application and the runtime system in both directions. Consequently, it can be used as a bridge between those.

- Asynchronous operation on process sets: Any changes on a process set are queued and performed on the library level. As part of the static design of MPI process sets, any changes are not necessarily propagated to the user application. This reduces performance impact on the application while ensuring backward compatibility.

With our prototype implementation, we have demonstrated the feasibility of these design criteria. We have also shown that our MPI sessions and MPI process sets concepts are fully compatible with existing MPI implementation because our library does not overwrite any existing MPI functions.

We have also proven the usefulness of MPI sessions and MPI process sets for fault tolerance. Since information such as changes to a given process set can be queried by the application, the programmer can choose to react on receiving such information, allowing the application to perform a fault mitigation mechanism, such as data migration. The added dynamic and exposure of runtime system information is beneficial for future applications.

7.4.2. Advantages of MPI Sessions and MPI Process Sets for Fault Tolerance

The main advantages of MPI sessions and MPI process sets for fault tolerance are:

- Simple porting of existing applications: Complex HPC applications, which are based on MPI, usually contain carefully designed communication routines and optimizations in MPI. Our design provides full backward compatibility, which reduces the workload of programmers in porting existing applications. In fact, for the highly optimized example application LULESH, it only takes one hour to program first fully-functional adaptation with our sessions prototype. Application programmers can rely heavily on old code and reuse all of the existing implementation.

- Programmer controlled: Everything is controlled by the programmer: What is being migrated, when a migration is carried out, and what happens to the objects, such as `MPI_Groups` and `MPI_Communicators`. Fine-grained control is often desired by professional application programmer for HPC systems. Furthermore, it limits

the overhead or fault tolerance, as programmers know best what data are required for migration.

- Low design overhead: We do not change any semantics nor implementation of existing MPI functions (except `MPI_Comm_Spawn`, where the implementation, but not semantics is changed). This results in zero overhead introduced by our library.

7.4.3. Disadvantages of MPI Sessions and MPI Process Sets for Fault Tolerance

Our design still has some disadvantages:

- Rookie user-unfriendly: Since we count on the programmers to react to changes to a process set, a higher requirement is placed on the application programmer. Most programmers do not deal with fault tolerance questions nowadays because they prefer to just restart the application if it has failed. This is a significant drawback of our approach when compared to application transparent solutions.
- Changing the program structure to support fault tolerance: in order to support fault tolerance with MPI sessions and MPI process set concepts, major changes are required. Such changes can be error-prone and demanding, e.g., for our LULESH example, roughly about half of the time required for the implementation was spent on debugging (but still lower than for *laik lulesh*). Furthermore, as shown with our evaluations (cf. Section 7.3.3), such changes (e.g., addition of `MPI_Barrier`) in the program flow can lead to degradation of performance.
- The whole design of our MPI process sets interface relies on a scalable and efficient key-value store because shared data is processed by the key-value store. Consequently, such a key-value store must be scalable and efficient, otherwise our design cannot be scalable. Fortunately, current research on runtime systems such as PMIx [Cas+17] targets this issue and may be a promising solution to this dependency.

7.4.4. Limitations of Our Prototype

Since our prototype has been designed for demonstrating of fault tolerance, some shortcomings remain:

- Our prototype does not support the termination of empty processes. An idle process, which is no longer part of an active computation, may call `MPI_Session_finalize`

and terminate itself. However, as the prototype is based on a standard MPI distribution, this call does not terminate the process. Instead, it puts the process into a sleep state until every MPI process has been terminated.

- Our prototype does not implement all the functionalities of MPI sessions proposed by the MPI Forum. Instead, it only provides a minimum set of functionalities to enable support for fault tolerance based on data migration. A list of all the implemented functions is given in Appendix E.1.
- Our prototype only supports shared-memory systems. This is a big drawback because MPI is designed for distributed memory system. However, we were not able to find a suitable key-value store that meets our requirements and, at the same time, is designed for HPC systems. Consequently, we have accepted this limitation because out goal was to demonstrate out technique.

Despite all the disadvantages and limitations mentioned above, we firmly believe that MPI sessions and MPI process sets are promising extensions to the MPI standard. The added failure prediction and runtime system information will greatly help in making future applications more fault-tolerant, which has been demonstrated in our examples. Furthermore, since MPI is the de facto standard programming model for HPC systems, the embodiment of dynamic concepts will also help to change the paradigm of future HPC applications.

8. Discussion

8.1. Fault Tolerance with Data Migration

With LAIK, MPI sessions, and MPI process sets (in the following, just *MPI sessions*), we have shown two different ways of achieving fault tolerance with data migration. Using two example applications, namely MLEM and LULESH, we have demonstrated the effectiveness of both methods. Furthermore, we have shown low performance overhead for mid-scale applications with both LAIK and MPI sessions.

As a result, we believe that, combined with a proper failure prediction system, data migration is a promising technique to achieve fault tolerance for future High Performance Computing (HPC) systems. As introduced in Section 4.2, existing fault mitigation mechanisms such as *checkpoint&restart*, process migration, and data migration, suffer from a high overhead. Since our proposed solution, data migration, is more closer to the application itself, the overhead of data migration can be reduced to a minimum. This is also confirmed in our evaluations.

However, data migration also comes with a significant drawback: For any existing application, data migration requires code changes to be made in the existing implementation. This results in a considerable amount of work for any application programmer. Nevertheless, with MPI sessions, we have shown that the adaptation can be easily done, given that the underlying communication library provides support for proactive fault tolerance. In this case, redistributing the data structures is the only change that an application programmer has to make in order to support data migration. Even for complex applications, such as LULESH, it was possible to port the application within a single day.

Unlike MPI sessions, LAIK goes one step further by providing automatic creation of partitionings, data management, and communication management. This way, fault tolerance with data migration is automatically achieved. However, LAIK follows a different programming model that is based on partitioning the data rather than on the typical message-passing model. Consequently, one has to change a considerable amount of existing code in order to support LAIK. This significantly reduces the usability of LAIK for any existing application.

In summary, we conclude that data migration combined with failure prediction is a

promising approach to achieving fault tolerance in HPC systems for both existing and new applications. For existing applications, MPI sessions is a good solution because it provides full backward compatibility. For new applications, LAIK is a promising new programming model, which automatically provides a solution for data migration. However, the current design of LAIK is highly turned for performance, which results in poor usability.

8.2. From Fault Tolerance to Malleability

The focus of this dissertation is on fault tolerance. However, the trend in HPC systems as well as an increasing demand for efficiency also raises the question of whether an application can be resource adaptive during execution?

The HPC community is carrying out active research on this issue. The so-called "malleability", which is the ability of being adaptive and elastic at different timepoints throughout execution, is a major research topic nowadays. Unlike classic HPC applications, which run in a "fire-and-forget" manner, malleable applications should be resource-aware and be able to adapt to changes in resources during execution. This way, the overall system efficiency can be improved. An example use case is given in Figure 8.1 and described as follows:

- A malleable parallel application is running on three nodes.
- At a later timepoint, this application is informed that one node which it is running on is undergoing a high priority maintenance task and the application has to be removed from the node.
- Upon this request, the application repartitions itself and redistributes its data across the remaining nodes.
- The application continues its calculation.
- Maintenance is performed quickly, and the application can restore its initial configuration and continue its calculation.
- At a later timepoint, the runtime system informs the application that it can use more resources in the HPC system because many of its nodes are idle.
- The application decides to expand itself to six nodes instead of three in order to finish sooner.

In this example, we can see that malleability is very similar to fault tolerance. The only difference is that the application has to not only support data migration to a smaller process size, but also data migration to a larger process size.

Consequently, if an application supports fault tolerance based on data migration, the steps for enabling malleability based on data migration are then trivial.

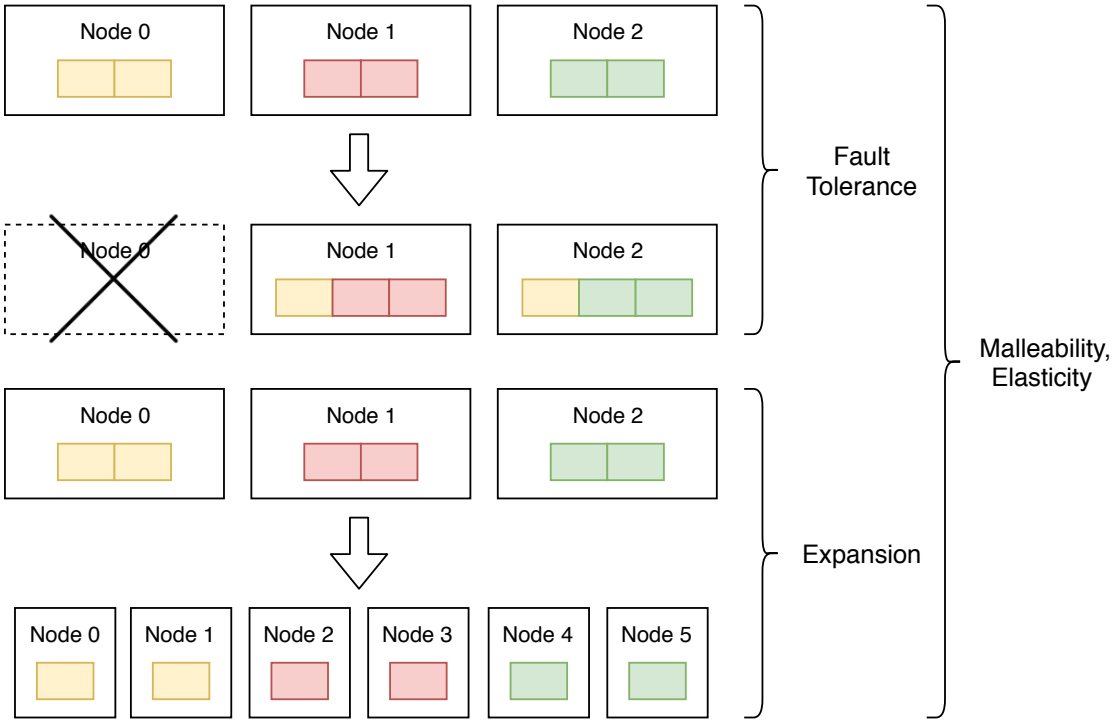

Figure 8.1.: Fault Tolerance vs. Malleability

This means that all the concepts and design we have featured for both LAIK and MPI sessions are still valid and suitable for future application to support malleability based on data migration. With our current MPI sessions implementation, expansion of any process set is already supported because it does not require any additional interfaces from the application's point of view. For LAIK, the support for malleability again relies on the support of the communication backend.

9. Related Work

9.1. State-of-the-practice on Failure Prediction

The main motivation for failure prediction within HPC is the reduction of resource used for re-executing applications.

Two DARPA white papers by Cappello et al. [Cap+14; Cap+09] on resilience for exascale systems showed that HPC systems waste 20% of their computing resources on failure and recovery.

The most important state-of-the-practice survey on failure prediction is provided by Salfner et al. [SLM10], last released in 2010. Their survey provides a classification of online fault prediction methods, in the form of a 1D taxonomy of failure prediction methods. Our work presents an enhanced, 2-D classification of failure prediction methods.

Schroeder et al. repeatedly present different large scale studies on failure mode and failure effect analysis in HPC systems, covering both system level and component level failures. The most prominent examples include but are not limited to [SG07b; SG10; SG07a; SPW09; MSS17].

Many authors criticize the deficiencies in classic prediction metrics. Salfner et al. [SLM10] point out that the *accuracy* metric is not useful because failure events are rare. Zheng et al. [Zhe+10] present new definitions of TP, FN, and FP with respect to lead time and locality of a predicted failure. Taerat et al. [Tae+10] also show the insufficiency of existing statistical metrics due to their not including failure mitigation effects. They introduce a proficiency metric with regard to the time required for mitigation.

9.2. Migration and Checkpointing

Most existing studies on fault tolerance techniques are based on *checkpoint&restart*. Among all the techniques the most prominent solutions are the Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters [HD06], Linux kernel module checkpoint/restart (CRAK) [ZN01], Distributed Multi-Threaded CheckPointing (DMTCP) [AAC09] and Checkpoint/Restore In User space (CRIU) [Vir].

Pickartz et al. [Pic+14] provide an overview of migration methods suitable for failure resilience based on a predicted failure. Gad et al. [Gad+18] provide ways of accelerating VM migration upon a predicted failure. The improved VM migration makes it possible to protect applications within a short lead time. Wang et al. [Wan+08] show that successful live process migration with a lead time of less than 10 seconds could take up to half a minute compared with VM migration.

Further details on migration and checkpointing methods can be found in Chapter 4.

9.3. Fault-tolerant Programming Models

Apart from the MPI-based message passing model for parallel programming, the so-called Partitioned Global Address Space (PGAS) model provides global address spaces and enables programming for good locality of accesses. Implementations for PGAS include new programming languages, such as Chapel [CCZ07] and X10 [Sar+14]. Chapel and X10 suffer from the drawback of requiring programmers to rewrite their legacy codes. Furthermore, user libraries such as the Global Array Toolkit (GAT) [Nie+06], GASPI [GS13], and DASH [Für+14; IFG16] provide support for the PGAS programming model. GAT and GASPI are based on RDMA technology which is barely failure resilient on state-of-the-art HPC communication backbones. DASH relies on C++ templates functionality to provide a selection of standard data structures for the application programmers, while its communication abstraction is built on top of the DART runtime library [Zho+14]. This requirement imposes an additional performance footprint which is hard to optimize by the programmer.

Other approaches similar to LAIK include Legion [TBA13] – a data region based programming system – and Charm++ [KK93; ZSK04] – a C++ library encapsulating workload as so called chares – handling partitioning in a manner similar to LAIK. However, unlike LAIK, both solutions come with their own runtimes, which impose additional overhead on the application.

Many efforts in extending MPI with fault tolerance capabilities have been made in recent years. Bland et al. [Bla+12] introduce User Level Failure Mitigation (ULFM) in MPI which provides fault handling using error codes and messages. Huang et al. [HLK04] present the Adaptive MPI (AMPI) which introduces an abstract mapping between processes and virtual processors. A virtual processor can be remapped to other healthy processors in order to achieve fault tolerance. Their implementation is based on Charm++ [KK93]. Urena et al. [Ure+12] have developed Invasive MPI (IMPI) alongside with the Invasive Resource Manager (IRM), which is capable of dynamic resource (re-)allocation for MPI-based applications while preserving backward compatibility.

10. Conclusion

In this dissertation, we have carried out a comprehensive survey on a large amount of recent literature on state-of-the-art failure prediction methods. We have identified several gaps in the current state of practice for failure prediction, including network components, accelerators (such as GPUs), and recent storage technologies (such as SSD and NVM). We have also determined that the restricted access to failure data and monitoring data, as well as the lack of standardized data sets for training and evaluation, have contributed to the poor comparability of research in failure prediction in HPC systems. Furthermore, We have concluded that the classic evaluation metrics for failure prediction are insufficient. Consequently, we have now committed ourselves to conducting future research that will focus on the practical relevance of fault prediction in HPC systems.

We have presented the concept of *Data Migration*, a proactive fault tolerance technique that is based on the repartitioning and redistribution of data combined with fault prediction. The basic idea behind using data migration is to achieve fault tolerance by removing application data from the failing nodes – this process is called *repartitioning*. Combined with a failure prediction with sufficient lead time, an application can perform repartitioning prior to any failure, thus reducing the demand of resource for re-calculation.

Using two implementations, LAIK and the MPI sessions/MPI process sets, together with two real-world example applications, MLEM and LULESH, we have proven that data migration is a promising concept for fault tolerance. We have also demonstrated the low overhead on performance for both example applications. We conclude that both LAIK and MPI sessions are suitable for fault tolerance based on data migration, while MPI sessions/MPI process sets provide a better backward compatibility and are especially useful for existing applications. Finally, we have extended the concept of data migration for malleability, which not only enables the removal of participating processes but also allows the expansion of process instances that are part of the parallel execution.

11. Future Work

With regard to future work, we are planning to investigate the real-world evaluation metrics of failure prediction methods that take important variables into account, such as the false positive rate and lead time. Furthermore, we intend to quantitatively investigate the impact of failure prediction on different fault-mitigation methods for large scale systems. Moreover, we are preparing to publish an open-access data set containing failure data and monitoring data from HPC systems.

For LAIK, we are planning to simplify LAIK's concept and provide a better API and tools to support application programmers in porting existing applications to LAIK. We are also going to investigate LAIK with other backends. Moreover, we are planning to evaluate concepts from LAIK for other programming models (e.g., MPI), such as the pre-calculation of communication action sequences.

We also intend to work closely with the MPI Forum in order to drive the discussions on the standardization of the MPI sessions/MPI process sets, which are very promising concepts for both fault tolerance and malleability. We are going to compose requirements engineering documents for the runtime system, which is crucial to the further development of MPI sessions. We are also planning to work on the MPI tools interface to provide development support for fault tolerance, such as a fault simulator and a simulated fault predictor.

Appendices

A. Description of Systems

The systems used for the evaluation in this work are introduced and described in this appendix.

A.1. SuperMUC Phase II

SuperMUC Phase II[1] is a high performance computing system installed and operated by the Leibniz Rechenzentrum (LRZ). It was installed by Lenovo in 2015. It consists of 3,072 nodes of Lenovo NeXtScale nx360M5 WCT machines, each equipped with 2 Intel Xeon Haswell Processors E5-2697 v3. The detailed information is shown in Table A.1.

[1] `https://www.lrz.de/services/compute/supermuc/systemdescription/`, accessed May 2019

Table A.1.: Configuration of SuperMUC Phase II

Components	SuperMUC Phase II
CPU	2x Intel Xeon E5-2697 v3
Nominal Freqency	2.6 GHz
Number of Nodes	3072
Number of Cores	86016
Cores per Node	28
NUMA Domains per Node	4
L3 Cache per Node	2x18 MB
Peak Performance	3.58 PFLOP/s
LINPACK Performance	2.814 PFLOP/s
Total Size of Memory	194 TB
Memory per Node	64 GB
Memory Bandwidth	137 GB/s
Number of Islands	6
Interconnect	InfiniBand FDR 14
Intra-Island Topology	non-blocking Tree
Inter-Island Topology	Pruned Tree 4:1
Bisection Bandwidth	5.1TB/s
Operating System	SUSE Linux Enterprise Server
Batch System	IBM Loadleveler
Parallel File System	IBM GPFS
HOME File System	NetApp NAS
Monitoring	lcinga, Splunk

A.2. CoolMUC-2

The CoolMUC-2[2] system is a cluster with Intel Haswell-based nodes and InfiniBand FDR14 interconnect. It is part of the Linux Cluster, which is operated by the Leibniz Rechenzentrum (LRZ). The hardware and system software configuration is shown in Table A.2.

[2] `https://www.lrz.de/services/compute/linux-cluster/overview/`, accessed May 2019

Table A.2.: Configuration of LRZ CoolMUC-2

Components	CoolMUC-2
CPU	2x Intel Xeon E5-2697 v3
Nominal Freqency	2.6 GHz
Number of Nodes	384
Number of Cores	10752
Cores per Node	28
NUMA Domains per Node	2
L3 Cache per Node	2x18 MB
Max Nodes per Job	60
Max Cores per Job	1680
Max Wall Time	48 h
Max Aggregated Memory	3.8 TB
Interconnect	InfiniBand FDR 14
Operating System	SUSE Linux Enterprise Server
Batch System	SLRUM
Parallel File System	IBM GPFS
HOME File System	NetApp NAS

A.3. CoolMUC-3

The CoolMUC-3[3] system is a recent cluster, based on the Intel manycore processor, which is also known as *Knight's Landing*. Unlike other LRZ systems, it features the new Intel OmniPATH Interconnect technology. It is also part of the Linux Cluster, which is operated by LRZ. The detailed hardware and system software configuration is shown in Table Table A.3.

[3] `https://www.lrz.de/services/compute/linux-cluster/coolmuc3/overview/`, accessed May 2019

Table A.3.: Configuration of LRZ CoolMUC-3

Components	CoolMUC-3
CPU	1x Intel Xeon Phi CPU 7210F
Nominal Freqency	1.3 GHz
Number of Nodes	148
Number of Cores	9,472
Cores per Node	64
Hyperthreads per Core	4
Peak Performance	394 TFlop/s
High Bandwidth Memory per Node	16 GB
Main Memory per Node	96 GB
Interconnect	Intel OmpiPATH
Maximum Bandwidth to Interconnect per Node	25 GB/s (2 Links)
Bisection Bandwidth	1.6 TB/s
Latency of Interconnect	2.3 μs
Operating System	SUSE Linux Enterprise Server
Batch System	SLRUM
Parallel File System	IBM GPFS
HOME File System	NetApp NAS
MPI	Intel MPI 2017
Compilers	Intel `icc, icpc, ifort`

B. The System Matrix of MADPET-II

All the data are referenced from [Küs+09; Küs+10].

Table B.1.: MADPET-II Matrix Characteristics

Parameter	**Value**
Total Size (Bytes)	12,838,607,884
Rows (Pair of detectors)	1,327,104
Columns (Voxels)	784,000
Total number of non-zero elements (NNZ)	1,603,498,863
Matrix density (%)	0.1541
Max NNZ in a row	6,537
Min NNZ in a row	0
Avg NNZ in a row	1,208
Max NNZ in a column	6,404
Min NNZ in a column	0
Avg NNZ in a column	2,045

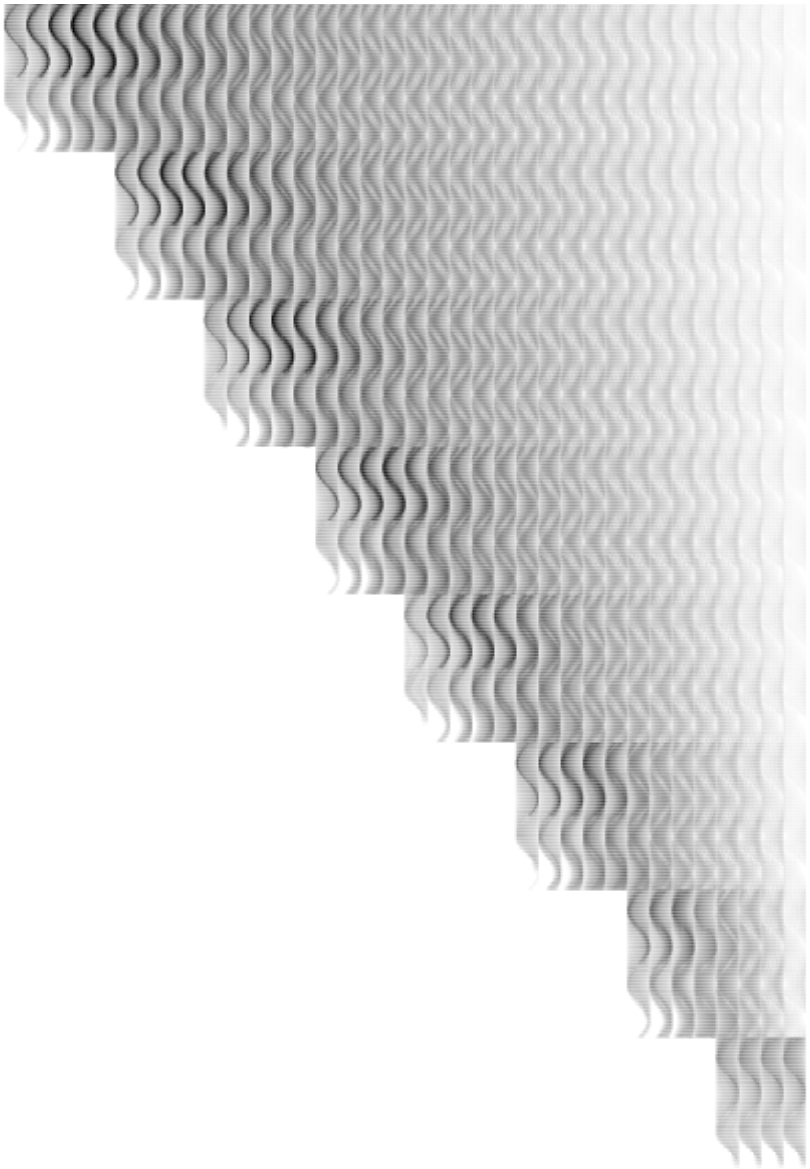

Figure B.1.: Density Plot of the System Matrix MADPET-II

C. Data Structures from LULESH 2.0

Three different types of data structures have been adapted for porting LULESH to both LAIK and the MPI Process Sets:

- Node-related (nodal) data: Listed in Table C.1.
- Element-related (elemental) data: Listed in Table C.2.
- Local temporary data: Listed in Table C.3

Table C.1.: Nodal Variables in LULESH [HKG]

Physical Variable	Description	Program Accessor
$\overrightarrow{X} = (x, y, z)$	position vector	x(), y(), z()
$\overrightarrow{U} = (U_x, U_y, U_z)$	velocity vector	xd(), yd(), zd()
$\overrightarrow{A} = (A_x, A_y, A_z)$	acceleration vector	xdd(), ydd(), zdd()
$\overrightarrow{F} = (F_x, F_y, F_z)$	force vector	fx(), fy(), fz()
m_0	nodal mass	nodalMass()

Table C.2.: Elemental Variables in LULESH [HKG]

Physical Variable	Description	Program Accessor
p	pressure	p()
e	internal energy	e()
q	artificial viscosity	q()
V	relative volume	v()
p	pressure	p()
p	pressure	p()
$\dot{V}/V$	relative volume change per volume	vdov()
$\Delta V = V^{n+1} - V^n$	relative volume change	delv()
V_0	initial volume	volo()
l_{char}	characteristic length	arealg()
$\epsilon = (\epsilon_{xx}, \epsilon_{yy}, \epsilon_{zz})$	diagonal terms of deviatoric strain	dxx(), dyy(), dzz()
q_{lin}	artificial viscosity linear term	ql()
q_{quad}	artificial viscosity quadratic term	qq()

Table C.3.: Additional Local Variables in LULESH

Description	Program Accessor
principle strains	dxx(), dyy(), dzz()
velocity gradients	delv_xi(), delv_eta(), delv_zeta()
coordinate gradients	delx_xi(), delx_eta(), delx_zeta()
new relative volume	vnew()

D. List of Proposed Calls for MPI Sessions

A list of proposed additional MPI calls related to the sessions proposal as given in [Mes18] and [Hol+16] is shown in Table D.1. This information is based on the draft version of MPI Standard 3.2 as of September 2018.

The respective calling parameters and return values are not listed. For detailed information, please refer to the standard draft [Mes18].

Table D.1.: Overview of Proposed Functions for MPI Sessions [Mes18; Hol+16]

Call	Description
Basic Session Functions	
MPI_Session_init()	Create a session object and initialize an MPI instance.
MPI_Session_finalize()	Clean up a session object and finalize an MPI instance.
MPI Group-related Operations	
MPI_Group_from_pset()	Create a process group, bounded to a session, with the processes in a given process set.
MPI_Comm_create_group()	Create a communicator from a group.
Runtime Queries	
MPI_Session_get_num_psets()	Query the runtime for available process sets.
MPI_Session_get_psetlen()	Returns the length of the n-th process set name in bytes.
MPI_Session_get_psetname()	Return the process set name of the n-th process set.
MPI_Info_get_nth_pset()	Get the information on a given process set by its ID.
MPI_Session_get_info	Get the information on a given process set by its name.
MPI Tools Interface	
MPI_T_pvar_handle_alloc()	Binds a specified performance variable to the MPI session object.
MPI_T_pvar_handle_free()	Unbind a performance variable from a session.

Table D.1.: Overview of Proposed Functions for MPI Sessions [Mes18; Hol+16]

Call	Description
MPI_T_pvar_read()	Query information from a performance variable in a given session.
MPI_T_pvar_write()	Write data into a performance variable in a given session.
MPI_T_pvar_reset()	Reset the performance variable in a given session.
MPI_T_pvar_readreset()	Read the information, and reset the performance variable in a given session.
Session Specific Caching Functions	
MPI_Session_create_keyval()	Create a caching key-value store (KVS) in a session.
MPI_Session_free_keyval()	Delete a caching in a session.
MPI_Session_set_attr()	Store a key-value pair in a KVS.
MPI_Session_get_attr()	Load a key-value pair from a KVS by its key.
MPI_Session_delete_attr()	Delete a key-value pair from a KVS.
Sessions Related Error Handling	
MPI_Session_create_errhandler()	Create an error handler that can be attached to a sessions object.
MPI_Session_set_errhandler()	Attaches a new error handler to a session.
MPI_Session_get_errhandler()	Retrieves the error handler currently associated with a session.
MPI_Session_call_errhandler()	Invokes the error handler assigned to a session.

E. New Calls Introduced by Our MPI Sessions / MPI Process Set Library

Detailed information on new calls and interfaces introduced by our MPI session and MPI process set library are documented in this appendix. The adapted designs are based on the previous work by Anilkumar Suma [Sum19].

E.1. MPI Sessions Interface

Table E.1.: Overview of MPI Sessions and MPI Process Sets Calls implemented in our Prototype

Call	Description
Basic Session Functions	
MPI_Session_preparation()	Facilitate internals to initialize existing MPI library.
MPI_Session_init()	Create a session object and initialize an MPI instance.
MPI_Session_finalize()	Clean up a session object and finalize an MPI instance.
MPI_Session_free()	Free all resources used by MPI sessions, including underlying MPI library.
MPI Set Management Operations	
MPI_Session_get_nsets()	Query the runtime for number of available process sets.
MPI_Session_get_pset_names()	Get the names of all available process sets.
MPI_Session_check_in_processet()	Check if current process is in a process set.
MPI_Session_iwatch_pset()	Issue an asynchronous watch for a process set to query its change status later.
MPI_Session_check_psetupdate()	Check if the current process set is updated.
MPI_Session_watch_pset()	Issue a synchronous watch for a process set and wait until it is changed.
MPI_Session_fetch_latestversion()	Get the latest version of a given process set.
MPI_Session_addto_pset()	Add the calling process to a given process set.

Table E.1.: Overview of MPI Sessions and MPI Process Sets Calls implemented in our Prototype

Call	Description
MPI_Session_deletefrom_pset()	Remove a calling process from a given process set.
MPI_Session_get_set_info()	Get internal information from a given process set.
MPI Group and Communicator-related Operations	
MPI_Group_create_from_session()	Create a process group, bounded to a session, with the processes in a given process set.
MPI_Comm_create_from_group()	Create a communicator from a group.
MPIS_Comm_spawn()	Overwrite for MPI Spawn to allow creating new communicator using sessions concept.

E.2. Key-value Store Interface

Table E.2.: Overview of Key-value Store Interface Calls

Call	Description
Basic KVS Functions	
KVS_Get()	Get an entry from the KVS.
KVS_Put()	Set an entry in the KVS.
KVS_Add()	Add a new entry into the KVS.
KVS_Del()	Delete an entry from the KVS.
KVS_initialise()	Initialize and create an entity of the KVS.
KVS_open()	Open an existing KVS.
KVS_free()	Clean up and destroy a KVS.
Derived KVS Functions	
KVS_Get_local_nsets()	Get the number of process sets from KVS.
KVS_Get_local_processsets()	Get the name of process sets from the KVS.
KVS_addto_world()	Add a process to the "world" process set in KVS.
KVS_Watch_keyupdate()	Issue a watch on a KVS entry.
KVS_ask_for_update()	Query whether a previously issued watch is triggered.
KVS_Watch_keyupdate_blocking()	Issue a blocking watch on a KVS entry.

F. List of Own Publications

Conference Full Papers

- Alvaro Frank, Dai Yang, Tim Süß, Martin Schulz, and André Brinkmann. **Reducing False Node Failure Predictions in HPC.** 26th IEEE International Conference on High Performance Computing, Data and Analytics (HiPC) 2019. Accepted for publication.
- Bengisu Elis, Dai Yang, and Martin Schulz. 2019. **QMPI: A Next Generation MPI Profiling Interface for Modern HPC Platforms.** In Proceedings of the 26th European MPI Users' Group Meeting (EuroMPI '19), Torsten Hoefler and Jesper Larsson Träff (Eds.). ACM, New York, NY, USA, Article 4, 10 pages
- David Jauk, Dai Yang, and Martin Schulz. **Predicting Faults in High Performance Computing Systems: An In-Depth Survey of the State-of-the-Practice.** In SC 19': Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, 2019, Denver, Colorado, United States. Accepted for publication.
- Dai Yang, Josef Weidendorfer, Tilman Küstner, Carsten Trinitis, and Sibylle Ziegler. **Enabling Application-Integrated Proactive Fault Tolerance.** In Parallel Computing (PARCO) 2017, Bologna, Italy

Workshop Papers

- Amir Raoofy, Dai Yang, Josef Weidendorfer, Carsten Trinitis, and Martin Schulz. **Enabling Malleability for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics using LAIK**. In PARS-Mitteilungen 2019, Berlin, Germany
- Thomas Becker, Nico Rudolf, Dai Yang, and Wolfgang Karl. **Symptom-based Fault Detection in Modern Computer Systems.** In PARS-Mitteilungen 2019, Berlin, Germany
- Thomas Becker, Dai Yang, Tilman Küstner, and Martin Schulz. **Co-Scheduling in a Tasked-Based Programming Model.** In Workshop on Co-Scheduling of HPC

Applications (COSH'18) in Conjunction with HiPEAC Conference 2018. January 2018, Manchester, United Kingdom

- Josef Weidendorfer, Dai Yang, and Carsten Trinitis. **LAIK: A Library for Fault Tolerant Distribution of Global Data for Parallel Applications.** In PARS Mitteilungen 2017, Hagen, Germany

Articles

- Martin Schulz, Marc-André Hermanns, Michael Knobloch, Kathryn Mohror, Nathan T. Hjelm, Bengisu Elis, Karlo Kraljic, and Dai Yang. **The MPI Tool Interfaces: Past, Present, and Future**.

Posters

- Dai Yang, Moritz Dötterl, Sebastian Rückerl and Amir Raoofy. **Hardening the Linux Kernel agains Soft Errors.** Poster for The 13th International School on the Effects of Radiation on Embedded Systems for Space Applications (SERESSA'17), Garching, Germany.
- Tejas Kale, Dai Yang, Sebastian Rückerl, Martin Schulz. **Event Driven Programming for Embedded Systems.** Poster for 10th European CubeSat Symposium 2018. Toulouse, France

Acronyms

ABFT	Algorithm-based Fault Tolerance.
AFR	Annualized Failure Rates.
AMPI	Adaptive MPI.
ANL	Argonne National Laboratory.
API	Application Programming Interface.
ARMA	AutoRegressive Moving Average.
ARR	Annualized Replacement Rate.
AVX	Advanced Vector Extensions.
BJPCS	Batch Job Processing Clustering System.
BLAS	Basic Linear Algebra Subprograms.
BLCR	Berkeley Lab Checkpoint Restart.
CDF	Cumulative Distribution Function.
CFDR	Computer Failure Data Repository.
COMA	Cache-Only Memory Architecture.
COTS	Commercial-off-the-Shelf.
CPU	Central Processing Unit.
CRIU	Checkpoint/Restart in Userspace.
CSR	Compressed Sparse Row.
CU	Control Unit.
DARPA	Defense Advance Research Projects Agency.
DBMS	Database Management System.
DFS	Distributed File System.
DMA	Direct Memory Access.
DMTCP	Distributed Multi-Threaded Checkpointing.
DNN	Deep Neural Network.
DRAM	Dynamic Random Access Memory.
ECC	Error Correcting Code.

FC	Fibre Channel.
FLOPS	Floating Point Operations per Second.
FOV	Field of View.
FPGA	Field Programmable Gate Array.
FTA	Fault Tree Analysis.
GPU	Graphical Processing Unit.
HBM	High Bandwidth Memory.
HDD	Hard Disk Drive.
HMM	Hidden Markov Models.
HPC	High Performance Computing.
HPCG	High Performance Conjugate Gradient.
HPL	High Performance LINPACK.
IMPI	Invasive MPI.
IPMI	Intelligent Platform Management Interface.
IRM	Invasive Resource Manager.
LAIK	Lightweight Application-Integrated Fault-Tolerant Data Container.
LANL	Los Alamos National Laboratory.
LLNL	Lawrence Livermore National Laboratory.
LOR	Line of Response.
LRZ	Leibniz Rechenzentrum der Bayerischen Akademie der Wissenschaft.
LSTM	Long Short-Term Memory.
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics.
MIMD	Multiple Instruction Streams Multiple Data Streams.
MISD	Multiple Instruction Streams Single Data Stream.
MLEM	Maximum-Likelihood Expectation-Maximization.
MPI	Message Passing Interface.
MPMD	Multiple Program Multiple Data.
MTBF	Mean Time Between Failure.
NAS	Network Accessed Storage.

NEON	Advanced SIMD Extension.
NN	Neural Network.
NUMA	Non-uniform Memory Access.
NVM	Non-volatile Memory.
ORNL	Oak Ridge National Laboratory.
OS	Operating System.
PCIe	Peripheral Component Interconnect Express.
PET	Positron Emission Tomography.
PGAS	Partitioned Global Address Space.
PMI	Process Management Interface.
PU	Processing Unit.
RAID	Redundant Array of Independent/Inexpensive Disks.
RBML	Rule-based Machine Learning.
RDMA	Remote Direct Memory Access.
RNN	Recurrent Neural Network.
ROCC	Receiver Operating Characteristic Curve.
SAN	Storage Area Network.
SATA	Serial AT Attachment.
SBP	Sedov Blast Problem.
SCSI	Small Computer System Interface.
SIMD	Single Instruction Stream Multiple Data Streams.
SISD	Single Instruction Stream Single Data Stream.
SLURM	Simple Linux Utility for Resource Management Workload Manager.
SMART	Self-monitoring, Analysis and Reporting Technology.
SNL	Sandia National Laboratory.
SPMD	Single Program Multiple Data.
SpMV	Sparse Matrix-Vector.
SSD	Solid State Drive.
SVM	Support Vector Machine.
TCP	Transport Control Protocol.
UDP	User Datagram Protocol.

ULFM	User Level Failure Mitigation.
UMA	Uniform Memory Access.
UPS	Uninterruptible Power Supply.
URI	Uniform Resource Identifier.
VLIW	Very Long Instruction Word.
VM	Virtual Machine.
VMM	Virtual Machine Manager.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AAC09] J. Ansel, K. Arya, and G. Cooperman. "DMTCP: Transparent checkpointing for cluster computations and the desktop." In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–12.

[Ahn+14] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz. "Flux: a next-generation resource management framework for large HPC centers." In: *2014 43rd International Conference on Parallel Processing Workshops*. IEEE. 2014, pp. 9–17.

[AL88] C. J. Anfinson and F. T. Luk. "A linear algebraic model of algorithm-based fault tolerance." In: *IEEE Transactions on Computers* 37.12 (1988), pp. 1599–1604.

[Amd67] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities." In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[Ata98] M. J. Atallah. *Algorithms and theory of computation handbook*. CRC press, 1998.

[Aug+11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures." In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.

[Avi+04] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[Avi76] A. Avizienis. "Fault-tolerant systems." In: *IEEE Trans. Computers* 25.12 (1976), pp. 1304–1312.

[AWR15] B. Agrawal, T. Wiktorski, and C. Rong. "Analyzing and Predicting Failure in Hadoop Clusters Using Distributed Hidden Markov Model." In: *Revised Selected Papers of the Second Int'l Conf. on Cloud Computing and Big Data - Volume 9106*. CloudCom-Asia 2015. Huangshan, China: Springer New York, 2015, pp. 232–246.

[Bab+00] Š. H. Babič, P. Kokol, V. Podgorelec, M. Zorman, M. Šprogar, and M. M. Štiglic. "The Art of Building Decision Trees." In: *Journal of Medical Systems* 24.1 (Feb. 2000), pp. 43–52. ISSN: 1573-689X.

[Bal+10] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. "PMI: A scalable parallel process-management interface for extreme-scale systems." In: *European MPI Users' Group Meeting*. Springer. 2010, pp. 31–41.

[Ban+90] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham. "Algorithm-based fault tolerance on a hypercube multiprocessor." In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1132–1145.

[BAS04] R. Budruk, D. Anderson, and T. Shanley. *PCI express system architecture*. Addison-Wesley Professional, 2004.

[Bau+12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. "Legion: Expressing locality and independence with logical regions." In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–11.

[Bau+16] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith. "Unprotected Computing: A Large-Scale Study of DRAM Raw Error Rate on a Supercomputer." In: *SC16: Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*. Nov. 2016, pp. 645–655.

[Ber+08] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. "Exascale computing study: Technology challenges in achieving exascale systems." In: *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep* 15 (2008).

[BGV92] B. E. Boser, I. M. Guyon, and V. N. Vapnik. "A Training Algorithm for Optimal Margin Classifiers." In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT '92. Pittsburgh, Pennsylvania, USA: ACM, 1992, pp. 144–152. ISBN: 0-89791-497-X.

[Bla+12] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. *Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting*. Vol. 28. Chapter: An Evaluation of User-Level Failure Mitigation Support in MPI, pages 193-203. Vienna, Austria: Springer Berlin Heidelberg, Berlin,Heidelberg, 2012.

[Bon00] A. B. Bondi. "Characteristics of scalability and their impact on performance." In: *Proceedings of the 2nd international workshop on Software and performance*. ACM. 2000, pp. 195–203.

[Bos+09] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. "Algorithm-based fault tolerance applied to high performance computing." In: *Journal of Parallel and Distributed Computing* 69.4 (2009), pp. 410–416.

[Bra+09] J. Brandt, A. Gentile, J. Mayo, P. Pébay, D. Roe, D. Thompson, and M. Wong. "Methodologies for Advance Warning of Compute Cluster Problems via Statistical Analysis: A Case Study." In: *Proceedings of the 2009 Workshop on Resiliency in High Performance*. Resilience '09. Garching, Germany: ACM, 2009, pp. 7–14.

[Cap+09] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. "Toward exascale resilience." In: *The International Journal of High Performance Computing Applications* 23.4 (2009), pp. 374–388.

[Cap+14] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. "Toward exascale resilience: 2014 update." In: *Supercomputing frontiers and innovations* 1.1 (2014), pp. 5–28.

[Cas+17] R. H. Castain, D. Solt, J. Hursey, and A. Bouteiller. "PMIx: Process Management for Exascale Environments." In: *Proceedings of the 24th European MPI Users' Group Meeting*. EuroMPI '17. Chicago, Illinois: ACM, 2017, 14:1–14:10. ISBN: 978-1-4503-4849-2. DOI: 10.1145/3127024.3127027.

[CAS12] T. Chalermarrewong, T. Achalakul, and S. C. W. See. "Failure Prediction of Data Centers Using Time Series and Fault Tree Analysis." In: *2012 IEEE 18th Int'l Conf. on Parallel and Distributed Systems*. Dec. 2012, pp. 794–799.

[CCZ07] B. L. Chamberlain, D. Callahan, and H. P. Zima. "Parallel programmability and the chapel language." In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312.

[CD08] Z. Chen and J. Dongarra. "Algorithm-based fault tolerance for fail-stop failures." In: *IEEE Transactions on Parallel and Distributed Systems* 19.12 (2008), pp. 1628–1641.

[Cha+01] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[Che13] Z. Chen. "Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods." In: *ACM SIGPLAN Notices*. Vol. 48. 8. ACM. 2013, pp. 167–176.

[CKC12] H. Chung, M. Kang, and H.-D. Cho. "Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM® big.LITTLE Technology." In: *Samsung White Paper* (2012).

[Cla+05] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. "Live migration of virtual machines." In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association. 2005, pp. 273–286.

[CLG05] J. Cao, Y. Li, and M. Guo. "Process migration for MPI applications based on coordinated checkpoint." In: *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*. Vol. 1. IEEE. 2005, pp. 306–312.

[CLP14] X. Chen, C.-D. Lu, and K. Pattabiraman. "Failure Prediction of Jobs in Compute Clouds: A Google Cluster Case Study." In: *2014 IEEE Int'l. Symp. on Software Reliability Engineering Workshops*. Nov. 2014, pp. 341–346.

[Coh95] W. W. Cohen. "Fast Effective Rule Induction." In: *Proceedings of the Twelfth Int'l Conf. on Machine Learning*. ICML'95. Tahoe City, California, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 115–123. ISBN: 1-55860-377-8.

[Cos+14] C. H. Costa, Y. Park, B. S. Rosenburg, C.-Y. Cher, and K. D. Ryu. "A System Software Approach to Proactive Memory-Error Avoidance." In: *SC14: Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*. Nov. 2014, pp. 707–718.

[Dar+88] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. "A single-program-multiple-data computational model for EPEX/FORTRAN." In: *Parallel Computing* 7.1 (1988), pp. 11–24.

[Del97] T. J. Dell. "A white paper on the benefits of chipkill-correct ECC for PC server main memory." In: *IBM Microelectronics Division* 11 (1997), pp. 1–23.

[DHL15] J. Dongarra, M. A. Heroux, and P. Luszczek. "HPCG benchmark: a new metric for ranking high performance computing systems." In: *Knoxville, Tennessee* (2015).

[DHS05] C. Ding, X. He, and H. D. Simon. "On the equivalence of nonnegative matrix factorization and spectral clustering." In: *Proceedings of the 2005 SIAM international conference on data mining*. SIAM. 2005, pp. 606–610.

[Di +15] C. Di Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer. "Measuring and Understanding Extreme-Scale Application Resilience: A Field Study of 5,000,000 HPC Application Runs." In: *2015 45th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks*. June 2015, pp. 25–36.

[DLT09] L. Di-Jorio, A. Laurent, and M. Teisseire. "Mining Frequent Gradual Itemsets from Large Databases." In: *Advances in Intelligent Data Analysis VIII: 8th Int'l. Symp. on Intelligent Data Analysis, IDA 2009, Lyon, France*. Ed. by N. M. Adams, C. Robardet, A. Siebes, and J.-F. Boulicaut. Berlin, Heidelberg: Springer, 2009, pp. 297–308.

[DM98] L. Dagum and R. Menon. "OpenMP: An industry-standard API for shared-memory programming." In: *Computing in Science & Engineering* 1 (1998), pp. 46–55.

[DO91] F. Douglis and J. Ousterhout. "Transparent process migration: Design alternatives and the Sprite implementation." In: *Software: Practice and Experience* 21.8 (1991), pp. 757–785.

[Don+79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK users' guide*. Vol. 8. Siam, 1979.

[Dur+11] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. "Ompss: a proposal for programming heterogeneous multi-core architectures." In: *Parallel processing letters* 21.02 (2011), pp. 173–193.

[EA05] H. El-Rewini and M. Abd-El-Barr. *Advanced computer architecture and parallel processing*. Vol. 42. John Wiley & Sons, 2005.

[Ell+12] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. "Combining Partial Redundancy and Checkpointing for HPC." In: *2012 IEEE 32nd International Conference on Distributed Computing Systems*. June 2012, pp. 615–626. DOI: 10.1109/ICDCS.2012.56.

[ES13] N. El-Sayed and B. Schroeder. "Reading between the lines of failure logs: Understanding how HPC systems fail." In: *2013 43rd Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN)*. June 2013, pp. 1–12.

[ET93] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. New York: Chapman & Hall, 1993.

[EZS17] N. El-Sayed, H. Zhu, and B. Schroeder. "Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations." In: *2017 IEEE 37th Int'l Conf. on Distributed Computing Systems (ICDCS)*. June 2017, pp. 1333–1344.

[Faw06] T. Fawcett. "An Introduction to ROC Analysis." In: *Pattern Recogn. Lett.* 27.8 (June 2006), pp. 861–874. ISSN: 0167-8655. DOI: 10.1016/j.patrec.2005.10.010.

[FD17] D. Foley and J. Danskin. "Ultra-performance Pascal GPU and NVLink interconnect." In: *IEEE Micro* 37.2 (2017), pp. 7–17.

[FH60] G. E. Forsythe and P. Henrici. "The cyclic Jacobi method for computing the principal values of a complex matrix." In: *Transactions of the American Mathematical Society* 94.1 (1960), pp. 1–23.

[FHT01] J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[Fir+08] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. "Intel AVX: New frontiers in performance improvements and energy efficiency." In: *Intel white paper* 19.20 (2008).

[Fly72] M. J. Flynn. "Some Computer Organizations and Their Effectiveness." In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[For73] G. D. Forney. "The viterbi algorithm." In: *Proceedings of the IEEE* 61.3 (1973), pp. 268–278.

[Fra+19] A. Frank, D. Yang, T. Süß, and A. Brinkmann. "Reducing False Node Failure Predictions in HPC." In: *26th IEEE International Conference on High Performance Computing, Data, Analtics and Data Science* (2019). Accepted for Publication.

[Fu+12] X. Fu, R. Ren, J. Zhan, W. Zhou, Z. Jia, and G. Lu. "LogMaster: Mining Event Correlations in Logs of Large-Scale Cluster Systems." In: *2012 IEEE 31st Symp. on Reliable Distributed Systems*. Oct. 2012, pp. 71–80.

[Fu+14] X. Fu, R. Ren, S. A. McKee, J. Zhan, and N. Sun. "Digging deeper into cluster system logs for failure prediction and root cause diagnosis." In: *2014 IEEE Int'l Conf. on Cluster Computing (CLUSTER)*. Sept. 2014, pp. 103–112.

[Für+14] K. Fürlinger, C. Glass, J. Gracia, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedheb, and H. Zhou. "DASH: Data structures and algorithms with support for hierarchical locality." In: *European Conference on Parallel Processing*. Springer. 2014, pp. 542–552.

[Gad+18] R. Gad, S. Pickartz, T. Süß, L. Nagel, S. Lankes, A. Monti, and A. Brinkmann. "Zeroing memory deallocator to reduce checkpoint sizes in virtualized HPC environments." In: *The Journal of Supercomputing* 74.11 (2018), pp. 6236–6257.

[Gai+12] A. Gainaru, F. Cappello, M. Snir, and W. Kramer. "Fault prediction under the microscope: A closer look into HPC systems." In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 Int'l Conf. for*. Nov. 2012, pp. 1–11.

[Gai+13] A. Gainaru, F. Cappello, M. Snir, and W. Kramer. "Failure prediction for HPC systems and applications: Current situation and open issues." In: *The Int'l Journal of High Performance Computing Applications* 27.3 (2013), pp. 273–282.

[Gai+14] A. Gainaru, M. S. Bouguerra, F. Cappello, M. Snir, and W. Kramer. "Navigating the blue waters: Online failure prediction in the petascale era." In: *Argonne National Laboratory Technical Report, ANL/MCS-P5219–1014* (2014).

[Gan+16] S. Ganguly, A. Consul, A. Khan, B. Bussone, J. Richards, and A. Miguel. "A Practical Approach to Hard Disk Failure Prediction in Cloud Platforms: Big Data Model for Failure Management in Datacenters." In: *2016 IEEE Second Int'l Conf. on Big Data Computing Service and Applications (BigDataService)*. Mar. 2016, pp. 105–116.

[GCK12] A. Gainaru, F. Cappello, and W. Kramer. "Taming of the Shrew: Modeling the Normal and Faulty Behaviour of Large-scale HPC Systems." In: *26th IEEE Int'l Parallel and Distributed Processing Symp.* May 2012, pp. 1168–1179.

[GDU07] J. M. Guerrero, L. G. De Vicuna, and J. Uceda. "Uninterruptible power supply systems provide protection." In: *IEEE Industrial Electronics Magazine* 1.1 (2007), pp. 28–38.

[Gha02] Z. Ghahramani. "Hidden Markov Models." In: River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2002. Chap. An Introduction to Hidden Markov Models and Bayesian Networks, pp. 9–42. ISBN: 981-02-4564-5.

[Ghi+16] S. Ghiasvand, F. M. Ciorba, R. Tschüter, and W. E. Nagel. "Lessons Learned from Spatial and Temporal Correlation of Node Failures in High Performance Computers." In: *2016 24th Euromicro Int'l Conf. on Parallel, Distributed, and Network-Based Processing (PDP)*. Feb. 2016, pp. 377–381.

[GM03] P. Gillingham and B. Millar. *High bandwidth memory interface*. US Patent 6,510,503. Jan. 2003.

[GR17] A. Geist and D. A. Reed. "A survey of high-performance computing scaling challenges." In: *The Int'l Journal of High Performance Computing Applications* 31.1 (2017), pp. 104–113.

[GS13] D. Grünewald and C. Simmendinger. "The GASPI API specification and its implementation GPI 2.0." In: *7th International Conference on PGAS Programming Models*. 2013, p. 243.

[Gsc+06] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. "Synergistic processing in cell's multicore architecture." In: *IEEE micro* 26.2 (2006), pp. 10–24.

[Gu+08] J. Gu, Z. Zheng, Z. Lan, J. White, E. Hocks, and B.-H. Park. "Dynamic Meta-Learning for Failure Prediction in Large-Scale Systems: A Case Study." In: *2008 37th Int'l Conf. on Parallel Processing*. Sept. 2008, pp. 157–164.

[Gus88] J. L. Gustafson. "Reevaluating Amdahl's law." In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[GZF11] Q. Guan, Z. Zhang, and S. Fu. "Proactive Failure Management by Integrated Unsupervised and Semi-Supervised Learning for Dependable Cloud Systems." In: *2011 Sixth Int'l Conf. on Availability, Reliability and Security*. Aug. 2011, pp. 83–90.

[HD06] P. H. Hargrove and J. C. Duell. "Berkeley Lab Checkpoint/Restart (BLCR) for linux clusters." In: *Journal of Physics: Conference Series*. Vol. 46. 1. IOP Publishing. 2006, p. 494.

[HKG] R. D. Hornung, J. A. Keasler, and M. B. Gokhale. *Hydrofailure prediction Challenge Problem, Lawrence Livermore National Laboratory*. Tech. rep. LLNL-TR-490254. Livermore, CA, pp. 1–17.

[HLK04] C. Huang, O. Lawlor, and L. V. Kalé. "Adaptive MPI." In: *Languages and Compilers for Parallel Computing*. Ed. by L. Rauchwerger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 306–322. ISBN: 978-3-540-24644-2.

[Ho95] T. K. Ho. "Random Decision Forests." In: *Proceedings of the Third Int'l Conf. on Document Analysis and Recognition (Vol. 1)*. ICDAR '95. Washington, DC, USA: IEEE Computer Society, 1995. ISBN: 0-8186-7128-9.

[Hol+16] D. Holmes, K. Mohror, R. E. Grant, A. Skjellum, M. Schulz, W. Bland, and J. M. Squyres. "MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale." In: *Proceedings of the 23rd European MPI Users' Group Meeting*. EuroMPI 2016. Edinburgh, United Kingdom: ACM, 2016, pp. 121–129. ISBN: 978-1-4503-4234-6. DOI: 10.1145/2966884.2966915.

[HP11] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[HRC09] T. J. Hacker, F. Romero, and C. D. Carothers. "An analysis of clustered failures on large supercomputing systems." In: *Journal of Parallel and Distributed Computing* 69.7 (2009), pp. 652–665. ISSN: 0743-7315.

[HSS12] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. "Cosmic Rays Don'T Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design." In: *SIGPLAN Not.* 47.4 (Mar. 2012), pp. 111–122. ISSN: 0362-1340.

[HTC+89] A. R. Hoffman, J. F. Traub, N. R. Council, et al. *Supercomputers: directions in technology and applications*. National Academies, 1989.

[Hua+84] K.-H. Huang et al. "Algorithm-based fault tolerance for matrix operations." In: *IEEE transactions on computers* 100.6 (1984), pp. 518–528.

[IFG16] K. Idrees, T. Fuchs, and C. W. Glass. "Effective use of the PGAS paradigm: Driving transformations and self-adaptive behavior in dash-applications." In: *arXiv preprint arXiv:1603.01536* (2016).

[IM17] T. Islam and D. Manivannan. "Predicting Application Failure in Cloud: A Machine Learning Approach." In: *2017 IEEE Int'l Conf. on Cognitive Computing (ICCC)*. June 2017.

[Int04] Intel Corporation. "Enhanced SpeedStep® technology for the Intel® Pentium® M processor." In: *Intel Technology White Paper* (2004).

[Jau17] D. Jauk. "Failure Prediction in High Performance Computing." MA thesis. Boltzmannstr. 3, 85748 Garching: Technical University Munich, 2017.

[JDD12] W. M. Jones, J. T. Daly, and N. DeBardeleben. "Application Monitoring and Checkpointing in HPC: Looking Towards Exascale Systems." In: *Proceedings of the 50th Annual Southeast Regional Conference*. ACM-SE '12. Tuscaloosa, Alabama: ACM, 2012, pp. 262–267. ISBN: 978-1-4503-1203-5. DOI: 10.1145/2184512.2184574.

[Joy+83] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher. *4.2 BSD system manual*. Tech. rep. Technical report, Computer Systems Research Group, University of California, 1983.

[JR13] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[JYS19] D. Jauk, D. Yang, and M. Schulz. "Predicting Faults in High Performance Computing Systems: An In-Depth Survey of the State-of-the-Practice." In: *SC: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. Accepted for Publication. ACM Press. 2019, t.b.d.

[KK11] D. M. Kunzman and L. V. Kale. "Programming Heterogeneous Systems." In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. May 2011, pp. 2061–2064. DOI: 10.1109/IPDPS.2011.377.

[KK93] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*. Vol. 28. Citeseer, 1993.

[Kli+17] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Müller. "Data Mining-Based Analysis of HPC Center Operations." In: *2017 IEEE Int'l Conf. on Cluster Computing (CLUSTER)*. Sept. 2017, pp. 766–773.

[Küs+09] T. Küstner, J. Weidendorfer, J. Schirmer, T. Klug, C. Trinitis, and S. Ziegler. "Parallel MLEM on Multicore Architectures." In: *Computational Science – ICCS 2009*. Ed. by G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, and P. M. A. Sloot. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 491–500. ISBN: 978-3-642-01970-8.

[Küs+10] T. Küstner, P. Pedron, J. Schirmer, M. Hohberg, J. Weidendorfer, and S. I. Ziegler. "Fast system matrix generation using the detector response function model on Fermi GPUs." In: *2010 Nuclear Science Symposium and Medical Imaging Conference*. 2010.

[KZP06] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas. "Machine learning: a review of classification and combining techniques." In: *Artificial Intelligence Review* 26.3 (Nov. 2006), pp. 159–190. ISSN: 1573-7462.

[Lan+10] Z. Lan, J. Gu, Z. Zheng, R. Thakur, and S. Coghlan. "A study of dynamic meta-learning for failure prediction in large-scale systems." In: *Journal of Parallel and Distributed Computing* 70.6 (2010), pp. 630–643.

[Law+77] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic linear algebra subprograms for Fortran usage." In: (1977).

[LH10] O. Laadan and S. E. Hallyn. "Linux-CR: Transparent application checkpoint-restart in Linux." In: *Linux Symposium*. Vol. 159. Citeseer. 2010.

[LHY10] H.-C. W. Lin, Y. (He, and W.-S. Yang. "Franklin Job Completion Analysis." In: *CUG 2010 Proceedings*. 2010, pp. 1–12.

[Lia+06] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. "BlueGene/L Failure Analysis and Prediction Models." In: *Int'l Conf. on Dependable Systems and Networks (DSN'06)*. June 2006, pp. 425–434.

[Lia+07] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. "Failure Prediction in IBM BlueGene/L Event Logs." In: *Seventh IEEE Int'l Conf. on Data Mining (ICDM 2007)*. Oct. 2007, pp. 583–588.

[Lie11] J. H. Lienhard. *A heat transfer textbook*. Courier Corporation, 2011.

[Lin+14] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. "Microsoft coco: Common objects in context." In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[Liu+11a] Q. Liu, G. Jin, J. Zhou, Q. Sun, and M. Xi. "Bayesian serial revision method for RLLC cluster systems failure prediction." In: *Journal of Systems Engineering and Electronics* 22.2 (Apr. 2011).

[Liu+11b] Q. Liu, J. Zhou, G. Jin, Q. Sun, and M. Xi. "FaBSR: a method for cluster failure prediction based on Bayesian serial revision and an application to LANL cluster." In: *Quality and Reliability Engineering Int'l* 27.4 (2011), pp. 515–527. ISSN: 1099-1638.

[Ma+15] A. Ma, R. Traylor, F. Douglis, M. Chamness, G. Lu, D. Sawyer, S. Chandra, and W. Hsu. "RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures." In: *ACM Trans. Storage* 11.4 (Nov. 2015), 17:1–17:28. ISSN: 1553-3077.

[Mes18] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.2 (draft)*. Unofficial, for comment only. 2018.

[Mil+00] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. "Process migration." In: *ACM Computing Surveys (CSUR)* 32.3 (2000), pp. 241–299.

[Mit16] S. Mittal. "A Survey Of Architectural Techniques for Managing Process Variation." In: *ACM Computing Surveys* 48 (Feb. 2016). DOI: 10.1145/2871167.

[Mos93] D. Mosberger. "Memory Consistency Models." In: *SIGOPS Oper. Syst. Rev.* 27.1 (Jan. 1993), pp. 18–26. ISSN: 0163-5980. DOI: 10.1145/160551.160553.

[MSS17] F. Mahdisoltani, I. Stefanovici, and B. Schroeder. "Proactive error prediction to improve storage system reliability." In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 2017, pp. 391–402.

[MYL17] L. Ma, S. Yi, and Q. Li. "Efficient service handoff across edge servers via docker container migration." In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM. 2017, p. 11.

[NAC11] N. Nakka, A. Agrawal, and A. Choudhary. "Predicting node failure in high performance computing systems from failure and usage logs." In: *IEEE Int'l. Parallel and Distributed Processing Symp., Workshops and Phd Forum, (IPDPSW)*. 2011, pp. 1557–1566. ISBN: 9780769543857.

[Nad+17] S. Nadgowda, S. Suneja, N. Bila, and C. Isci. "Voyager: Complete container state migration." In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 2137–2142.

[Nag+07] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. "Proactive Fault Tolerance for HPC with Xen Virtualization." In: *Proceedings of the 21st Annual International Conference on Supercomputing*. ICS '07. Seattle, Washington: ACM, 2007, pp. 23–32. ISBN: 978-1-59593-768-1. DOI: 10.1145/1274971.1274978.

[Nie+06] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. "Advances, applications and performance of the global arrays shared memory programming toolkit." In: *The International Journal of High Performance Computing Applications* 20.2 (2006), pp. 203–231.

[Nie+17] B. Nie, J. Xue, S. Gupta, C. Engelmann, E. Smirni, and D. Tiwari. "Characterizing Temperature, Power, and Soft-Error Behaviors in Data Center Systems: Insights, Challenges, and Opportunities." In: *2017 IEEE 25th Int'l. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2017, pp. 22–31.

[NLH+05] M. Nelson, B.-H. Lim, G. Hutchins, et al. "Fast Transparent Migration for Virtual Machines." In: *USENIX Annual technical conference, general track*. 2005, pp. 391–394.

[Pat+17] A. Patwari, I. Laguna, M. Schulz, and S. Bagchi. "Understanding the Spatial Characteristics of DRAM Errors in HPC Clusters." In: *Proceedings of the 2017 Workshop on Fault Tolerance for HPC at Extreme Scale*. FTXS '17. Washington, DC, USA: ACM, 2017, pp. 17–22.

[Pat+89] D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz. "Introduction to redundant arrays of inexpensive disks (RAID)." In: *Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*. IEEE. 1989, pp. 112–117.

[Pat98] D. W. Patterson. *Artificial Neural Networks: Theory and Applications*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998. ISBN: 0132953536.

[Pel+14] A. Pelaez, A. Quiroz, J. C. Browne, E. Chuah, and M. Parashar. "Online failure prediction for HPC resources using decentralized clustering." In: *2014 21st Int'l Conf. on High Performance Computing (HiPC)*. Dec. 2014, pp. 1–9.

[Pic+14] S. Pickartz, R. Gad, S. Lankes, L. Nagel, T. Süß, A. Brinkmann, and S. Krempel. "Migration techniques in HPC environments." In: *European Conference on Parallel Processing*. Springer. 2014, pp. 486–497.

[Pic+16a] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti. "Migrating LinuX containers using CRIU." In: *International Conference on High Performance Computing*. Springer. 2016, pp. 674–684.

[Pic+16b] S. Pickartz, S. Lankes, A. Monti, C. Clauss, and J. Breitbart. "Application migration in HPC - A driver of the exasclae era?" In: *2016 International Conference on High Performance Computing and Simulation (HPCS)*. IEEE. 2016, pp. 318–325.

[Pic+18] S. Pickartz, C. Clauss, J. Breitbart, S. Lankes, and A. Monti. "Prospects and challenges of virtual machine migration in HPC." In: *Concurrency and Computation: Practice and Experience* 30.9 (2018), e4412.

[PWB07] E. Pinheiro, W.-D. Weber, and L. A. Barroso. "Failure Trends in a Large Disk Drive Population." In: *5th USENIX Conference on File and Storage Technologies (FAST 2007)*. 2007, pp. 17–29.

[Qiu+17] Y. Qiu, C. Lung, S. Ajila, and P. Srivastava. "LXC Container Migration in Cloudlets under Multipath TCP." In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. July 2017, pp. 31–36. DOI: 10.1109/COMPSAC.2017.163.

[Rao+19] A. Raoofy, D. Yang, J. Weidendorfer, C. Trinitis, and M. Schulz. "Enabling Malleability for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics using LAIK." In: *PARS-Mitteilungen 2019, Berlin, Germany*. Accepted for Publication. 2019.

[RBP12] R. Rajachandrasekar, X. Besseron, and D. K. Panda. "Monitoring and Predicting Hardware Failures in HPC Clusters with FTB-IPMI." In: *26th IEEE Int'l Parallel and Distributed Processing Symp., Workshops and PhD Forum*. May 2012.

[Red08] V. G. Reddy. "Neon technology introduction." In: *ARM Corporation* 4 (2008), p. 1.

[Rin+17] C. A. Rincón, J.-F. Pâris, R. Vilalta, A. M. Cheng, and D. D. Long. "Disk failure prediction in heterogeneous environments." In: *2017 Int'l. Symp. on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. July 2017, pp. 1–7.

[RN03] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Pearson Education, 2003. ISBN: 0137903952.

[RS15] E. Ruijters and M. Stoelinga. "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools." In: *Computer Science Review* 15-16.Supplement C (2015), pp. 29–62. ISSN: 1574-0137.

[RT+15] D. Rossetti, S. Team, et al. *GPUDirect: integrating the GPU with a network interface*. 2015.

[Rus78] R. M. Russell. "The CRAY-1 computer system." In: *Communications of the ACM* 21.1 (1978), pp. 63–72.

[Saa03] Y. Saad. *Iterative methods for sparse linear systems*. Vol. 82. siam, 2003.

[Sar+14] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, D. Grove, A. Shinnar, M. Takeuchi, et al. *X10 Language Specification Version 2.5*. Citeseer, 2014.

[SB16] A. Sîrbu and O. Babaoglu. "Towards operator-less data centers through data-driven, predictive, proactive autonomics." In: *Cluster Computing* 19.2 (June 2016), pp. 865–878.

[Sed46] L. I. Sedov. "Propagation of strong shock waves." In: *Journal of Applied Mathematics and Mechanics* 10 (1946), pp. 241–250.

[SG07a] B. Schroeder and G. Gibson. "Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?" In: *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. FAST '07. San Jose, CA: USENIX Association, 2007.

[SG07b] B. Schroeder and G. A. Gibson. "Understanding failures in petascale computers." In: *Journal of Physics: Conference Series*. Vol. 78. 1. IOP Publishing. 2007, p. 012022.

[SG10] B. Schroeder and G. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems." In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (Oct. 2010), pp. 337–350. ISSN: 1545-5971.

[SG84] A. Spector and D. Gifford. "The space shuttle primary computer system." In: *Communications of the ACM* 27.9 (1984), pp. 872–900.

[Sha06] A. Shan. "Heterogeneous processing: a strategy for augmenting moore's law." In: *Linux Journal* 2006.142 (2006), p. 7.

[SKP06] S. Sur, M. J. Koop, and D. K. Panda. "High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis." In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 105.

[SKT15] M. Soualhia, F. Khomh, and S. Tahar. "Predicting Scheduling Failures in the Cloud: A Case Study with Google Clusters and Hadoop on Amazon EMR." In: *Proceedings of the 2015 IEEE 17th Int'l Conf. on High Performance Computing and Communications, 7th Int. Symp. on Cyberspace Safety and Security, and 12th Int'l Conf. on Embedded Software and Systems*. HPCC-CSS-ICESS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 58–65.

[SLM10] F. Salfner, M. Lenk, and M. Malek. "A Survey of Online Failure Prediction Methods." In: *ACM Comput. Surv.* 42.3 (Mar. 2010), 10:1–10:42. ISSN: 0360-0300.

[Sod15] A. Sodani. "Knights landing (knl): 2nd generation Intel® xeon phi processor." In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE. 2015, pp. 1–24.

[SPW09] B. Schroeder, E. Pinheiro, and W.-D. Weber. "DRAM errors in the wild: a large-scale field study." In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 37. 1. ACM. 2009, pp. 193–204.

[Sri+13] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi. "Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults." In: *Proceedings of the Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013.

[Ste96] G. Stellner. "CoCheck: Checkpointing and process migration for MPI." In: *Proceedings of International Conference on Parallel Processing*. IEEE. 1996, pp. 526–531.

[Str+03] D. Strul, R. B. Slates, M. Dahlbom, S. R. Cherry, and P. K. Marsden. "An improved analytical detector response function model for multilayer small-diameter PET scanners." In: *Physics in Medicine and Biology* 48 (2003), pp. 979–994.

[Sum19] V. A. Suma. "Exploring and Prototyping the MPI Process Set Management of MPI Sessions." Masterarbeit. 85375 Neufahrn Bei Freising: Technische Universität München, 2019.

[SV82] L. A. Shepp and Y. Vardi. "Maximum likelihood reconstruction for emission tomography." In: *IEEE transactions on medical imaging* 1.2 (1982), pp. 113–122.

[SW89] J. Simpson and E. Weiner. *The Oxford English Dictionary*. Oxford, England: Oxford University Press, 1989.

[Tae+10] N. Taerat, C. Leangsuksun, C. Chandler, and N. Naksinehaboon. "Proficiency Metrics for Failure Prediction in High Performance Computing." In: *Int'l. Symp. on Parallel and Distributed Processing with Applications*. Sept. 2010, pp. 491–498.

[TBA13] S. Treichler, M. Bauer, and A. Aiken. "Language support for dynamic, hierarchical data partitioning." In: *ACM SIGPLAN Notices*. Vol. 48. 10. ACM. 2013, pp. 495–514.

[Tho+10] J. Thompson, D. W. Dreisigmeyer, T. Jones, M. Kirby, and J. Ladd. "Accurate fault prediction of BlueGene/P RAS logs via geometric reduction." In: *2010 Int'l Conf. on Dependable Systems and Networks Workshops (DSN-W)*. June 2010, pp. 8–14.

[Tiw+15a] D. Tiwari, S. Gupta, G. Gallarno, J. Rogers, and D. Maxwell. "Reliability Lessons Learned from GPU Experience with the Titan Supercomputer at Oak Ridge Leadership Computing Facility." In: *Proceedings of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: ACM, 2015, 38:1–38:12. ISBN: 978-1-4503-3723-6.

[Tiw+15b] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, et al. "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation." In: *2015 IEEE 21st Int'l. Symp. on High Performance Computer Architecture (HPCA)*. Feb. 2015, pp. 331–342.

[TOP19] TOP500.org. *TOP 500, The List.* 2019. URL: `https://www.top500.org` (visited on 03/17/2019).

[Ure+12] I. A. C. Ureña, M. Riepen, M. Konow, and M. Gerndt. "Invasive MPI on Intel's Single-Chip cloud computer." In: *International Conference on Architecture of Computing Systems*. Springer. 2012, pp. 74–85.

[Vir] Virtuozzo. *Checkpoint/Restore In Userspace*. `https://criu.org/Main_Page`. accessed on 29.06.2019.

[Voo+09] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. "Cost of virtual machine live migration in clouds: A performance evaluation." In: *IEEE International Conference on Cloud Computing*. Springer. 2009, pp. 254–265.

[Wan+08] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. "Proactive process-level live migration in HPC environments." In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press. 2008, p. 43.

[Wan+10] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. "Hybrid checkpointing for MPI jobs in HPC environments." In: *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. IEEE. 2010, pp. 524–533.

[Wan+12] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. "Proactive process-level live migration and back migration in HPC environments." In: *Journal of Parallel and Distributed Computing* 72.2 (2012), pp. 254–267.

[Wan+14] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. "Intel math kernel library." In: *High-Performance Computing on the Intel® Xeon Phi(TM)*. Springer, 2014, pp. 167–188.

[Wat+12] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto. "Online failure prediction in cloud datacenters by real-time message pattern learning." In: *4th IEEE Int'l Conf. on Cloud Computing Technology and Science Proceedings*. Dec. 2012, pp. 504–511.

[WD96] D. W. Walker and J. J. Dongarra. "MPI: a standard message passing interface." In: *Supercomputer* 12 (1996), pp. 56–68.

[WOM14] Y. Watanabe, H. Otsuka, and Y. Matsumoto. "Failure Prediction for Cloud Datacenter by Hybrid Message Pattern Learning." In: *2014 IEEE 11th Intl Conf on Ubiquitous Intelligence and Computing*. Dec. 2014, pp. 425–432.

[Woo+07] T. Wood, P. J. Shenoy, A. Venkataramani, M. S. Yousif, et al. "Black-box and Gray-box Strategies for Virtual Machine Migration." In: *NSDI*. Vol. 7. 2007, pp. 17–17.

[XSC13] Z. Xiao, W. Song, and Q. Chen. "Dynamic resource allocation using virtual machines for cloud computing environment." In: *IEEE transactions on parallel and distributed systems* 24.6 (2013), pp. 1107–1117.

[Xu+10] L. Xu, H.-Q. Wang, R.-J. Zhou, and B.-Y. Ge. "Autonomic failure prediction based on manifold learning for large-scale distributed systems." In: *The Journal of China Universities of Posts and Telecommunications* 17.4 (2010), pp. 116–124. ISSN: 1005-8885.

[Xue+07] Z. Xue, X. Dong, S. Ma, and W. Dong. "A Survey on Failure Prediction of Large-Scale Server Clusters." In: *Eighth ACIS Int'l Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*. Vol. 2. July 2007, pp. 733–738.

[Yan+18] D. Yang, J. Weidendorfer, C. Trinitis, T. Küstner, and S. Ziegler. "Enabling Application-Integrated Proactive Fault Tolerance." In: *PARCO 2017: Parallel Computing is Everywhere*. IOS Press. 2018, pp. 475–484.

[YJG03] A. B. Yoo, M. A. Jette, and M. Grondona. "Slurm: Simple linux utility for resource management." In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[Yu+11] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan. "Practical online failure prediction for Blue Gene/P: Period-based vs event-driven." In: *2011 IEEE/IFIP 41st Int'l Conf. on Dependable Systems and Networks Workshops (DSN-W)*. June 2011, pp. 259–264.

[Yua+12] Y. Yuan, Y. Wu, Q. Wang, G. Yang, and W. Zheng. "Job failures in high performance computing systems: A large-scale empirical study." In: *Computers & Mathematics with Applications* 63.2 (2012). Advances in context, cognitive, and secure computing, pp. 365–377.

[Zha00] G. P. Zhang. "Neural networks for classification: a survey." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 30.4 (Nov. 2000), pp. 451–462.

[Zhe+10] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman. "A practical failure prediction with location and lead time for Blue Gene/P." In: *2010 Int'l Conf. on Dependable Systems and Networks Workshops (DSN-W)*. June 2010, pp. 15–22.

[Zho+14] H. Zhou, Y. Mhedheb, K. Idrees, C. W. Glass, J. Gracia, and K. Fürlinger. "Dart-mpi: An mpi-based implementation of a pgas runtime system." In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM. 2014, p. 3.

[Zhu+13] B. Zhu, G. Wang, X. Liu, D. Hu, S. Lin, and J. Ma. "Proactive drive failure prediction for large scale storage systems." In: *2013 IEEE 29th Symp. on Mass Storage Systems and Technologies (MSST)*. May 2013, pp. 1–5.

[ZN01] H. Zhong and J. Nieh. *CRAK: Linux checkpoint/restart as a kernel module*. Tech. rep. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2001.

[ZSK04] G. Zheng, L. Shi, and L. V. Kalé. "FTC-Charm++: an in-memory checkpoint-based fault-tolerant runtime for Charm++ and MPI." In: *2004 ieee international conference on cluster computing (ieee cat. no. 04EX935)*. IEEE. 2004, pp. 93–103.