universität wien

DISSERTATION / DOCTORAL THESIS

Titel der Dissertation / Title of the Doctoral Thesis

"Exact state reconstruction to recover from node failures in certain iterative linear algebra algorithms"

verfasst von / submitted by

Carlos Andres Pachajoa Mejia

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt/ degree programme code as it appears on the student record sheet:	UA 786 880
Dissertationsgebiet lt. Studienblatt/ field of study as it appears on the student record sheet:	Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wilfried Gansterer, M.Sc.
Betreut von / Supervisor:	Univ.-Prof. Dr. Jesper Larsson Träff, MSc PhD.

Zusammenfassung

Die Verbindung von einzelnen Rechenkomponenten erlaubt die Anwendung großerer Rechenkapazitäten, zu einem Zeitpunkt der Geschichte, in dem die Rechenleistung eines einzelnen Kerns aufgrund von physikalischen Grenzen nicht mehr gesteigert werden kann. Die Zuverlässigkeit der einzelnen Komponenten bleibt allerdings ungefähr konstant, daher nimmt die Zeit zwischen Fehlern der Ansammlung ab. Um diesem Problem in zukünftigen, extrem großen Computerclusters vorzubeugen ist Forschung in Resilienzmethoden erförderlich.

Diese Doktorarbeit befasst sich mit Resilienzmethoden gegen Knotenausfälle für bestimmte iterative Algorithmen der linearen Algebra, die auf der Rekursion von einer endlichen Anzahl von Termen basieren, insbesondere mit dem Verfahren der konjugierten Gradienten. Der zugrundeliegende Gedanke ist die Rekonstruktion des Zustands des Algorithmus –Unter Zustand versteht man die relevanten Variablen der Iterationen– damit er weiter arbeiten kann, als wäre er unbeeinträchtigt.

Wir zeigen die Vorteile von einer *exakten Rekonstruktion des Zustands* (ESR von engl. exact state reconstruction) eines Algorithmus gegenüber eine annäherende oder partielle Rekonstruktion des Zustands auf. In unseren Experimenten führt ESR zu einer schnelleren Konvergenz nach einem Knotenausfall.

Wir bauen auf dem ESR-Ansatz, der von Zizhong Chen eingeführt wurde, auf. Bei diesem Ansatz wird das Matrix-Vektor Produkt erweitert, so dass die Knoten zusätliche redundante Information mit einem geringen Overhead übertragen, die die Rekonstruktion im Falle eines Knotenausfalls ermöglichen. Ein Ersatzknoten kann danach die gespeicherte redundante Information nutzen, um all die sachdienlichen Variablen zu Rekonstruieren, möglicherweiße durch das Lösen eines kleinen, lokalen, linearen Systems.

Wir präsentieren drei Erweiterungen für den ESR-Ansatz. Die erste Erweiterung ermöglicht die Wiederherstellung simultan ausfällender Knoten. Zweitens, wir erweitern den Ansatz, so dass die Rekonstruktion auch ohne Ersatzknoten möglich ist. Drittens beschreiben wir die notwendigen Änderungen, um die Speicherfrequenz redundanter Information zu reduzieren.

Abschließend stellen wir ein Meta-Algorithmus vor, der automatisch mittels der Durchquerung des Abhängigkeitsgraphs ESR-Strategien herstellt. Wir zeigen die Verwendung

dieses Meta-Algorithmus mit drei linearen Algebra Algorithmen: das Verfahren der konjugierten Gradienten, das BiCGStab Verfahren und den Lanczos Algorithmus. Wir schlagen darüber hinaus eine Herangehensweise vor, um die Kondition des lokalen, während der Rekonstruktion zu lösenden linearen Systems zu verbessern.

Abstract

Connecting more individual computing components together enables the use of more processing power on a given task at a point in history when a single core cannot be made faster due to physical limitations. However, since the reliability of the individual components tends to remain constant, the mean time between failures for the ensemble decreases. Mitigating this problem in future, extreme-scale clusters requires research in resilience techniques.

We focus on resilience strategies against node failures for certain iterative linear algebra algorithms based on finite-term recurrences, most prominently the *preconditioned conjugate gradients method*. The main idea is the reconstruction of the *state* of the algorithm, comprising the relevant variables involved in the iteration, such that the algorithm can continue operating as if unaffected.

We illustrate advantages of the *exact state reconstruction* (ESR) instead of performing an approximate or partial state reconstruction. In our experiments, ESR leads to a faster convergence after a node failure.

The ESR approach was first presented by Zizhong Chen and, in this thesis, we build upon it. In this approach, the matrix-vector product is augmented such that nodes additionally communicate, with a small overhead, redundant information that enables reconstruction in the event of a node failure taking place, after which a replacement node can use the stored redundant information to reconstruct all relevant variables, possibly by solving a small local linear system.

We introduce three extensions to the ESR approach. The first extension enables recovery from multiple, simultaneous node failures. The second extension is the ability to recover from node failures *without* the use of spare nodes. For the third extension, we discuss the necessary modifications to reduce the frequency at which redundant information is stored.

Finally, we present a meta-algorithm to automatically produce ESR strategies by traversing the dependency graph of the target algorithm. We illustrate how to use this meta-algorithm with three different target algorithms: the conjugate gradients method, the BICGStab method and the Lanczos algorithm. Additionally, we propose a way to improve the condition of the local linear system to be solved during the reconstruction phase.

Acknowledgments

To Prof. Wilfried Gansterer for his thorough guidance and support.

To Markus and Jesper for all their work.

To Christina, Viktoria and Robert for their contributions.

To Dominik for being my rock.

To Alma for her perspective.

To the Vienna Science and Technology Fund (WWTF), who made this work possible with their funding through project ICT15-113.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

To better explain the problems that this thesis deals with, we begin by briefly introducing some of the vocabulary we use. A *failure* is an event when the application deviates from the correct behavior, an *error* is the change of the state of the program that causes the failure, and a *fault* it the underlying cause of the error. Furthermore, we use the concepts of *error detection* and *error correction*. The former is the problem of finding out if an error has taken place, for example, through the use of parity bites in memory, and the latter is the problem of bringing the application back to a correct state. We classify hardware faults in computing systems in two large categories. On the one hand, we have *soft faults*, where bits in memory are flipped to incorrect values, and might cause failures and application errors to occur. One the other hand, we have *node failures*, where a node of a computer cluster stops working and the contents of its memory become inaccessible. We can also distinguish between permanent and transient node failures. In the former, the node is effectively unavailable during the lifetime of the application, while in the latter, the node starts working again after a period of time and the application can continue using it, along with the application data stored in its memory. Our terminology is explained in more detail in Section 1.4.1. This thesis focuses exclusively on strategies to provide resilience against permanent node failures.

We present methods to allow some iterative linear algebra algorithms based on a finite-term recursion, to recover efficiently from node failures in non-reliable computer clusters. We work most with the *preconditioned conjugate gradients* method, but we also present strategies for the conjugate gradients method without preconditioning, the BiCGStab method and the Lanczos algorithm in Chapter 7. In the following sections we lay out the problem, motivate the necessity for resilience in linear algebra, and explain our notation and terminology.

1.1 Resilience in computational linear algebra for large-scale computers

As computers grow larger in scale and become more parallel in order to satisfy the growing demand of computing power, the likelihood of a failure occurring in some component of the machine in a given window of time is increased. That is to say, if the mean time between failures for a component in a parallel computer remains constant, increasing the number of components reduces the mean time between failures for the ensemble. This, of course, represents a problem for computational linear algebra as well, with very large problems requiring the use of computer clusters to obtain a solution in a reasonable amount of time, or just to provide space for the required variables in the combined memory of many computational nodes.

Brief overview of existing solutions

There are existing techniques addressing the problem of reliability at different levels, be it the hardware, system or algorithmic levels, or a mixture of them. In this section, we discuss general categories of resilience methods. The state of the art in research on resilience for linear algebra algorithms is presented in Chapter 2.

At the hardware level, there are *error correction code* (ECC) hardware modules [Tom+17] in memory chips that mitigate, to an extent, the effects of spontaneous bit flips. They operate with dedicated hardware and the use of additional memory to check the integrity of the stored data. ECC mechanisms can also perform error correction. However, these incur an overhead both in terms of physical space in the hardware, as well as of the electric power necessary for their operation [Baj+07].

At the system level, an approach to mitigate soft failures is operation replication, such as in the case of *triple modular redundancy* [LV62], in which the computation is repeated three times, and the result is decided by voting from the different branches. This approach makes an undetected failure unlikely, but the overhead is obviously very large because the work has to be performed three times. If a fault-detection scheme is in place, resilience can be provided by *checkpoint/restart*, where the application stores the state of the program periodically, and if a failure is detected, it can revert to a previous, valid state [Egw+13; KK20]. These strategies are easy to apply for a wide range of algorithms, but they are less scalable for very-large-scale machines.

At the algorithmic level, resilience approaches exist that are tailored to exploit the way the algorithm operates to protect it from certain classes of errors. For example, the *algorithm-based fault tolerance* (ABFT) for matrix-matrix multiplication [HA84] can detect and correct bit flips of entries in the result by using checksum vectors and the properties of this linear algebra operation. These strategies efficiently provide resilience, but the set

of algorithms they can protect is very narrow.

There are also strategies addressing resilience at the three levels. For example, running sensitive parts of an algorithm on hardened hardware (more failure-resistant, but more expensive to run), and other parts in error-prone hardware, improving the likelihood that the algorithm reaches the correct solution compared to running on error-prone hardware only. Such approach is called *selective reliability* [EHM14; Cap+14; Li+13; SV13].

Regarding the resilience of linear solvers against node failures, there is work that attempts to reconstruct part of the state of the solver, such as the iterand, using different interpolation strategies [Lan+08]. We build on work that attempts to completely reconstruct the state of the *conjugate gradients* method, which is a linear solver for positive-definite matrices that introduce in Sec 3.1. After a node failure, the entire state of the solver is recovered in a reconstruction phase, such that it can continue in the same trajectory as an undisturbed solver. This is done by redundantly storing a small subset of the variables at every iteration [Che11].

1.2 Problem definition

We work with sparse linear algebra algorithms based on finite term-recurrences in a distributed-memory setting. Most often, we work with the *preconditioned conjugate gradients* (PCG) method, which is a Krylov solver used on linear systems defined by symmetric, positive definite matrices [Saa03].

The algorithm's data is distributed, and the problem is solved on N nodes of a computer cluster. Disjoint subsets I_s of consecutive indices are distributed among the nodes when the solver is started, such that their union forms the set of all indices, I. Node s is assigned the rows of the matrix, and entries of the vectors, corresponding to the indices in its subset I_s. This *block row distribution* is common, and is used in prominent libraries such as PETSc [Abh+18]. The scalars used by the algorithm are replicated in all nodes.

Where relevant, we work with a message-passing programming model. We often refer to MPI functions [Mes15] in particular.

We consider a situation in which the computer cluster is unreliable, specifically, that it can suffer node failures in which one or more nodes fail simultaneously. In terms of the linear algebra algorithm, these faults represent the loss of information owned by the affected nodes: blocks of vector entries, matrix rows, and the scalars also residing in the nodes' memory. We only consider the case of permanent node failures, that is, we assume that the affected node or nodes remain inaccessible for the entire lifetime of the application.

We look for extensions and generalizations of exact state reconstruction strategies for iterative linear algebra algorithms, which can recover from node failures and converge to the correct solution of the problem. In this work, the cost metric is measured in terms

of the runtime for the iterative algorithm to converge up to a predetermined tolerance. We often measure performance in terms of the *runtime overhead* of running an algorithm resiliently, or of recovering from a node failure.

In this context, we pose our research questions.

- How can we extend exact state reconstruction methods to work under a wider set of scenarios, including the simultaneous failure of multiple nodes, recovery without the use of spare nodes and maintaining resilience with a small overhead?

- How can we generalize the already existing exact state reconstruction methods to be able to deal with other linear algebra algorithms based on a finite-term recurrence, besides the preconditioned conjugate gradients method?

1.2.1 Scientific gap

Regarding resilience strategies for linear algebra algorithms that fully reconstruct the state, there is space for improvement in the conditions under which the linear algebra algorithm works, and algorithms these strategies can be applied to. They are summarized in the following:

1. The literature, before this work, had considered only the case of a failure of a single node. An analysis for the case of multiple simultaneous node failures and the corresponding experimental data were not available.

2. Can the reconstruction be performed without the use of spare nodes? How efficiently? This would represent a considerable reduction in hardware overhead.

3. What is the behavior of a method which fully reconstructs the state if the frequency at which redundant data is stored decreases? This would reduce the necessary communication during failure-free iteration and thus it could reduce the runtime overhead.

4. So far, resilient methods which fully reconstruct the state after node failures have been designed for solving linear systems of equations. Can this approach be generalized to a larger family of problems?

5. Is it possible to efficiently reconstruct the state without the need to store additional redundant data? For example, by exploiting the A-orthogonality of the search directions and other geometric properties of the conjugate gradients method?

6. What is the influence of the data distribution in the cluster? Should the reconstruction strategies consider the topology and the sparsity pattern of the system matrix?

We address some of these questions in this thesis. Question 1 is the topic of Section 6.1, question 2 is dealt with in Section 6.2, question 3 is explored in Section 6.3 and, finally, Chapter 7 presents our solution to the question posed in question 4.

1.2.2 Assumptions

Here, we collect the assumptions we make when developing and evaluating our algorithms.

1. We assume that there is no available information about when the next node failure will take place. The methods described in this thesis do not need to know this in order to operate.

2. In most of this thesis, we assume that spare nodes are available in the cluster. An exception is the work presented in Section 6.2, which deals with approaches in the absence of spare nodes.

3. We assume that problem-defining information, such as the system matrix, or the right-hand-side vectors of linear systems, are stored securely, and their affected entries can be recovered in the event of a node failure.

4. Our approach builds on top of a fault detection and fault notification framework assumed to be available on a parallel machine. This framework notifies the application in the event of a node failure. In this thesis, we do not discuss solutions to this problem. However, there are frameworks that provide this functionality [Bla+13; Mes17]. We discuss failure detection and node replacement further in Section 6.1.1.

5. Additionally, we assume the availability of functionality to add new nodes to the application, if they are provided by the cluster, and the ability to adjust the communicators to adapt to the new node distribution after a node failure.

1.3 Motivation

Although computers are often thought of as infallible machines, they are error prone. The *mean time between failures* (MTBF) of a personal computer is so large that, for everyday tasks, the infallibility assumption holds well. In a computer cluster, however, with many nodes working simultaneously on a task, the mean time between failures at any component will be reduced. In very large scale machines, the MTBF might be short enough to be an obstacle to run the application correctly [GR17].

So far, we have been protected from the effects of failures by hardware investment, such as more reliable interconnections and error correction codes in memory. However, there exists a trade-off between the reliability of the components and the energy consumption

of the machine, and the power overhead for the added resilience is magnified in very large scale computers [Hel+20].

In this work, we consider approaches to resilience in computer clusters using the properties of the algorithm itself.

1.3.1 Modern parallel supercomputers

Computers are an integral part of the infrastructure of the modern world. Simulation is used extensively in science and engineering, and has paved new ways to study the world.

Computers are, approximately, Turing machines. Any computer can solve the same problem as any other. That is to say, provided with the necessary storage space, a laptop processor can solve any problem that a supercomputer can solve, albeit taking a longer time. The task might demand, however, that the solution is produced in a timely manner. For example, a weather forecast must be produced before its predictions take place. With growing demand for more detailed results and more complex models, the computational requirements to complete these task on time are increased.

For fifty years, Moore's law predicted the increase in transistor density and, directly related to this, also the reduction in transistor sizes. This stems from continuous miniaturization from improved semiconductor manufacturing techniques [Mac11].

To understand the relationship between the size of a processor's transistors and its speed, let's remember that, in a digital computer, transistors fulfill the role of switches. The maximum clock speed of the processor depends on how fast its transistor can switch. In an analogy to mechanical systems, consider that a mechanical switch will be harder to turn the larger and more massive it is. Similarly, an electronic transistor has inertia: a parasitic electrical capacity, and it takes work to change its state. This capacity grows with the dimensions of the transistor. Switching the transistor with a higher frequency, therefore, consumes more power and produces more heat, and is only feasible with the associated miniaturization of transistors and components.

In order to satisfy the need for more computational power, modern machines become more parallel. Instead of solving a problem on a single, faster-running processor with a higher clock frequency, it is sometimes possible to split the problem in some way and have several computers working on it simultaneously, reducing the runtime to reach a solution. Thus, the fastest computers of today are computer clusters, composed of many interconnected nodes. Modern machines are already running in the petascale domain (in this context, petascale means running 10^{15} floating-point operations per second), and their parallelism continues to increase. The history of the power of the fastest supercomputers of the world is compiled in the Top500 list [DMS97; DLP03]. We present this history in Fig. 1.1.

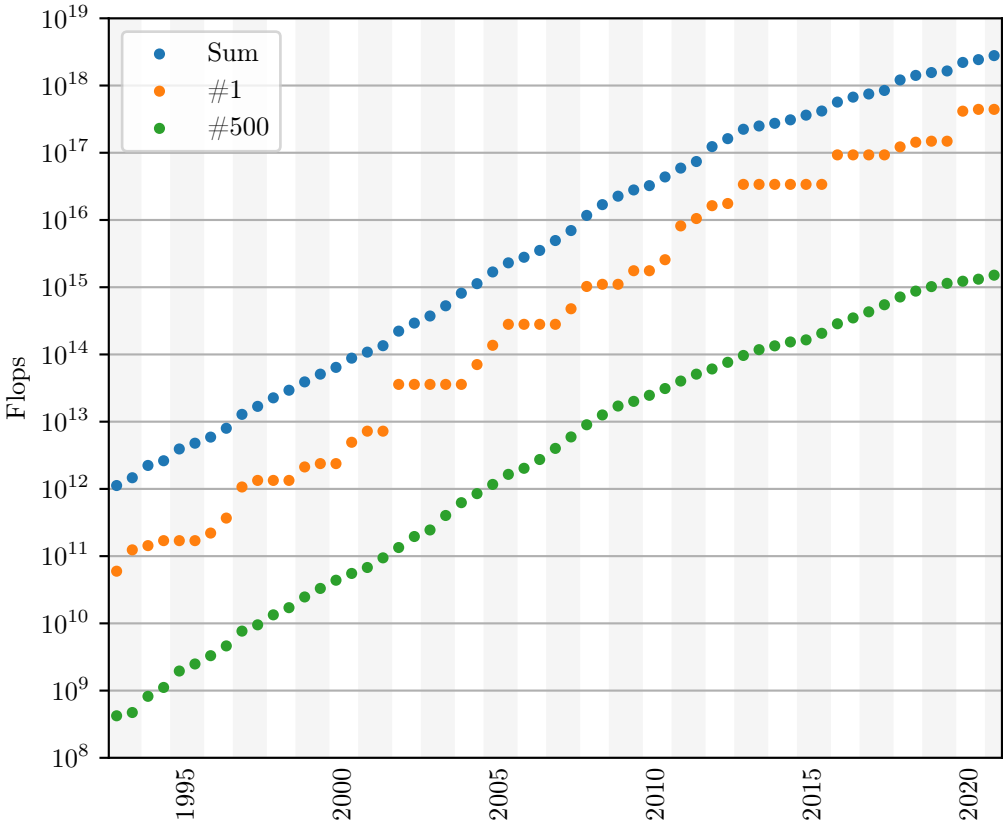


Figure 1.1: Historic data from the Top500 list for performance on the LINPACK benchmark (www.top500.org). The plot showcases the always-increasing capabilities of the fastest computers in the last few decades. The *Sum* data is the combined power of all computers in the Top500 list, while *#1* and *#500* are the power of the first and last supercomputers in the list.

1.3.2 The tyranny of numbers

Early semiconductor-based computers were revolutionary. Instead of a power-hungry vacuum tube, the transistor, invented in 1947 [BBM82], could perform switching using comparatively minuscule power and occupying a tiny fraction of the volume of a tube. Semiconductors opened the door to more powerful machines.

Faster computers were created with increasing counts of transistors and other components. At the time, the computer was assembled by soldering the components on a board. As computers were becoming more complex, the number of soldered joints grew. Discrete components are liable to be defective themselves, but soldering is a particularly error-prone operation. Problems arising from defective components and solder joints became more frequent and there was a level of complexity at which the manufacturing and maintenance costs for these computers were prohibitively high. This was an effective limit for the power of discrete-component semiconductor computers: a phenomenon known as the *tyranny of numbers* [MP58].

Overcoming this limit required a new revolution: integrated circuits, first introduced in 1959. The new manufacturing technique was the *planar process*, which consisted of connecting discrete components in the integrated circuit with metal connections evaporated into the semiconductor surface; it was no longer necessary to solder individual components together [BBM82]. Critically, it dramatically reduced the number of soldered connections necessary. Thus, the limit imposed by the unreliability of the manufacturing process of old was overcome, and it was once again feasible to build faster reliable computers.

Until recently, it was practical to increase the clock frequency of a processor without consuming much more power, and to pack more transistors per unit area in it. Nowadays, though, producing smaller features is difficult. The fastest consumer processors reached nominal clock frequencies of above 4GHz. Clock frequencies have stagnated since [Fra17]. Further miniaturization requires shorter wavelengths from the light source in the manufacturing process to permit the resolution of finer features in the circuit. Presently, the European Union has signed a declaration to develop processors with 2nm lithography[1], with great investments in research and development. However, even if we could form features of arbitrarily small scale, there are physical limits on how small a component can be made before thermal noise becomes too dominant and makes it unreliable [Key88; Kis02] or, because of their tiny size, become too sensitive to manufacturing defects [Sri+04].

Hardware faults on the way to exascale Once again, we are moving towards a point where complexity is imposing a limit on computing power. Even though it is reasonable to

[1]Member States join forces for a European initiative on processors and semiconductor technologies. https://digital-strategy.ec.europa.eu/en/news/member-states-join-forces-european-initiative-processors-and-semiconductor-technologies. Visited on August 9, 2021.

consider a single computing node (think of a desktop workstation) as a reliable machine, a set of hundreds of thousands to millions of cores is liable to fail with relatively short mean time between failures. The community already advocates for considering large-scale computer clusters as unreliable machines [HR15; Sni+14; Dau+17; Tah+21], and research to manage this problem is ongoing.

1.4 Terminology and notation

In this section, we introduce some terminology to address failures in computing, our performance metrics, the way we classify nodes according to their role during a node-failure event, the concepts of *state* and *trajectory*, and the linear algebra symbols used in this work,

1.4.1 Terminology regarding computer failures

In order to address events in which a computer fail to complete correct calculations, we introduce the classification proposed by Avizienis et al. in [Avi+04]:

Failure Occurs when the behavior of the application is not correct. For example, in the case of linear solvers, a failure would represent the inability to obtain the correct solution of the system.

Error The part of the total state of the system that may cause a failure. For example, the spontaneous change of the value of a variable in a linear solver, depending on when and where it happens, it may or may not affect the result.

Fault The declared cause of an error. For example, a bit flip causing the change of a variable in a linear solver.

Soft faults and silent data corruption

Soft faults are spontaneous changes in one or more bits in a computer, whether in memory or in a register. Consequently, they might change the state of the program. If undetected, we talk about or *silent data corruption* (SDC) [Con+08]. This class of faults can be roughly classified into *transient* and *intermittent*. Their description, along the mechanisms that lead to a bit flip, follow:

Transient SDC These are faults that happen at random without a pattern to their location. These can be caused by the impact of neutrons or alpha particles with enough energy to force the change of state of a bit [Bau05].

Intermittent SDC These are faults that occur systematically, for example, with the same bit in some position in memory flipping often. They are caused by physical defects in the hardware, either from manufacturing problems [RPG14] or by sustained damage, like the capacitive breakdown in MOSFET ultra thin oxide layer [Con08].

Integrated circuits are made more vulnerable by further miniaturization and increased number of transistors [Con+08; Shi+02].

If undetected, SDC can potentially cause the application to produce the wrong result, causing the computation to be discarded if the user is fortunate. In the worst case, the user does not notice that the result is wrong and it is accepted as correct. In this context, research focuses not only in error correction; a significant portion of the literature dealing with SDC focuses on error detection schemes.

Off-the-shelf electronics rely on hardware error detection and correction encoding to maintain reliability under the incidence of SDC. However, the introduction of more complex error-correcting schemes has acted as a source of errors in itself [Con+08]. Furthermore, there is a non-zero probability of these mechanisms missing an error [HSS12], and the additional power necessary to use the error-correction features becomes significant in very-large-scale systems [Baj+07].

Computing centers tend to be secretive about the reliability of their machines, but there are still studies about the incidence of SDC in some prominent systems, e.g., [Sni+14] present detailed results for the incidence of soft faults in the Blue Gene/P Solution Intrepid supercomputer.

Node failures

In a computer cluster, a node failure is an event in which a node stops working, and the data contained in it becomes inaccessible.

In tasks where the output of all nodes is required, such event cannot be ignored, since part of the solution is missing, and the user will notice that their application was affected. As such, the literature in this area focuses on error correction, usually the recovery of the variables of the application to a state from which it can produce the correct solution. These failures are, thus, not as insidious as SDC. Node failures have a wide range of causes, such as power fluctuations, issues with the operating system external to the application, and connectivity issues in the cluster [ES13], as well as bugs in the program [DMR21].

We can distinguish between transient and permanent node failures. In the former scenario, the node will become available again, along with its contents, and continue working on the task. In the latter scenario, we consider that the node and its contents will not be available again during the lifetime of the application.

The algorithms described in this thesis deal exclusively with the mitigation of the effects of permanent node failures.

Notice that the term *node failure* does not describe a failure as introduced in the classification at the beginning of Section 1.3.2, where it would be classified as an error. However, given the mainstream use of this expression, we also use it in this work.

We use the term *node failure* to refer to general node failure events taking place at a single point in time. If more specificity is required and we want to distinguish between cases in which a single node or multiple nodes fail simultaneously, we refer to these as *single-node failure* and *multiple-nodes failure*, respectively.

1.4.2 Definition of the runtime overhead

In this work we most often use a metric based on the runtime necessary to reach convergence, the latter being achieved when the relative residual norm of the iterative solver falls below a predetermined tolerance.

We introduce the following quantities:

Reference time (t_0) The time required for a non-resilient iterative algorithm to converge to the solution.

Time to failure (t_1) In a run in which node failures are introduced, it is the runtime at which the failure occurs.

Recovery time (t_r) Runtime necessary to perform the exact state reconstruction and getting the iterative algorithm ready to run again.

Time to convergence after failure (t_2) Runtime from the reconstruction until convergence of the solver.

They are also illustrated in Figure 1.2.

We measure the impact of a node failure with the relative overhead in runtime to reach convergence. We define the following metric:

$$\textit{relative overhead due to a node failure} = \frac{t_1 + t_2 + t_r - t_0}{t_0}.$$

Thus, the relative overhead reflects the additional work required to reach the target relative residual norm in comparison to a non-resilient iterative algorithm.

1.4.3 Node classification according to role in node failure event

We distinguish between different categories of nodes depending on their role during the recovery process:

Lost node This is a node that stops working in the event of a node failure. They become permanently inaccessible and the information contained in them is lost.

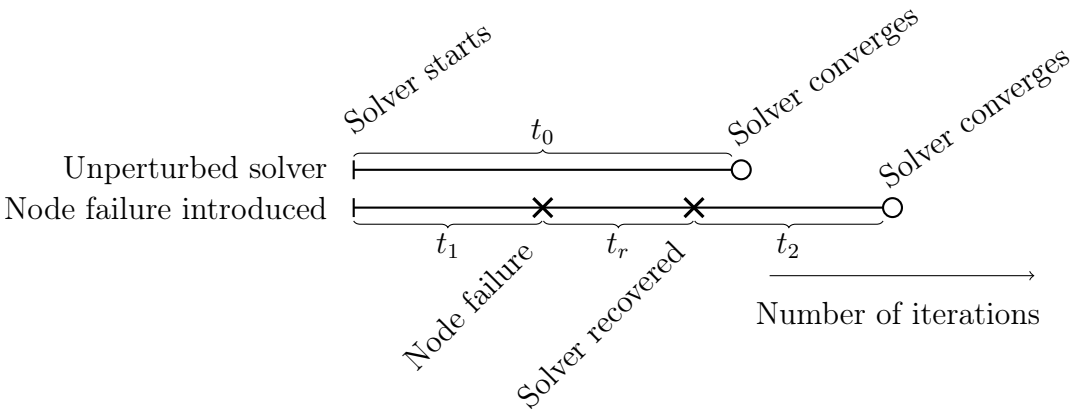


Figure 1.2: Runtime segments used to define overhead metrics for iterative algorithms. The variable t_0 represents the runtime that an unperturbed, non-resilient solver requires to converge, t_1 is the time elapsed from the start until a node failure occurs, and t_2 is the time elapsed from recovery until convergence. The time elapsed during recovery is represented with t_r.

Spare nodes This is a node that remains in standby and can become active if the application suffers a node-failure event, at which point it replaces one of the lost nodes by reconstructing its variables and continues iterating in its place.

Replacement node This is a spare node that, upon the occurrence of a node failure, takes the place of a lost node.

Surviving node This is a node that is still working after a node failure, its information remains accessible, and it also continues working after the reconstruction.

1.4.4 Algorithm categories

We refer to three types of algorithms:

Target / iterative algorithm This is the iterative linear algebra algorithm that we want to protect. In this thesis, our instances of target algorithms are the PCG method the CG method, the BiCGStab method and the Lanczos algorithm.

Exact-state-reconstruction (ESR) algorithm This algorithm is used only in the event of a node failure, and it rebuilds the variables of the target algorithm to the state they were before they were affected by the node failure.

Meta-algorithm In Chapter 7, this refers to a meta-algorithm to generate an ESR algorithm from the dependency graph of the variables of the target algorithm.

1.4.5 State and trajectory of the iterative algorithm

This work is motivated by the idea of reconstructing the *state* of an iterative linear algebra algorithm such that, after a node failure, it continues operating in the same way as an unaffected iterative algorithm would. That is, following the same *trajectory*. We now define the concepts of *state* and *trajectory* more precisely.

State The state of an iterative algorithm refers to the minimal amount of information that completely determines its future behaviour. We exclude static data, that is, problem parameters and information that remains constant for all iterations, from our definition of the state. Instead, the state is composed of dynamic data, which are values that are computed as the algorithm iterates.

Trajectory The trajectory of an iterative algorithm is the sequence of states that it goes through until it reaches convergence. The trajectory of the iterative algorithm is fully determined by its current state.

For example, the PCG solver, as presented in Algorithm 1 in Section 3.1, holds four vectors ($\boldsymbol{x}$,$\boldsymbol{r}$,$\boldsymbol{z}$ and $\boldsymbol{p}$) and two scalars (α and β). A minimal set of data for the state would be the iterand $\boldsymbol{x}$ and the search direction $\boldsymbol{p}$. Then, $\boldsymbol{r}$ can be computed from $\boldsymbol{x}$, $\boldsymbol{z}$ can be computed from $\boldsymbol{r}$, and the scalars α and β can be then found from the vectors by continuing Algorithm 1 from Line 1. The static data for PCG would be the system matrix $\boldsymbol{A}$, the preconditioner $\boldsymbol{P}$, and the right-hand-side vector $\boldsymbol{b}$.

1.4.6 Symbols and linear algebra notation

We employ the notation used in [Che11] to refer to indices of vectors and matrices. We say that a node *owns* an index, and thus the corresponding vector entry and matrix row, if the index is in the set assigned to it.

The set of all indices is referred to as I. For a problem of size n, the cardinality of I is n. A subindex restricts this index set to the indicated node or set of nodes: For example, the entries owned by node s are denoted as I_s. If f is the set of nodes that fail simultaneously, the set of indices corresponding to the lost elements is denoted as I_f, and we can refer to the set of indices for elements in surviving nodes, using the set subtraction operation, as $I \setminus I_f$. We use index sets as subscripts to refer to entries of vectors and matrices, for example, if we have a distributed vector $\boldsymbol{x}$, we refer to the surviving vector elements as $\boldsymbol{x}_{I \setminus I_f}$, and for a matrix $\boldsymbol{A}$, the rows owned by the lost nodes can be written as $\boldsymbol{A}_{I_f,I}$. In the context of the iterative methods, a superscript (as in $\boldsymbol{x}^{(j)}$) indicates the iteration number of a vector or scalar.

The number of nodes in the cluster is denoted with N.

1.5 Synopsis

In this section we briefly present the publications resulting from this work and the contributions of the author of this thesis, and then we present the structure of the rest of this document.

Publications and contributions of the author of this thesis

Here, we list the publications that were produced in this doctoral project, along with a brief description of the work and my contributions to each of them.

Some of the following chapters and sections in this documents are built upon, and thus strongly overlap with these publications. In that case, the publication in question is mentioned at the beginning of the chapter or section.

- Carlos Pachajoa et al. "On the resilience of conjugate gradient and multigrid methods to node failures". In: *European Conference on Parallel Processing*. Springer. 2017, pp. 569–580

  We present this paper in Section 4. My co-authors and I discuss the inherent resilience of multigrid and conjugate gradients methods against node failures. I, in particular, focused on the experimental evaluation.

- Carlos Pachajoa et al. "Extending and Evaluating Fault-Tolerant Preconditioned Conjugate Gradient Methods". In: *2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE. 2018, pp. 49–58

  We present this paper in Section 5.2. My co-authors and I evaluate the ESR approach for the conjugate gradients method and propose extensions of the method to work with differently-formulated preconditioners, and to tolerate the simultaneous failure of multiple nodes. We also compare it with the linear interpolation method from [Lan+08] to recover from node failures. In this publication, I participated in the development of the underlying ideas and produced the experimental results and their analysis.

- Carlos Pachajoa et al. "How to make the preconditioned conjugate gradient method resilient against multiple node failures". In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10

  We present this paper in Section 6.1. My co-authors and I generalize the ESR method to the scenario of the simultaneous failure of multiple nodes, perform an experimental evaluation of the ESR method simulating the simultaneous failure of multiple nodes, and present a model for the runtime overhead of the additional communication required for node-failure resilience.

- Carlos Pachajoa et al. "Node-failure-resistant preconditioned conjugate gradient method without replacement nodes". In: *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE. 2019, pp. 31–40

  We discuss this paper in Section 6.2. This paper presents our strategies for data redistribution used in order to enable recovery using ESR without the need for spare nodes. We also make observations about the impact of the preconditioner in the redistribution process and the reconstruction of the state. I focused on developing the reconstruction strategies and on the evaluation.

- Carlos Pachajoa et al. "Algorithm-based checkpoint-recovery for the conjugate gradient method". In: *49th International Conference on Parallel Processing (ICPP)*. 2020, pp. 1–11

  This paper is presented in Section 6.3. My co-authors and I formulate the ESR approach as a checkpoint/restart strategy, and propose changes that allow the ESR reconstruction for preconditioned conjugate gradients to store redundant information less frequently, and therefore with lower overheads in failure-free situations. I focused on developing a method that stores redundant data less frequently.

- Carlos Pachajoa et al. "A Generic Strategy for Node-Failure Resilience for Certain Iterative Linear Algebra Methods". In: *2020 IEEE/ACM 10th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE. 2020, pp. 41–50

  We discuss this paper in Chapter 7. My co-authors and I present a meta-algorithm to generate ESR approaches automatically. Furthermore, we propose strategies to select a certain submatrix, necessary during the reconstruction process, with a good condition number. I developed of the methods presented in the paper.

The rest of this thesis is structured as follows: We review the state of the art and previous work in Chapter 2. We introduce the algorithmic background for the work in this thesis in Chapter 3.

In Chapter 4 we consider naively restarting different iterative algorithms, including conjugate gradients, by setting the lost entries of the iterand with a strategy and using this vector as an initial guess for a subsequent run. In Section 4.5, we present our argument for attempting to reconstruct the entirety of the state before the algorithm continues iterating.

The exact state reconstruction (ESR) approach, on which we base our work, is presented in Section 5. In Section 5.2, we present our work comparing it to an approach based on approximate reconstruction of the iterand.

We lay out our extensions to the reconstruction algorithm in Chapter 6: the ability to work if multiple nodes fail simultaneously in Section 6.1, the ability to recover in the

absence of spare nodes in Section 6.2, and the reduction of the storage frequency to reduce the failure-free overhead in Section 6.3.

In Chapter 7, we present a generic meta-algorithm to generate ESR reconstruction algorithms to protect iterative algorithms based on finite-term recurrences. Finally, we conclude in Chapter 8.

Chapter 2

State of the art and related work

In this chapter, we visit some of the most relevant previously existing work related to resilience for linear algebra algorithms. Somewhat more classical, general purpose strategies for resilience can be found in [HR15]. A more recent survey by Agullo et al. [Agu+20a], collects resilience techniques for simulation technology. Both of these works have straightforward applications in computational linear algebra.

2.1 Checkpoint/restart

Today, the most used approach to resilience against node failures for iterative algorithms is checkpoint/restart (CR). While an algorithm progresses, it periodically stores its state in secure storage. Depending on the user's choice and the framework used, the checkpoint can contain different sets of data. Usually, the user selects the variables they want to store. The period at which checkpoints are created for the state is called the *checkpointing interval*. In the event of a node failure, the cluster retrieves the lost data from secure storage and can thus continue iterating, losing the work performed in iterations since the last checkpoint. This is called a *restart* in this context. The discussion about checkpoint/restart techniques presented in this section is based on [HR15].

There is a trade-off between the cost of creating checkpoints, when the state has to be copied to some secure location, and the amount of work lost after a node failure. We want to create checkpoints less frequently, pushing us to increase the checkpointing interval, but if a node failure occurs, we want the last checkpoint to be as recent as possible, adding pressure to increase this checkpointing frequency.

Assuming that the node failures occur following a Poisson process with a *mean time between failures* (MTBF) of μ, an optimal checkpoint interval can be determined that minimizes the expected overhead of running an algorithm with resilience. A formula is

derived in [HR15, Eq. (1.9)]. Using their notation, the optimal checkpoint interval is

$$T_{CO} = \sqrt{2(\mu - (D + R))C}, \tag{2.1}$$

where

T_{CO} is the optimal checkpoint interval.

C is the time it takes to make a checkpoint.

D is the downtime, for example, while the cluster allocates replacement nodes.

R is the restart time.

As for checkpointing in parallel, if each component suffers failures following independent Poisson processes with a MTBF of μ_{ind}, a computer cluster with N nodes suffers node failures following a Poisson process with a MTBF of $\frac{\mu_{ind}}{N}$.

Work exists in the specialization of CR strategies for iterative algorithms [Du+20]. We present it separately, as an approach to provide resilience for iterative solvers against node failures in Section 2.4.2.

2.1.1 Coordinated checkpointing

In a *coordinated checkpoint* approach, all nodes in the cluster checkpoint a state simultaneously, and the restart also takes place in all nodes. Working in parallel, the time to checkpoint the state C and the restart time R are functions of N. If the state data is stored externally in secure storage, there are two extremes for the time required to make a checkpoint. On the one side, if the I/O is a bottleneck in this operation, the transfers are serialized. On the other side, if the I/O bandwidth is sufficient, then, for a fixed problem size, an increasing number of nodes, writing in parallel, will take less time to store the checkpoint.

2.1.2 Hierarchical checkpointing

The synchronization of all nodes required to perform coordinated checkpointing reduces the performance of the application. One of the alternatives is *hierarchical checkpointing*. The nodes are separated into groups. Checkpoints are coordinated within these groups, but not with nodes outside of the group.

2.1.3 Asynchronous checkpointing

In asynchronous checkpointing, threads write to intermediate buffers and then, from there, the checkpointed data is moved to secure storage [Sat+12]. This allows the program

to overlap checkpoint creation and computation, increasing the scalability of the resilience scheme. Some authors propose to extract the task of managing the transfer of checkpointed data to secure storage to its own back end process, thus allowing for flexible dynamic spawning of threads to complete the I/O operations. [Nic+19].

2.2 Algorithm-based fault tolerance

Checkpoint/restart, introduced in Section 2.1, can provide resilience to any program, in the sense that it can bring it back to a valid state if and error is detected. Similarly, the hardware error detection and correction codes mentioned in Section 1.4.1 are intended to work with arbitrary applications. In contrast, in this section we introduce specialized strategies that work in a very narrow set of problems but, by exploiting the properties of the algorithms, are more efficient than the more general-purpose approaches.

Algorithm-based fault tolerance (ABFT) was first introduced by Huang et al. [HA84]. There, an approach is proposed that uses the properties of the matrix-matrix product to detect and correct silent data corruption that might occur during the calculation and lead to wrong entries in the result matrix. It works by extending the matrices with checksum vectors, themselves defined by the coefficient vectors $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$. For example, matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ can be extended with an additional checksum row to become $\begin{pmatrix} \boldsymbol{A} \\ \boldsymbol{c}_1^\intercal \boldsymbol{A} \end{pmatrix}$ and $\begin{pmatrix} \boldsymbol{B} & \boldsymbol{B}\boldsymbol{c}_2 \end{pmatrix}$ respectively. The product of the extended matrices would look as follows:

$$\begin{pmatrix} \boldsymbol{A} \\ \boldsymbol{c}_1^\intercal \boldsymbol{A} \end{pmatrix} \begin{pmatrix} \boldsymbol{B} & \boldsymbol{B}\boldsymbol{c}_2 \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}\boldsymbol{B} & \boldsymbol{A}\boldsymbol{B}\boldsymbol{c}_2 \\ \boldsymbol{c}_1^\intercal \boldsymbol{A}\boldsymbol{B} & \boldsymbol{c}_1^\intercal \boldsymbol{A}\boldsymbol{B}\boldsymbol{c}_2 \end{pmatrix} \tag{2.2}$$

Along the product $\boldsymbol{AB}$, this operation also produces checksum vectors for the rows and columns of the product. If the checksums are inconsistent with the entries of the product then we know that an error took place during the operation. Furthermore, since errors are located by identifying discrepancies with the row and column checksum vectors, it is possible to identify the affected entry, assuming that the failure affected a single position of the matrix. This approach can be extended with a careful selection of the entries of the checksum vectors that enables detection of multiple entries with errors [HA84; Wu+16].

ABFT can be used for operations besides the matrix-matrix multiplication, e.g., [WC14; Du+12] present work on matrix factorization, including Cholesky, QR and LU, protected by ABFT. Besides, there are specializations of ABFT for iterative algorithms in sparse linear algebra [Tao+16; SKB12; SKB13]. Applications of ABFT specifically aimed to iterative algorithms in sparse linear algebra are left out of this section and presented in Section 2.3

Further work introduces *ABFT&PeriodicCkpt*, a technique combining ABFT with check-

point/restart to protect against SDC [Bos+14; Bos+15; HR15]. Here, ABFT is used to protect a subset of the operations of an application, while checkpoint/restart provides resilience for its global state. The larger the portion of the program protected by ABFT, the less frequently checkpoints of the state must be produced, and the smaller the overhead for running with resilience is, if we make the very reasonable assumption that correcting an instance of SDC is cheaper than rolling back the application to the last checkpoint.

2.3 Algorithm-based resilience against silent data corruption for iterative linear algebra algorithms

The effects of soft errors on iterative linear algebra algorithms are studied by Bronevetsky et al. [BS08]. In this paper, the authors evaluate several linear solvers, including the conjugate gradients method. Errors are introduced randomly, both in time and in the affected variable and bits, and the effects on the runtime are observed. They conclude that consideration of soft errors and strategies to control them are necessary for success in extreme-scale computing.

There are specializations of ABFT for iterative algorithms in sparse linear algebra. In [Tao+16], Tao et al. propose an encoding scheme for matrix-vector products that does not require verification after every vector-generating operation. In work in [SKB12; SKB13], there is use of checksum vectors with a small, contiguous set of non-zero entries that are used in conjunction of a binary search tree to efficiently locate errors in the sparse matrix-vector product.

Shantharam et al. work on the propagation of silent errors in iterative solvers [SSR11]. The authors also perform numerical experiments, comparing PCG and successive over-relaxation (SOR) solvers, and show that they are affected differently by soft errors, with SOR coming out as less peformant in the absence of faults, but outpacing PCG in the presence of even a single instance of SDC.

Some authors have proposed resilience methods based on the ability to run parts of the code in resilient infrastructure, that is, with a guarantee that the code will produce correct results in an approach called *selective reliability* [EHM14; Cap+14; Li+13]. Such approach can reduce the power overhead of the error correction modules of the memory by reducing the amount of work that must be performed on this hardware. In [SV13], Sao et al. show that most of the operations of a linear solver can be run without protection against silent errors, with the solver still converging to the correct solution: The gradient descent solver can be made resilient by periodically computing the residual on resilient hardware, and the PCG method converges to the solution if we periodically run an entire iteration with correctness guarantees.

Another approach to combat soft errors is the replication of computation. Dichev et al. [DN16] evaluate *dual modular redundancy*, where all computations are replicated in order to detect soft errors. If the work is instead run three times, we have *triple modular redundancy* [LV62], which not only can detect a failure, but correct it through voting among the three branches.

Newer work by Agullo et al. [Agu+20b] studies the effect of soft errors, affecting different bits of the variables, in operations with the system matrix and the preconditioner in the conjugate gradients method. They argue that traditional checksum approaches lack robustness when working in floating-point arithmetic, and introduce feasibility criteria using (a) the deviation of the computed residual from the actual one that arises when working with floating-point arithmetic and (b) the scalar α that should lie within a certain interval. If either of these is violated, it indicates the occurrence of silent data corruption.

2.4 Algorithm-based resilience against node failures for iterative linear algebra algorithms

These are approaches that protect iterative linear algebra algorithms against node failures by exploiting properties of the target algorithm. Therefore, they are not general-purpose approaches and instead are tailored to a narrow set of algorithms, but can reduce the use of resources in comparison, be that memory or runtime.

2.4.1 Interpolation techniques

In [Lan+08], Langou et al. present interpolation techniques for several iterative linear solvers, where the missing entries of the iterand are set to values that reduce the residual norm of the vector. The interpolation to produce new iterand, $\boldsymbol{x}^{(new)}$, looks as follows:

$$\begin{aligned} \boldsymbol{x}_{I\setminus I_f}^{(new)} &= \boldsymbol{x}_{I\setminus I_f}^{(old)} \\ \boldsymbol{x}_{I_f}^{(new)} &= \boldsymbol{A}_{I_f,I_f}^{-1}(\boldsymbol{b}_{I_f} - \boldsymbol{A}_{I_f,I\setminus I_f}\,\boldsymbol{x}_{I\setminus I_f}^{(old)}). \end{aligned} \tag{2.3}$$

The goal is to reduce the effort necessary for the solver to converge from the point of the node failure, starting the solver with the new iterand. This approach also bounds the norms of the residual and the error vectors. If $\boldsymbol{x}^*$ is the solution of the linear system, then we have:

$$\begin{aligned} \|\boldsymbol{x}^{(new)} - \boldsymbol{x}^*\|_2 &\leq \left(1 + \|\boldsymbol{A}_{I_f,I_f}\|_2^2\,\|\boldsymbol{A}_{I\setminus I_f,I_f}\|_F^2\right)^{\frac{1}{2}} \|\boldsymbol{x}^{(old)} - \boldsymbol{x}^*\|_2 \\ \|\boldsymbol{r}^{(new)}\|_2 &\leq \left(1 + \|\boldsymbol{A}_{I_f,I_f}\|_2^2\,\|\boldsymbol{A}_{I_f,I\setminus I_f}\|_F^2\right)^{\frac{1}{2}} \|\boldsymbol{r}^{(old)}\|_2. \end{aligned} \tag{2.4}$$

where $\boldsymbol{r}^{(new)}$ is the residual vector obtained from the iterand after interpolation is performed

for the lost entries.

Building on this idea, the least-squares interpolation approach instead solves a global (all nodes of the application) least-squares problem to produce the new entries of the iterand [Agu+13; Agu+16]. In this case, the reconstruction looks as follows:

$$\begin{aligned} \boldsymbol{x}_{I\setminus I_f}^{(LSI)} &= \boldsymbol{x}_{I\setminus I_f}^{(old)} \\ \boldsymbol{x}_{I_f}^{(LSI)} &= arg\,min_{\boldsymbol{x}_{I_f}} \|(\boldsymbol{b}_{I_f} - \boldsymbol{A}_{I,I\setminus I_f}\,\boldsymbol{x}_{I\setminus I_f}^{(old)}) - \boldsymbol{A}_{I,I_f}\boldsymbol{x}_{I_f}\|_2. \end{aligned} \tag{2.5}$$

where the superscript LSI indicates vectors after the least-squares interpolation.

This approach produces a better bound for the residual norm:

$$\|\boldsymbol{r}^{(LSI)}\|_2 \leq \|\boldsymbol{r}^{(old)}\|_2. \tag{2.6}$$

However, the requirement to solve a global least-squares problem makes this approach costly.

These approaches produce an approximation to the iterand as it was before the node failure took place, even with reductions to the corresponding residual norm. In Section 5.2, we evaluate the interpolation approach of [Lan+08] and compare it to the exact state reconstruction approach.

2.4.2 Checkpoint/restart specialized in iterative algorithms

In [Du+20], Du et al. discuss checkpoint/restart strategies for iterative algorithms. A single iteration of an iterative algorithm is composed of a chain of tasks, which are repeated as multiple iterations are performed. The contribution of this paper is the selection of optimal *schedules* that determine what information is stored and when. The authors illustrate that their approach produces strategies with significantly better performance than checkpointing all tasks.

2.4.3 Exact state reconstruction

This approach attempts to reconstruct the state of a linear algebra algorithm exactly as it was before the node failure happened. The approaches presented in this thesis fall into this category.

There is previous work by Chen [Che11] on the redundancy that can be achieved from the matrix-vector product, and how it can be used to reconstruct the state of an iterative solver, including the preconditioned conjugate gradients method. The research presented in this thesis builds on top of this work.

This topic will be discussed in more detail in Chapter 5.

2.5 Other fault tolerance techniques

In this section we mention some work in fault tolerance that does *not* overlap with our work in order to better delimit the boundaries of the topic of this thesis.

2.5.1 Regarding the detection of node failures

The problem of *detection* of node failures is outside of the scope of this thesis.

Several frameworks to serve error detection are already available. The MPI extension *User-Layer Fault Mitigation* (MPI-ULFM) [Bla+13; Mes17] has functions to detect node failures, and the MPI standard includes functions to perform the required communicator operations. Fenix [Gam+14], a framework that hides the implementation of resilience strategies focused on scientific applications, uses ULFM for node-failure detection. An overview of the resilience possibilities of ULFM is provided in [Los+20].

MPI-Reinit [Lag+16] is an interface for application resilience with checkpoint/restart, well suited for global, non-shrinking restart, where non-shrinking means that the number of nodes working on the task does not change after recovery. Regarding node-failure detection, MPI-Reinit introduces a notification function that is activated by one of the surviving nodes to notify the other nodes in the cluster about the event. In this programming model, the failure is not detected via error codes of the MPI calls since these could be error prone if introduced in large code bases. Instead, the problem of failure detection is managed transparently by an MPI library with Reinit extensions, and all nodes are notified automatically.

Furthermore, the EReinit interface [Cha+20] moves the responsibility of node-failure detection (this includes error detection, propagation–notifying all nodes of the problem–, and providing the replacement nodes to continue working on the task) to a component below MPI called the *resource manager*. In this work, Chakraborty et al. use a modified version of the Slurm manager [YJG03] to realize EReinit. A different interface, *Checkpointable MPI* [Ema+17], applies a similar approach for node-failure detection, using the process manager *PMix* [Cas+18] along with Slurm as the layer below MPI.

Another approach, *Fault tolerance assistant MPI* (FTA-MPI) [Fan+16] introduces the concept of process conversations, and constructs like try/catch blocks for error detection. When a process detects a process failure, they will proceed to the catch block, and after a subsequent consensus operation, all processes will have been notified.

2.5.2 Rumor spreading algorithms

In rumor spreading algorithms, also called *gossip* or *epidemic* algorithms, messages from a node are communicated to its immediate neighbors, with the destination selected randomly.

As such, they are well suited for networks prone to broken connections and lost nodes [JBA15]. An important application of these algorithms is the reduction operation of a quantity across the cluster [CGW19], but they have also been extended to more specific problems in linear algebra, such as orthogonalization [Gan+13].

Katti et al. propose the use of rumor spreading algorithms to detect node failures, and perform the `MPI_comm_shrink` operation by stochastic pinging [Kat+18]. Periodically, all processes ping a random node in the cluster. If a node fails to respond after a timeout, it will be added to the list of nodes known to have failed.

2.6 Cost metrics

As mentioned in Section 1.4.2, we measure the performance of our methods with the runtime overhead, or the number of iterations (or V-cycles for multigrid) that acts as a surrogate of the runtime in some of our experiments. However, there are other cost metrics that are relevant to fault tolerance.

If redundancy is required to provide resilience, the methods will incur in a storage overhead, be it in memory or in external storage. Checkpoint/restart (cf. Section 2.1), for example, requires enough additional space to store the state of the application, which would not be necessary if the problem was to be solved without resilience against node failures.

There is a trade-off between power usage and fault tolerance. Additional power is required to perform restart or reconstruction operations, to send larger or more numerous messages between nodes to redundantly store data, and to use error detection and correction modules in memory. Bounding the power consumption in exascale and extreme-scale computing is a particularly important problem given the societal impact of energy production [Bor10; Sim12; Lia+18].

2.7 Summary

In this chapter, we described the previous work on which our research is based.

We introduced checkpoint/restart, which is a resilience strategy that can protect a very wide range of algorithms and is in practice the most often used approach to achieve fault tolerance in the face of node failures.

We also briefly described algorithm-based resilience strategies, which attempt to be more efficient by exploiting properties of the algorithm they protect. Consequently, they are specialized tools tailored for a given target algorithm. Here, we distinguished between methods to protect against silent data corruption, and methods to protect against node failures.

Additionally, we further delimited the topic of this thesis by discussing other methods for resilience in high-performance computing not directly related to it. We also mentioned some metrics used to evaluate the performance of resilience techniques besides the ones used in our work.

In the next chapter, we introduce the algorithmic background, where we present some better-known tools that we use in our work.

Chapter 3

Algorithmic background

In the last chapter, we described resilience strategies that are less established. That is, with some exceptions, they are documented mostly in research papers. In this chapter we introduce well-known methods used in this work: the ones that appear in textbooks and are taught in classrooms.

3.1 The conjugate gradients method

The conjugate gradients method (CG) is a Krylov method applied to the solution of linear systems of equations $\boldsymbol{Ax} = \boldsymbol{b}$ defined by *symmetric, positive definite* (SPD) matrices. A symmetric matrix $\boldsymbol{A}$ is positive-definite if it fulfills the following condition:

$$\boldsymbol{x}^\intercal \boldsymbol{Ax} \geq 0 \qquad \forall \boldsymbol{x} \neq \boldsymbol{0} \tag{3.1}$$

The CG algorithm is presented in [Saa03], and a very nice introduction to it can also be found in [She94], where CG is introduced as a minimization method for quadratic problems.

In practice, the algorithm is used in conjunction with a preconditioner. The preconditioner $\boldsymbol{M}$ is a constant matrix that resembles the matrix $\boldsymbol{A}$ in some relevant way, such that multiplying its inverse $\boldsymbol{M}^{-1}$ with the matrix $\boldsymbol{A}$ improves the latter's condition number, and thus reduces the number of iterations required to converge to the solution. It is said that the preconditioner should be easy to invert, although often we do not attempt to invert any matrix and work with a linear operator that represents $\boldsymbol{M}^{-1}$ already. We denote the operator of the action of the inverse of the preconditioner, $\boldsymbol{M}^{-1}$, as $\boldsymbol{P}$. The application of the preconditioner incurs in a runtime cost, and therefore there is often a trade off between the overhead of the preconditioner application per iteration and the number of iterations that it saves.

The preconditioned conjugate gradients method (PCG) is shown in Alg. 1.

Algorithm 1 Preconditioned conjugate gradients (PCG) method [Saa03, Alg. 9.1]

$\boldsymbol{r}^{(0)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(0)}, \boldsymbol{z}^{(0)} := \boldsymbol{P}\boldsymbol{r}^{(0)}, \boldsymbol{p}^{(0)} := \boldsymbol{z}^{(0)}$
for $j = 0, 1, \ldots,$ until convergence **do**
  $\alpha^{(j)} := \boldsymbol{r}^{(j)\intercal}\boldsymbol{z}^{(j)} / \boldsymbol{p}^{(j)\intercal}\boldsymbol{A}\boldsymbol{p}^{(j)}$
  $\boldsymbol{x}^{(j+1)} := \boldsymbol{x}^{(j)} + \alpha^{(j)}\boldsymbol{p}^{(j)}$
  $\boldsymbol{r}^{(j+1)} := \boldsymbol{r}^{(j)} - \alpha^{(j)}\boldsymbol{A}\boldsymbol{p}^{(j)}$
  $\boldsymbol{z}^{(j+1)} := \boldsymbol{P}\boldsymbol{r}^{(j+1)}$
  $\beta^{(j)} := \boldsymbol{r}^{(j+1)\intercal}\boldsymbol{z}^{(j+1)} / \boldsymbol{r}^{(j)\intercal}\boldsymbol{z}^{(j)}$
  $\boldsymbol{p}^{(j+1)} := \boldsymbol{z}^{(j+1)} + \beta^{(j)}\boldsymbol{p}^{(j)}$
end for

3.2 Stationary iterative solver: Jacobi iterations

The method of Jacobi iterations consists in splitting the matrix $\boldsymbol{A}$ in two parts, a diagonal matrix $\boldsymbol{D}$, and an off-diagonal matrix $\boldsymbol{E}$ whose diagonal entries are zero. We have $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{E}$, and thus, the linear system becomes $\boldsymbol{D}\boldsymbol{x} + \boldsymbol{E}\boldsymbol{x} = \boldsymbol{b}$. We can also write

$$\boldsymbol{x} = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{E}\boldsymbol{x}) = \boldsymbol{D}^{-1}\boldsymbol{b} - \boldsymbol{D}^{-1}\boldsymbol{E}\boldsymbol{x}.$$

This equation motivates the Jacobi iterations:

$$\boldsymbol{x}^{(j+1)} = \boldsymbol{D}^{-1}\boldsymbol{b} - \boldsymbol{D}^{-1}\boldsymbol{E}\boldsymbol{x}^{(j)} \tag{3.2}$$

Next, we study the convergence of this algorithm. We define $\boldsymbol{x}^*$ to be the vector that solves the linear system: $\boldsymbol{A}\boldsymbol{x}^* = \boldsymbol{b}$. Of course, $\boldsymbol{x}^*$ satisfies

$$\boldsymbol{x}^* = \boldsymbol{D}^{-1}\boldsymbol{b} - \boldsymbol{D}^{-1}\boldsymbol{E}\boldsymbol{x}^*. \tag{3.3}$$

We can now define the error vector for iteration j as $\boldsymbol{e}^{(j)} = \boldsymbol{x}^{(j)} - \boldsymbol{x}^*$. By subtracting Eq. (3.3) from Eq. (3.2), we obtain

$$\boldsymbol{e}^{(j+1)} = \boldsymbol{D}^{-1}\boldsymbol{E}\boldsymbol{e}^{(j)}. \tag{3.4}$$

Thus, this algorithm converges if the spectral radius of the matrix $\boldsymbol{D}^{-1}\boldsymbol{E}$ (the greatest eigenvalue in magnitude) is strictly less than one. This is the case for strictly diagonally dominant matrices ($\boldsymbol{A} \in \mathbb{R}^N$ is diagonally dominant if its entries satisfy $a_{i,i} > \sum_{j \neq i} |a_{i,j}|$).

From Eq. (3.4) we can see that the error of the current approximation is reduced more in components corresponding to eigenvectors of the matrix $\boldsymbol{D}^{-1}\boldsymbol{E}$ of smaller eigenvalues. The rate of convergence of the method depends on the largest eigenvalue of $\boldsymbol{D}^{-1}\boldsymbol{E}$ in modulo, which determines the most slowly vanishing component of the error.

3.3 Hierarchical solver: Multigrid

A multigrid (MG) method creates a hierarchy of problems, from the finest/largest to the coarsest/smallest, such that error components that decay more slowly are mapped to components that decay faster in the coarser problem of the level below. We present MG for the solution of the Poisson equation in a one-dimensional domain. We follow [BHM00] and use the same notation.

The boundary value problem for the Poisson equation follows:

$$\frac{du}{dx} = f, \quad x \in [0,1], \quad u(0) = u(1) = 0. \tag{3.5}$$

Consider that the problem for a given level of the hierarchy is discretized with a uniform grid of size $n = 2l + 1$ for some $l > 0$. This produces a system matrix $\boldsymbol{A}$ generated by the stencil

$$\frac{1}{h^2}[1 \quad -2 \quad 1], \quad h = \frac{1}{n-1}, \quad n = 2l + 1. \tag{3.6}$$

In order to explain the geometric multigrid method for our problem, we present the eigendecomposition of the matrix $\boldsymbol{A}$. The k^{th} largest eigenvalue of this tridiagonal matrix is

$$\lambda_k = 4\, sin^2\left(\frac{k\pi}{2n}\right), \tag{3.7}$$

and the i^{th} entry of the corresponding eigenvector is

$$\boldsymbol{v}_{k,i} = sin\left(\frac{ik\pi}{n}\right). \tag{3.8}$$

Now, we describe the components of a MG solver.

3.3.1 Smoothers

The *smoother* is an operation that reduces the error in *high-frequency* components. For the Poisson equation, they correspond to highly oscillatory components with larger wave numbers, but for more general problems, using *algebraic* multigrid, high-frequency components refer abstractly to error components that disappear when moving to a coarser grid. A variation of the Jacobi iteration, called *weighted* Jacobi, can act as a smoother during the solution of the Poisson problem. The weighted Jacobi iteration is defined as follows:

$$\boldsymbol{x}^{(j+1)} = \omega \boldsymbol{D}^{-1}\boldsymbol{b} + ((1-\omega)\boldsymbol{I} + \omega \boldsymbol{D}^{-1}\boldsymbol{E})\boldsymbol{x}^{(j)} = \omega \boldsymbol{D}^{-1}\boldsymbol{b} + \boldsymbol{R}_\omega \boldsymbol{x}^{(j)} \tag{3.9}$$

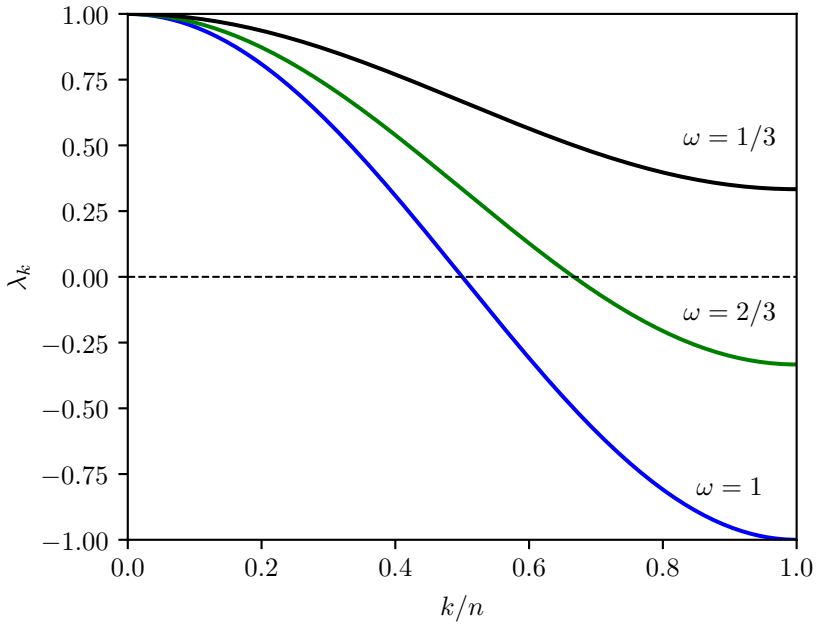


Figure 3.1: Distribution of the eigenvalues of $\boldsymbol{R}_\omega$ for different values of ω.

for some $\omega \in \mathbb{R}$, using the same notation as in Section 3.2. Following steps analogous to the case for Jacobi iterations, we find that the error changes as follows from one iteration to the next:

$$\boldsymbol{e}^{(j+1)} = \boldsymbol{R}_\omega \boldsymbol{e}^{(j)}. \tag{3.10}$$

Furthermore, we can also prove that, for the matrix $\boldsymbol{A}$ generated by the discretization of the Poisson equation in Eq. (3.6), The k^{th} eigenvector of matrix $\boldsymbol{R}_\omega$ is the same as the k^{th} eigenvector of $\boldsymbol{A}$, given in Eq. (3.8), and the corresponding k^{th} eigenvalue of $\boldsymbol{R}_\omega$ is

$$\lambda_k = 1 - 2\,\omega\, sin^2\left(\frac{k\pi}{2n}\right). \tag{3.11}$$

Thus, changing ω allows us to select what components of the error will be eliminated faster by bringing their corresponding eigenvalues very close to zero. The spectrum of the matrix $\boldsymbol{R}_\omega$ for different values of ω is plotted in Figure 3.1. Error components corresponding to eigenvalues close to zero are eliminated very fast.

With $\omega = 2/3$, the modulo of eigenvalues corresponding to $k/n \in [0.5, 1]$ is less than $1/3$. Because of this, one half of the error components (those with eigenvalues corresponding to the largest wave numbers), are eliminated fast. Importantly, the rate of decay for these components is bound by $1/3$, no matter how large n is.

3.3.2 Hierarchies of grids and problems

A hierarchy of grids is build, with the finest one corresponding to the full resolution desired in the solution, going down to the coarsest level where the grid is has so few points that it is cheap to compute the exact solution. We set the lowest, coarsest level, $l = 0$, to consist of two boundary points and one point in the middle of the domain, and the highest level to be $l = L$. The grid at level l consists of $n = 2l + 1$ points uniformly distributed in $[0, 1]$, including two points at the boundaries of the domain. This way, a segment between grid points of the coarse grid corresponds to the union of two segments between points of the grid a level above, and makes for simple value transfer operations between levels (these operations are called *interpolation* and *restriction*, and are explained in the next subsection).

The matrix $\boldsymbol{A}_l$ that discretizes the problem at level l is defined by the stencil presented in Eq. (3.6).

3.3.3 Restriction and interpolation

Restriction is the mapping of data from a fine grid down one level to a coarse grid. Interpolation is the reverse operation, mapping data from a coarse grid up one level to a fine grid. We denote the restriction matrix as $\boldsymbol{T}_{\text{fine,coarse}}$ and the interpolation matrix as $\boldsymbol{T}_{\text{coarse,fine}}$. In order to map from the coarse grid to the fine grid, we use

$$\boldsymbol{x}_{\text{fine}} = \boldsymbol{T}_{\text{coarse,fine}} \boldsymbol{x}_{\text{coarse}}. \tag{3.12}$$

These two operations are chosen to be the scaled transpose of one another. For example, we can use linear interpolation as the interpolation operation, which we represent as

$$\boldsymbol{T}_{\text{coarse,fine}} = \begin{pmatrix} 1 & & & & & & \\ 1/2 & 1/2 & & & & & \\ & 1 & & & & & \\ & 1/2 & 1/2 & & & & \\ & & 1 & & & & \\ & & & \ddots & & & \\ & & & & 1/2 & 1/2 & \\ & & & & & 1 & \end{pmatrix}. \tag{3.13}$$

The matrix for the restriction operation would then be

$$\boldsymbol{T}_{\text{fine,coarse}} = \frac{1}{2}\begin{pmatrix} 1 & 1/2 & & & & & & \\ 1/2 & 1 & 1/2 & & & & & \\ & & 1/2 & 1 & 1/2 & & & \\ & & & & & \ddots & & \\ & & & & & & 1/2 & \\ & & & & & & 1/2 & 1 \end{pmatrix}. \tag{3.14}$$

When the restriction operator is applied, error components in the higher frequencies are not represented, while the lower-frequency components are now high-frequency components in the coarser grid.

3.3.4 Putting multigrid together

Multigrid applies a correction to the current iterand that is computed recursively in coarser levels.

At level l, the solver tries to approximate the solution of a linear system

$$\boldsymbol{b}_l = \boldsymbol{A}_l \boldsymbol{u}_l \tag{3.15}$$

As we discussed before, in Section 3.3.1, applying a smoother on the linear system of Eq. (3.15) will rapidly remove high-frequency error components, but will be slower in low-frequency components.

After successive application of a smoother in level l, we have a new approximation to the iterand $\boldsymbol{x}_l$. We delegate removing the error components from $\boldsymbol{x}_l$ to a coarser level by approximating the error:

$$\boldsymbol{e}_l = \boldsymbol{u}_l - \boldsymbol{x}_l. \tag{3.16}$$

If we have the error, then it can be subtracted from the current iterand to obtain the solution, but finding the error is as difficult as solving the system itself. However, we can instead form a linear system for the error by using the residual $\boldsymbol{r}_l$, which is easy to find. Multiplying Eq. (3.16) to the left by $\boldsymbol{A}_l$ produces

$$\boldsymbol{A}_l \boldsymbol{e}_l = \boldsymbol{A}_l(\boldsymbol{u}_l - \boldsymbol{x}_l) = \boldsymbol{A}_l(\boldsymbol{A}_l^{-1}\boldsymbol{b}_l - \boldsymbol{x}_l) = \boldsymbol{b}_l - \boldsymbol{A}_l \boldsymbol{x}_l = \boldsymbol{r}_l \tag{3.17}$$

or in short

$$\boldsymbol{A}_l \boldsymbol{e}_l = \boldsymbol{r}_l. \tag{3.18}$$

We approximate the solution of the system in Eq. (3.18) in a coarser level, by first restricting the residual, delegating the resulting problem to a coarser level, and interpolating the resulting approximation of the error, that we use to correct the current iterand. The interpolation operation introduces high-frequency error components, since a vector in the coarser level cannot represent those components and they are produced through interpolation from low-frequency values. Therefore, the smoother is applied a number of times after the correction. This procedure is performed recursively from the finest grid, down to the coarsest grid.

Putting everything together, a each level completes the following steps:

1. First, the smoother is applied a number of times to the system of Eq. (3.15), reducing the high frequency components of the errors.

2. Then, the residual vector $\boldsymbol{r}_l$ for this system is computed, and it is restricted to the coarser grid. This defines a new linear system at level $l-1$,

$$\boldsymbol{A}_{l-1}\boldsymbol{u}_{l-1} = \boldsymbol{b}_{l-1}, \quad \boldsymbol{b}_{l-1} = \boldsymbol{T}_{l,l-1}\boldsymbol{r}_l. \tag{3.19}$$

3. The linear system for the coarser grid is solved recursively. This produces an approximation to the error in the coarser grid.

4. The error approximation is interpolated to the finer grid:

$$\boldsymbol{e}_l = \boldsymbol{T}_{l-1,l}\boldsymbol{u}_{l-1}. \tag{3.20}$$

5. The resulting approximation to the error, $\boldsymbol{e}_l$, is used as a correction for the iterand at level l:

$$\boldsymbol{x}_l \leftarrow \boldsymbol{x}_l + \boldsymbol{e}_l. \tag{3.21}$$

6. The smoother is, again, applied a number of times in level l to attenuate the error produced by the interpolation.

These steps describe a V-cycle (see [TOS00, p. 46]), called so because, in the recursion, the solver moves all the way down to the coarsest grid, and back again, forming a v-shape in time. These V-cycles are performed in sequence until the residual norm is as small as desired.

Notice that, since the error is being eliminated at a bounded rate for all wave numbers, the error norm decreases, with every V-cycle performed, at a convergence rate that is independent of the size of the system. If we work with a sparse matrix, achieving a given relative residual norm takes a time in $\mathcal{O}(n_L)$, where n_L is the number of grid points in the

finest grid of the system, equivalent to the size of the problem. This makes this method the fastest known way to solve the Poisson equation.

Multigrid methods are not limited to elliptic partial differential equations. They can be applied to other families of problems, such as hyperbolic partial differential equations [Kat91], and can be used on irregular grids and triangulations. Besides geometric multigrid, there are also algebraic multigrid approaches [Fal06] which operate directly of the matrix of the linear system instead of on the grid from the discretization of a physical problem, expanding the set of problems that can be tackled by this family of methods.

3.4 Summary

In this chapter we discussed important tools that we use along this thesis. We introduced the conjugate gradients method and the geometric multigrid method, which are linear solvers.

In the next chapter, we introduce an argument on the benefits of exactly reconstructing the state (that is, all relevant variables) of a linear algebra algorithm in the event of a node failure. Further chapters detail our strategies for exact state reconstruction in detail.

Chapter 4

Restarting iterative solvers

A natural question to ask is *in the event of a node failure, can we just restart the solver, using the approximation of the iterand we had, following some strategy to replace the entries affected by the fault?*

To explore this idea, we compare instances of two classes of solvers to solve the linear system $\boldsymbol{Ax} = \boldsymbol{b}$: a hierarchical method: geometric multigrid, and a Krylov solver: the conjugate gradients method. We also include an experimental comparison between the two. This chapter contains the work published in [PG17] and a motivating argument against restarting CG with incomplete information of the state in favor of a complete state reconstruction.

About measuring the overhead As an exception, in this chapter we do not measure the overhead as described in Section 1.4.2. Instead of the runtime, we focus on the number of iterations or V-cycles for the conjugate gradients method or multigrid respectively.

We first define the following quantities:

Reference iterations / V-cycles (i_0) Number of iterations or V-cycles required by a non-resilient iterative algorithm to converge to the solution in the absence of failures.

Iterations / V-cycles to failure (i_f) In a run with node failures, the iteration or V-cycle number at which the failures is introduced.

Iterations /V-cycles to convergence (i_c) Number of iterations or V-cycles from the repair of the iterand until convergence.

The relative overhead that we use in this chapter is defined as

$$\textit{relative overhead due to a node failure} = \frac{i_f + i_c - i_0}{i_0}.$$

4.1 Hierarchical solver: multigrid

In our experiments, we use multiplicative geometric multigrid (MG), as introduced in Section 3.3. We use a damped Jacobi as a smoother, with a damping parameter $\omega = \frac{2}{3}$, and V-cycles as the recursion strategy. The solution is smoothed two times before restriction and further two times after interpolation. The coarsest grid consists of three grid points and is solved exactly.

With a perturbation to its current approximation, MG still converges to the solution provided that the spectral radius of the system matrix is less than one, the right hand side is restored and additional MG cycles are performed (see [BHM00, p. 17]). This is equivalent to starting the method from a different initial approximation.

We are not concerned with recovery strategies for the coarser levels of MG. In the event of a node failure, we restart from a new approximation of the solution in the fine grid and the information of the coarser grids is recomputed in the solution process.

4.2 Krylov solver: the conjugate gradients method

The PCG method was introduced in Section 3.1. PCG can be seen as an optimization method of a quadratic function [She94].

The trajectory that the Multigrid solvers takes to reach convergence, and consequently the number of iterations it performs to converge, depends only on the iterand $\boldsymbol{x}$. It can be seen from Alg. 1 that this is not the case for PCG, and the search direction vector $\boldsymbol{p}$, which comes from the characterization of PCG as a convex optimizer, also determines the trajectory of the solver on the way to convergence.

After a node failure, reinitialization of the CG solver, with a naive selection of the initial iterand and search direction, can lead to very large overheads, sometimes losing all progress made so far, as we discuss in Section 4.4.

4.3 An additive model for node failures

Suppose that an iterative linear solver holds a vector $\boldsymbol{v}^{(j)}$ at iteration j. After a node failure, the solver reconstructs the solution as the vector $\hat{\boldsymbol{v}}^{(j)}$. The vectors $\boldsymbol{v}^{(j)}$ and $\hat{\boldsymbol{v}}^{(j)}$ can differ only for indices belonging to the node affected by the node failure. We can model the effect of the node failure on the vector as the addition of the error vector $\boldsymbol{nf}$, whose entries are

$$\boldsymbol{nf}_i = \begin{cases} \hat{\boldsymbol{v}}_i^{(j)} - \boldsymbol{v}_i^{(j)} & \text{if index } i \text{ is affected by the node failure} \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

Thus, after a node failure, the solver continues, using the vector $\boldsymbol{v}^{(j)} + \boldsymbol{rf}$.

A multigrid solver reduces the error exponentially and in all eigenvector components simultaneously. Thus, for MG, the spectral composition of the error term is less important, and is only impacted by changes in the residual norm caused by the addition of $\boldsymbol{rf}$.

The response of the PCG solver depends on the composition of the current error vector. If the error vector happens to be the addition of very few scaled eigenvectors of the matrix, the solver will converge in as many steps, but for other, more general cases, the effects of reinitializing with the added error term of the node failure are more difficult to characterize.

4.4 Experimental evaluation of restarting

We experimentally observe restarting strategies for MG and CG, and compare the effects for these two classes of solvers [PG17]. In the case of MG, we work only with the solution of the discretized Poisson equation. For CG, we also use matrices from the SuiteSparse Matrix Collection [DH11]. First we present the setup of the solvers we use, we then discuss our test problems and afterwards we present our results.

4.4.1 Multigrid setup

To ease the application of geometric MG, we use a grid of equally spaced points. The number of grid points is $2^k + 1$, for a given $k \in \mathbb{N}$, such that the number of segments of the domain is 2^k and can be successively divided by two to form coarser grids. Now, we consider the grids of two consecutive levels. We call the one in the higher level the *fine grid*, and the one in the coarser level the *coarse grid*. With this setup, the distance between grid points in the fine grid is $h = \frac{1}{2^k}$, The coarse grid is composed of 2^{k-1} segments and $2^{k-1} + 1$ grid points. We let each node work on an index set of 2^l grid points for a given $l \in \mathbb{N}$, $l < k$ for the fine grid, except for the node in charge of the leftmost index set, which works with $2^l + 1$ grid points. This grid-point distribution is used also for all arrays in the CG method when solving the discretized Poisson equation. We consider multiplicative geometric multigrid with a damped Jacobi smoother with a damping parameter $\omega = \frac{2}{3}$, and V-cycles (see [TOS00, p. 46]) as the recursion strategy. The solution is smoothed twice both before restriction and further twice after interpolation. The coarsest grid consists of three grid points and is solved exactly.

4.4.2 CG setup

We use the CG method to solve the one-dimensional Poisson equation with Dirichlet boundary conditions in a direct comparison with MG.

Furthermore, we use the CG method to solve linear systems defined by matrices from the SuiteSparse Matrix Collection. We use Jacobi and Gauss-Seidel preconditioners, and also CG without preconditioning. They are listed in Table 4.1. Here, the size of the index set of a node is set to $2^{-m} \times$ size of the domain for some $m \in \mathbb{N}$, and the data of the node is aligned such that the leftmost element of the array lies in the position $i \times 2^{-m} \times$ size of the domain, $i \in \{0, 1, ..., 2^m - 1\}$.

4.4.3 Test problems

Discretized Poisson equation

Our first model problem is the one-dimensional Poisson equation with Dirichlet boundary conditions:

$$x : [0, 1] \to \mathbb{R}, \quad \Delta x = f, \quad x(0) = x(1) = 0. \tag{4.2}$$

The equation is discretized using a uniform grid and finite differences, producing the three-point stencil $\frac{1}{h^2}\begin{bmatrix}1 & -2 & 1\end{bmatrix}$, where h is the distance between two neighboring grid points.

We fix the problem size to $2^{16} + 1$ grid points and the right-hand side is taken to be a vector of ones. We examine cases where the elements of the starting vector are either set to zero or drawn from a uniform probability distribution in the interval $[-0.5, 0.5]$.

The problem is then divided into workers that will take a set of grid points of size 2^6, 2^{10} and 2^{14}. Node failures are introduced at the edge of the domain or at its center. We terminate the iteration once the relative residual norm $\|r\|/\|b\|$ is below 10^{-5}, where b is the right-hand side vector resulting from the discretization of f. In each experiment, a single node is lost at a given time step.

We simulate node failures at fixed V-cycles in MG and store the values of the relative residual norm before perturbing the solution. In CG, we introduce failures when the residual reaches the values stored in the MG run. The location of the failed nodes and the number of lost grid points are the same for each case. This way, we can compare the two methods.

More general sparse matrices

For CG, we also experiment with positive-definite, full-rank matrices obtained from the SuiteSparse Matrix Collection [DH11]. Properties of the matrices that we use for these experiments are summarized in Table 4.1.

With the more complex sparsity patterns of these matrices, the effects of preconditioners, such as Gauss-Seidel and Jacobi, is more interesting.

Name	bcsstk28	mhd4800b
Application	Solid mechanics	Magnetohydrodynamics
Rows × columns	4410 × 4410	4800 × 4800
Non-zeros	219024	27520

Table 4.1: Positive definite matrices from the SuiteSparse Matrix Collection [DH11] used in the CG experiments.

All entries of the right-hand side vector are set to one. We consider that the solver has converged when the relative residual norm $\|r\|/\|b\|$ goes below 10^{-15}. Again, starting vectors are set to zero or drawn from a uniform probability distribution in the interval $[-0.5, 0.5]$, and we simulate node failures.

4.4.4 Results

We evaluate these results in terms of overhead in either number of iterations, for the CG solver, or number of cycles, for the MG solver, until convergence.

Poisson problem

Our results for solvers working on the Poisson equation are illustrated in Fig. 4.2, including experiments for all restart strategies, initial guesses, and location and iteration at which the node failure is introduced. Fig. 4.1 depicts one slice of this setting constellation, with variations on the iteration when the node failure is introduced. It shows that node failures cause a spike in the relative residual norm right after they occur. As depicted in Fig. 4.1, such a failure is not significant if it occurs early enough during the solution process, when the error is not dominated by the loss of the information of the failing node, but by the error in the initial approximation.

Fig. 4.2 shows that the relative overhead for MG after a node failure is introduced decreases linearly with the logarithm of the relative residual norm at the point the failure was introduced. The slope of the curves of residual norm as a function of the V-cycle number for MG is almost constant in all the cases. This is related to the reduction in the residual norm at a constant convergence rate depicted in Fig. 4.1. This rate of convergence depends on the eigenvalues of the matrix and the smoothers used. Introducing an error in the MG solver sets the relative residual close to a "ceiling" value that depends on the size of the lost region, and from there, the relative residual norm continues its reduction at the same rate as before.

The relative residual norm at the start of the experiments is smaller in scenarios where the initial guess is a zero vector, so the corresponding plots do not display failures introduced when the residual norm is larger. In the case of the Poisson problem, multiplying

the matrix of the problem times a vector produces a vector with amplified high-frequency components. Therefore, we can expect this difference, because a starting vector composed of random values as we build it has a lot of its energy in the high frequencies.

The relative overhead of MG is very high in experiments with a zero initial guess and where lost data is filled with zeros. This can be explained, again, with the spectrum of the matrix: Rewriting a region of the iterand with zeros adds high-frequency components that visibly increase the residual norm, above the residual is the initial guess is a vector of zeros. The time required to converge at a constant convergence rate is consequently longer than for the unperturbed solver.

For MG, the influence of the location of the failure is important only if the subdomain is reconstructed by filling the corresponding entries with zeros. If linear interpolation is used, there is little variation in the curves between the left and the right columns of the plot. For the Poisson equation, if the information of the lost nodes is reconstructed using linear interpolation and if the right-hand side is constant, the location of the lost subdomain does not affect the overhead for the Poisson problem. We present a proof of this in Theorem 2 in Appendix A. In general, the larger the region where the data is lost, the greater the overhead.

The CG curves in Fig. 4.2 show very little variation for different sizes of the lost subdomain. In the experiments where the initial approximation is zero the behavior is similar for variations in all other parameters: The relative overhead is close to two (meaning that it takes about three times the number of iterations of the unperturbed solver to converge), and the relative residual norm remains in a narrow interval before the solver suddenly converges.

Other test problems

To test the influence of the preconditioner on the CG method, we run experiments using the Krylov solver facilities of PETSc. Summaries of the results are presented in Fig. 4.3, showing only results for the cases with a random initial guess.

Our first observation is that, for the two matrices, the overhead remains under one in all experiments. We see for matrix `mhd4800b` that, as the node failure moves to the center of the domain, performing linear interpolation actually increases the overhead of the solver. Also, when the node failure happens close to the center, the smallest overhead results from using no preconditioner.

4.5 Is restarting a good solution to node failures?

We formulate the question *is restarting the solver from an approximation to the iterand as it was before the node failure a good way to recover?*. Part of the motivation for the work on

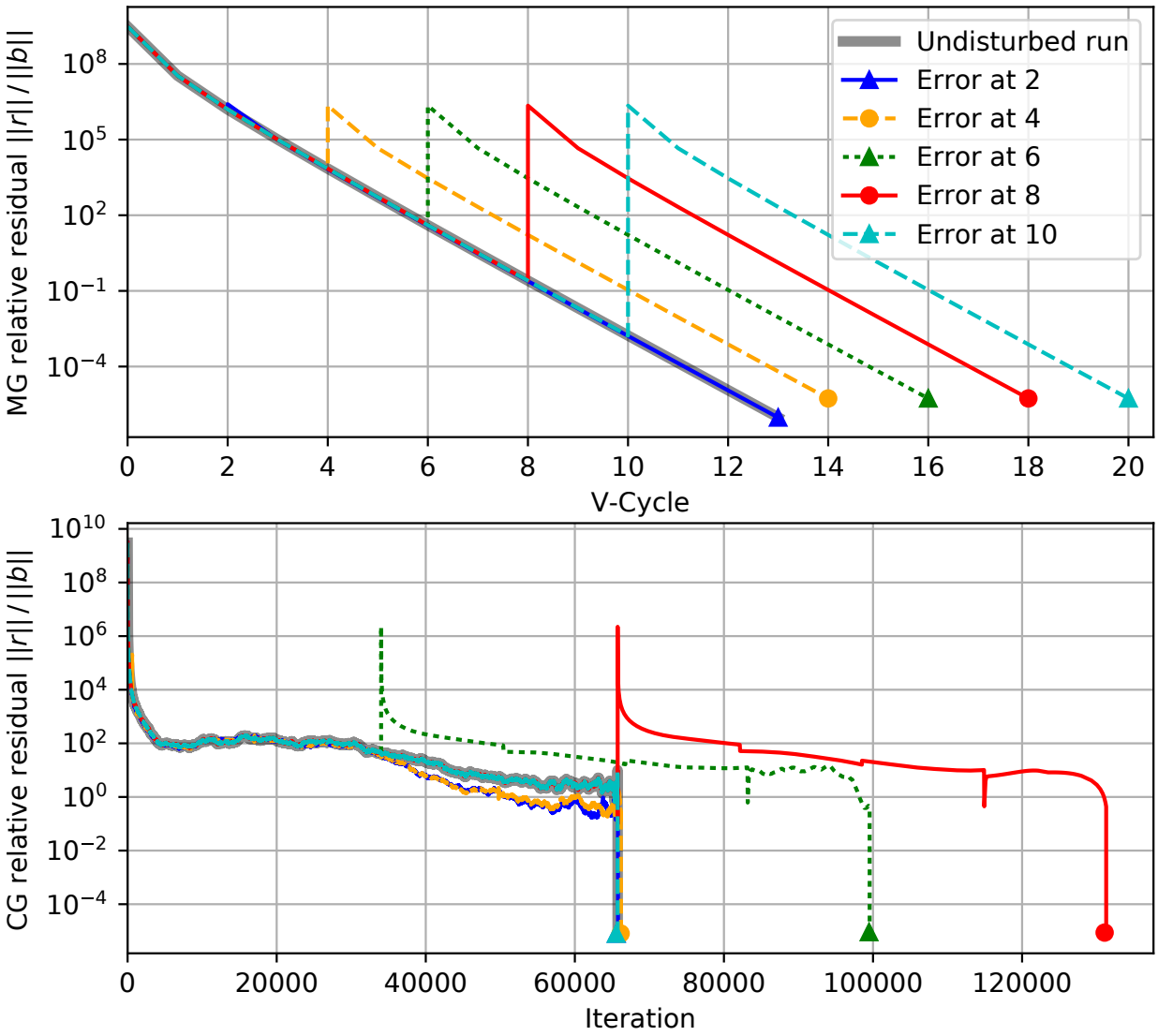


Figure 4.1: A single case of the comparison between MG (above) and CG (below) for a problem size of $N = 2^{16} + 1$. The entries of the initial approximation to the solution are random, following a uniform distribution in $[-0.5, 0.5]$. A region of size 2^{14} is lost at the edge of the domain when the responsible node fails. The recovery strategy is to initialize the values in the affected region with zeros and resume the solver. The introduction of faults appear as spikes in the corresponding curve. To make the plots comparable, failures in the CG plot are introduced once the relative residual reaches the relative residual that MG has reached right before the node failure is introduced. Because of the way the residual is reduced in CG, Consequently, for CG, the curves deviating visibly from the undisturbed run correspond to errors introduced in the V-cycles 6 and 8 in MG. Before that, the CG solver is in still performing the first few iterations and there is no visible overhead. The curve corresponding to a node failure introduced in the V-cycle 10 causes no difference for CG since it converges suddenly before the residual dips under that value.

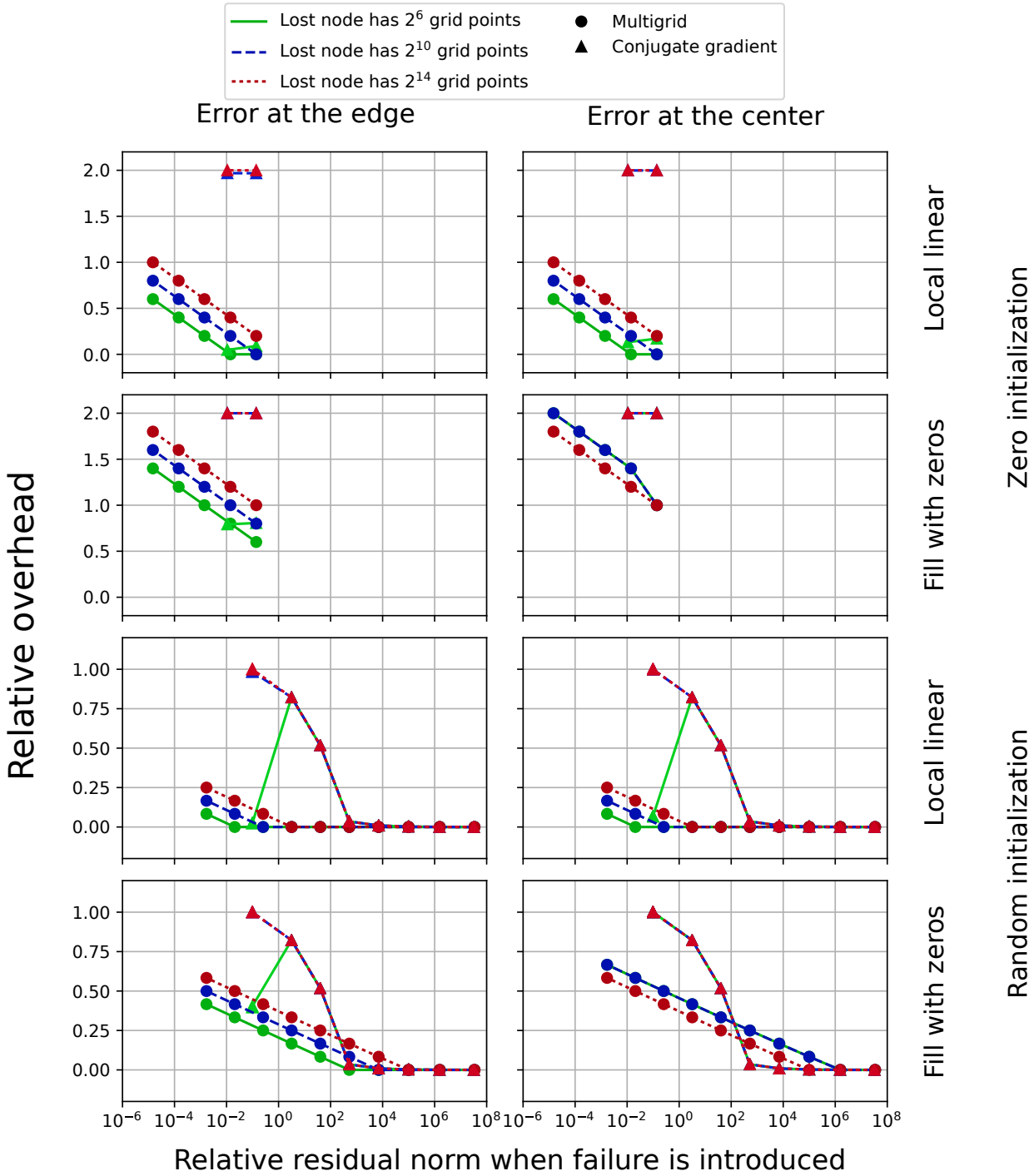


Figure 4.2: Relative runtime overhead of introducing node failures in the Poisson problem after reinitialization of CG and MG. *Fill with zeros* and *Local linear* refer to the reconstruction of the lost values by setting them to zero and performing linear interpolation, respectively. Some of the curves overlap.

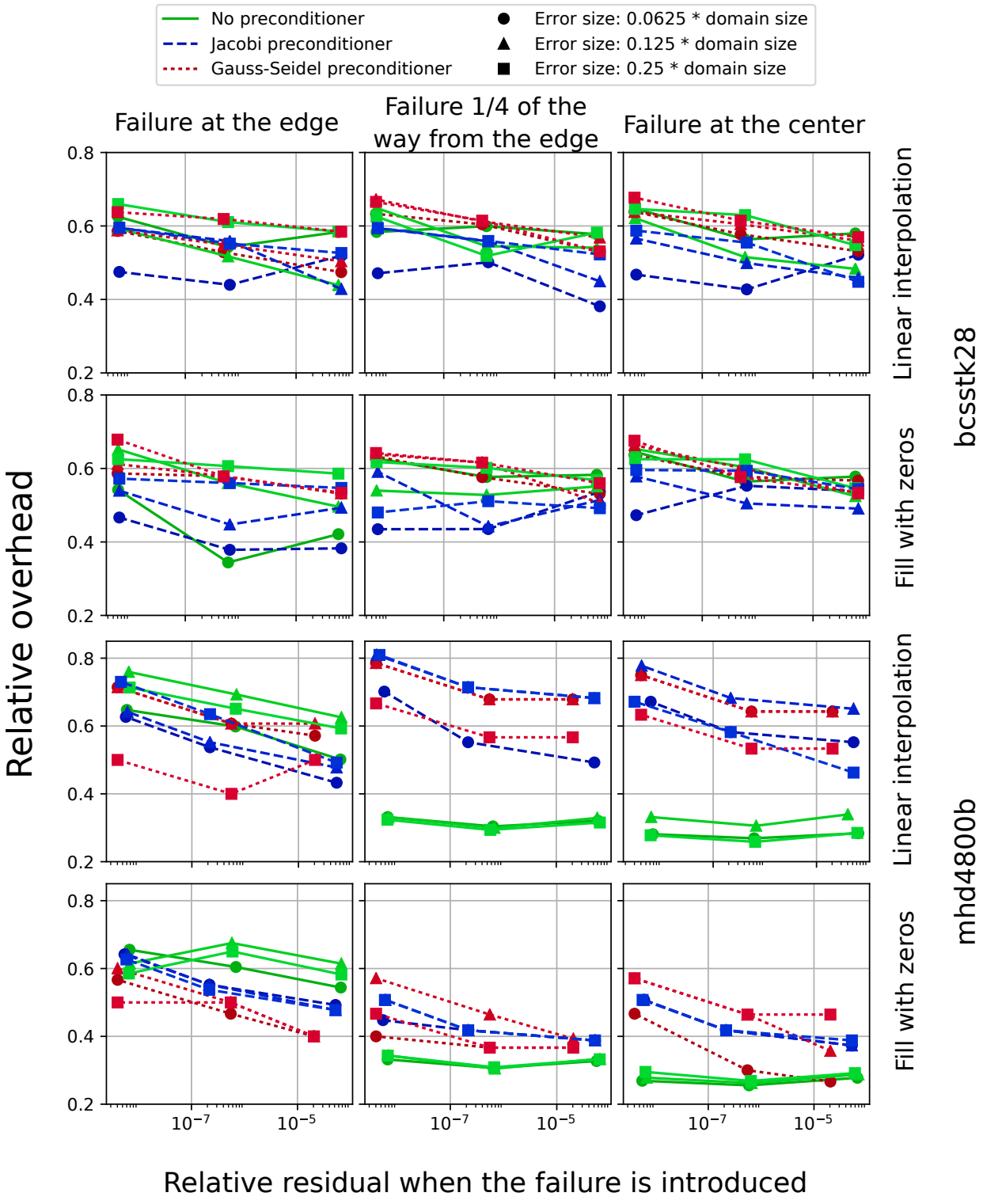


Figure 4.3: Relative runtime overhead after reinitialization of preconditioned CG for the matrices `bcsstk28` and `mhd4800b` in comparison to their reference run, in which no node failures are introduced. All depicted experiments are started with a random initial guess, with values drawn from a uniform distribution in $[-0.5, 0.5]$. *Fill with zeros* and *Linear interpolation* refer to the reconstruction of the lost values by filling the values for the index set with zeros and performing linear interpolation, respectively.

this thesis is the observation that, for a naive selection of the initial guess for the iterand, a problem of size n takes up to n iterations to reach the solution in an exact-arithmetic machine. Thus, restarting the solver with a reconstructed iterand that is not carefully built, even if it is closer to the solution than the original initial guess, might incur very high overheads. As the experiments we performed show, this is also the case in floating point machines.

The experimental results of Section 4.4 support this conclusion. In Fig. 4.1, we see the effect of restarting a multigrid solver and conjugate gradient for a Poisson equation solver with finite differences. In the case of CG, a node failure implies restarting, with up to 100% overhead if the node failure happens almost at the end. Fig. 4.2 shows the relative overhead of CG and MG for node failures introduced once the solvers teach a given relative residual norm. CG is worse off in most cases, both for scenarios with zero initialization of the lost entries, and for initialization using linear interpolation.

Fig. 4.3 shows experiments for CG with matrices from the SuiteSparse matrix collection. The overhead for a node failure is still high, although it drops to approximately 25% for the `mhd4800b` matrix if we do not use a preconditioner.

These experiments suggest that, in terms of relative overhead, restarting a CG solver is not and ideal way to deal with a node failure, even with good approximations to the solution. The state of CG cannot be described with the iterand alone, so, in order to improve the relative runtime overhead, the strategy used should consider more information about the state of the solver. The approach used in this thesis, which is discussed in Chapter 5, consists of reconstructing the entirety of the state of the CG solver (and later, some other iterative algorithms), such that, after a recovery process that takes place if a node failure occurs, the solver can continue iterating basically in the same trajectory as an unaffected solver. This approach greatly reduces the overhead caused by the failure.

4.6 Summary

In this chapter, We investigated some recovery strategies against node failures for the conjugate gradients (CG) and geometric multigrid (MG) methods.

The results for the solution of the Poisson equation gave an idea of the behavior of the solvers after recovering from node failures. For scenarios where a zero vector is the initial approximation to the solution, the closeness of the iterand to the solution had a great impact on the overhead caused by a node failure for the multigrid method, but it barely had an impact for the conjugate gradients method which, on the other hand, was more sensitive to variations of the size of the index set of the lost node.

The relative overhead tended to be smaller if the initial guess is randomized. In this situation, the relative overhead caused by a node failure for the CG solver was not affected

considerably by the size of the lost region, but it was increased if the failure occurs in later stages of the process and was, in general, greater than the relative overhead of MG.

Most importantly, we showed the effects of naive reinitialization on a conjugate gradients solver in the event of a node failure. In our experiments, it was often the case that reinitialization incurred very significant runtime overheads, even if the entries contained in the unaffected nodes were already close to the solution. In the next chapter, we present in detail the exact state reconstruction method (ESR) that overcomes the limitations of a naive restart of the solver.

Chapter 5

Resilience of the preconditioned conjugate gradients method based on exact state reconstruction

In Section 4.5, we discuss how, for PCG, reinitializing the solver is not an ideal way to recover from a node failure, since it might entail a significant overhead in terms of number of iterations and, consequently, in terms of runtime.

In this Chapter, we present an approach to reconstruct the values of the variables of the solver almost exactly (there are deviations that happen when we work on a floating-point computer).

5.1 Components of the exact state reconstruction method

In this section, we describe the main components used to enable the exact reconstruction of the state for a preconditioned conjugate gradients solver, as presented by Chen [Che11].

5.1.1 Inherent redundancy provided by the matrix-vector product: Augmented SpMV

Suppose that we want to compute the product of the row-block distributed matrix $\boldsymbol{A}$ and an analogously distributed vector $\boldsymbol{p}$: $\boldsymbol{\varrho} = \boldsymbol{A}\boldsymbol{p}$. The product, split in rows and columns to highlight the partial products computed by each node i, $\boldsymbol{\varrho}_{I_i}$, is shown here:

$$\begin{aligned}
\boldsymbol{\varrho}_{I_1} &= \boldsymbol{A}_{I_1,I_1}\boldsymbol{p}_{I_1} + \boldsymbol{A}_{I_1,I_2}\boldsymbol{p}_{I_2} + \cdots + \boldsymbol{A}_{I_1,I_N}\boldsymbol{p}_{I_N} \\
\boldsymbol{\varrho}_{I_2} &= \boldsymbol{A}_{I_2,I_1}\boldsymbol{p}_{I_1} + \boldsymbol{A}_{I_2,I_2}\boldsymbol{p}_{I_2} + \cdots + \boldsymbol{A}_{I_2,I_N}\boldsymbol{p}_{I_N} \\
&\vdots \\
\boldsymbol{\varrho}_{I_N} &= \boldsymbol{A}_{I_N,I_1}\boldsymbol{p}_{I_1} + \boldsymbol{A}_{I_N,I_2}\boldsymbol{p}_{I_2} + \cdots + \boldsymbol{A}_{I_N,I_N}\boldsymbol{p}_{I_N},
\end{aligned} \tag{5.1}$$

In order to compute the partial product $\boldsymbol{\varrho}_{I_i}$, node i needs to receive entries from other nodes. If $\boldsymbol{A}$ is a dense matrix, node i receives all the entries belonging to other nodes. If $\boldsymbol{A}$ is sparse, it is no longer necessary that a given node receives all entries from all other nodes in the cluster; The entries a node requires to compute its partial product are determined by the sparsity pattern of the matrix. Since the entries of a given node are being communicated to others in the cluster, the matrix-vector product provides some redundancy for the input vector. A simple illustration of the resulting redundancy is shown in Figure 5.1

Unless the matrix $\boldsymbol{A}$ is dense, whether an entry is sent to another node is determined by the sparsity pattern of the matrix. Thus, in general, the regular sparse matrix-vector product provides only partial redundancy. In order to describe what entries are not transferred to other nodes, we introduce the following sets:

$$\begin{aligned}
S_i &:= \text{all elements of } \boldsymbol{p}_i^{(j)}, \\
S_{ik} &:= \text{elements of } \boldsymbol{p}_i^{(j)} \text{ sent to node } k \text{ computing } \boldsymbol{A}\boldsymbol{p}^{(j)}, \\
R_i &:= \bigcup_{k\in[1..N]\setminus i} S_{ik} \text{ and } R_i^c := S_i - R_i.
\end{aligned} \tag{5.2}$$

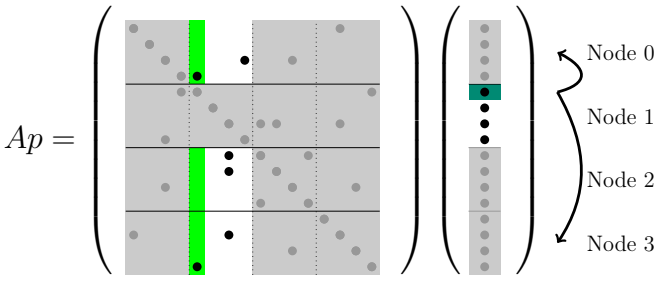
To achieve redundancy of all its entries, node i has to send the entries in R_i^c to a designated node d_i, even though communicating them to any additional nodes is not necessary in order to compute the matrix-vector product.

We refer to this operation, which sends additional information to achieve the desired redundancy, as *augmented sparse matrix-vector product* (ASpMV).

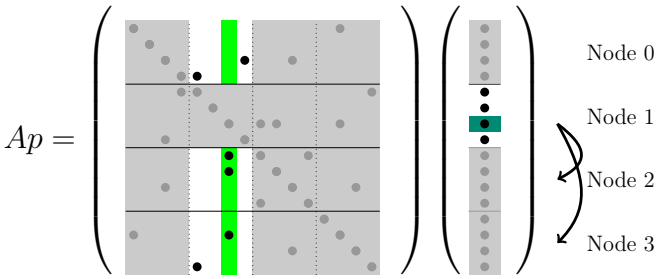
5.1.2 Reconstruction of the state

If some of the vectors involved in the iterations of a linear algebra algorithm can be efficiently stored redundantly, it is feasible to reconstruct the rest of its variables. In [Che11], it is shown how to perform this reconstruction for several types of linear solver, including PCG, using redundantly-stored vectors held from the last few iterations.

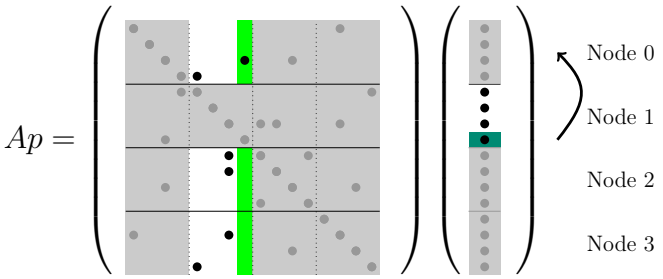
In PCG, shown in Alg. 1, the search direction vector $\boldsymbol{p}$ is involved in a matrix-vector product with matrix $\boldsymbol{A}$ in line 3 of Alg. 1, and can thus be stored redundantly. From line 8 of Alg. 1 we know that $\boldsymbol{p}^{(j)} = \boldsymbol{z}^{(j)} + \beta^{(j-1)}\boldsymbol{p}^{(j-1)}$ must hold. If we can gather the lost parts of last two search directions, $\boldsymbol{p}^{(j)}$ and $\boldsymbol{p}^{(j-i)}$, and provided that we know the scalar



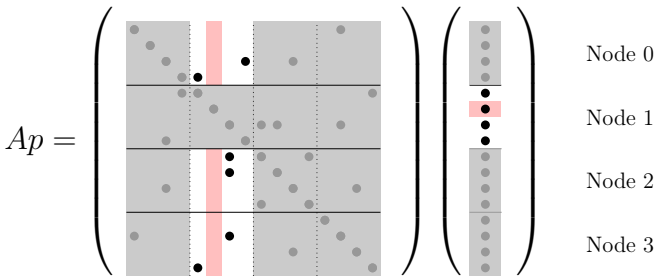
(a) First entry is sent to nodes 0 and 3.



(b) Second entry is sent to nodes 2 and 3.



(c) Third entry is sent to node 0.



(d) Third entry is not sent and would be lost if node 1 fails.

Figure 5.1: Illustration of the redundancy provided by the sparse matrix-vector product with a simple example. Black dots represent non-zero entries. We focus on node 1. Grayed out parts are not relevant to determine where an entry of node 1 is sent to.

$\beta^{(j-1)}$, it is possible to compute the lost values of the preconditioned residual $\boldsymbol{z}^{(j)}$ in a reconstruction node. An important observation here is that the scalar $\beta^{(j-1)}$ is replicated on every node as a consequence of the reduction operations of line 7 of Alg. 1, that is, $\beta^{(j-1)}$ has the same value on all nodes and, in the event of a node failure, it can be sent to the reconstruction node from any of the surviving nodes.

After the preconditioned residual $\boldsymbol{z}^{(j)}$ has been recovered, we can compute the missing parts of the residual $\boldsymbol{r}^{(j)}$ from line 6 of Alg. 1 also in the reconstruction node, and, with the residual vector complete, now we can find the lost parts of the iterand $\boldsymbol{x}$. Recovering $\boldsymbol{r}$ and $\boldsymbol{x}$ involves solving linear systems of equations for the preconditioned residual and the residual respectively. Namely, these systems are $\boldsymbol{P}\boldsymbol{r}^{(j)} = \boldsymbol{z}^{(j)}$ to reconstruct the residual, and then $\boldsymbol{A}\boldsymbol{x}^{(j)} = \boldsymbol{b} - \boldsymbol{r}^{(j)}$ to reconstruct the iterand. However, the entries of $\boldsymbol{r}^{(j)}$ and $\boldsymbol{x}^{(j)}$ owned by surviving nodes are known, and we only need to recover entries owned by the lost node. Suppose that we want to solve the general form of these linear systems:

$$\boldsymbol{B}\boldsymbol{y}^{(j)} = \boldsymbol{c}^{(j)}$$

where the right-hand-side $\boldsymbol{c}^{(j)}$ is a known vector. We can reorder rows and columns of the system of equations and split the system into entries of lost and surviving nodes. Furthermore, we can separate equations for the surviving nodes and the lost nodes:

$$\boldsymbol{B}_{I\setminus I_f,I}\boldsymbol{y}^{(j)} = \boldsymbol{c}^{(j)}_{I\setminus I_f} \quad \text{and} \quad \boldsymbol{B}_{I_f,I}\boldsymbol{y}^{(j)} = \boldsymbol{c}^{(j)}_{I_f}.$$

We can further separate the columns of the latter into entries from surviving and lost nodes, and rearrange the result to obtain a linear system for the lost entries of the vector $\boldsymbol{y}^{(j)}$:

$$\begin{pmatrix}\boldsymbol{B}_{I_f,I_f} & \boldsymbol{B}_{I_f,I\setminus I_f}\end{pmatrix}\begin{pmatrix}\boldsymbol{y}^{(j)}_{I_f} \\ \boldsymbol{y}^{(j)}_{I\setminus I_f}\end{pmatrix} = \boldsymbol{c}^{(j)}_{I_f} \iff \boldsymbol{B}_{I_f,I_f}\boldsymbol{y}^{(j)}_{I_f} = \boldsymbol{c}^{(j)}_{I_f} - \boldsymbol{B}_{I_f,I\setminus I_f}\boldsymbol{y}^{(j)}_{I\setminus I_f}.$$

Thus, we can first reconstruct $\boldsymbol{r}^{(j)}$ using a diagonal submatrix of $\boldsymbol{P}$, and then $\boldsymbol{x}^{(j)}$ using a diagonal submatrix of $\boldsymbol{A}$. These diagonal submatrices have the same size as the number of entries lost during the node failure, and are in general much smaller than the system matrix $\boldsymbol{A}$, thus making it feasible to quickly solve these linear systems. A requirement, however, is that these submatrices are non-singular. As we show with the Corollary 1 in Appendix A, a diagonal submatrix of a positive-definite matrix is itself positive definite and, therefore, non-singular. In the case of the conjugate gradients method, where $\boldsymbol{A}$ is positive definite, the reconstruction of the iterand $\boldsymbol{x}^{(j)}$ is always possible, and care must only be taken in the selection of the preconditioner $\boldsymbol{P}$ to make sure that it is possible to reconstruct the residual $\boldsymbol{r}^{(j)}$.

The reconstruction algorithm executed in a reconstruction node is presented in Alg. 2.

Algorithm 2 Exact reconstruction of the state on the replacement node f for the preconditioned conjugate gradients method (cf. Alg. 1) with the preconditioner $\boldsymbol{P} := \boldsymbol{M}^{-1}$

1: Retrieve the static data $\boldsymbol{A}_{I_f,I}$, $\boldsymbol{P}_{I_f,I}$, and $\boldsymbol{b}_f$
2: Gather $\boldsymbol{r}^{(j)}_{I\setminus I_f}$ and $\boldsymbol{x}^{(j)}_{I\setminus I_f}$
3: Retrieve the redundant copies of $\beta^{(j-1)}$, $\boldsymbol{p}_f^{(j-1)}$, and $\boldsymbol{p}_f^{(j)}$
4: Compute $\boldsymbol{z}_f^{(j)} := \boldsymbol{p}_f^{(j)} - \beta^{(j-1)}\boldsymbol{p}_f^{(j-1)}$
5: Compute $\boldsymbol{v} := \boldsymbol{z}_f^{(j)} - \boldsymbol{P}_{I_f,I\setminus I_f}\boldsymbol{r}^{(j)}_{I\setminus I_f}$
6: Solve $\boldsymbol{P}_{I_f,I_f}\boldsymbol{r}_f^{(j)} = \boldsymbol{v}$ for $\boldsymbol{r}_f^{(j)}$
7: Compute $\boldsymbol{w} := \boldsymbol{b}_f - \boldsymbol{r}_f^{(j)} - \boldsymbol{A}_{I_f,I\setminus I_f}\boldsymbol{x}^{(j)}_{I\setminus I_f}$
8: Solve $\boldsymbol{A}_{I_f,I_f}\boldsymbol{x}_f^{(j)} = \boldsymbol{w}$ for $\boldsymbol{x}_f^{(j)}$

5.2 Comparison between exact state reconstruction and approximate reconstruction using linear interpolation for PCG

In this section, we present the work published in [PLG18], providing, most relevantly, experiments comparing the ESR approach to the already existing linear interpolation (LI) method of [Agu+16]. We compare these two methods measuring the number of iterations until convergence after the recovery.

5.2.1 Resilience to node failures

There are basically three classes of approaches for improving the resilience of iterative linear solvers against node failures (cf. Section 2.1 and Section 2.4). Firstly, there are various types of checkpoint/restart methods, which have in common that they—even in the failure-free case—impose a usually considerable runtime overhead since they continuously need to save the state of the solver [HR15]. Secondly, there are heuristic interpolation/restart methods, which try to reconstruct the most current approximation to the solution vector after a node failure without requiring any overhead during the failure-free execution of the linear solver. Thirdly, there are methods which can exactly reconstruct the complete state of an iterative linear solver without checkpointing its entire state.

The state-of-the-art interpolation/restart methods were proposed by Langou et al. in [Lan+08] and by Agullo et al. in [Agu+16], which consider general Krylov subspace methods in a problem setting similar to ours (cf. Section 1.1) including the delegation of node failure detection and node replacement. Agullo et al. present two algorithms for the reconstruction of a lost part of the iterate after a node failure occurred. After the reconstruction by one of those interpolation methods, the linear solver has to be restarted with the reconstructed iterate as the new initial guess for the solver. The first method, which was

initially introduced by Langou et al., needs to *locally*, i.e., on one node, solve a linear system —much smaller than the original— during the reconstruction process and is thus called the *linear interpolation* (LI) method:

$$\begin{aligned} \boldsymbol{x}_i^{(\mathrm{LI})} &:= \boldsymbol{x}_i^{(j)} \quad \forall i \neq f, \\ \boldsymbol{x}_f^{(\mathrm{LI})} &:= \boldsymbol{A}_{I_f,I_f}^{-1} \left(\boldsymbol{b}_f - \sum_{i \neq f} \boldsymbol{A}_{I_f,I_i} \boldsymbol{x}_i^{(j)} \right), \end{aligned} \tag{5.3}$$

where the superscript (LI) refers to the vectors that have been reconstructed by the LI method after a node failure. The failed and afterwards replaced node is denoted by f, while i designates any given node.

The LI approach requires the diagonal block $\boldsymbol{A}_{I_f,I_f}$ of $\boldsymbol{A}$ to be non-singular, which is fulfilled if $\boldsymbol{A}$ is an SPD matrix (cf. Appendix A, Corollary 1) and, thus, always for the CG and PCG solvers. It can be proven (see [Agu+16]) that the $\boldsymbol{A}$-norm of the forward error $\boldsymbol{e}^{(\mathrm{LI})} := \boldsymbol{x} - \boldsymbol{x}^{(\mathrm{LI})}$ after the reconstruction is smaller than the forward error $\boldsymbol{e}^{(j)} := \boldsymbol{x} - \boldsymbol{x}^{(j)}$ before the node failure occurred, i.e., $\|\boldsymbol{e}^{(\mathrm{LI})}\|_{\boldsymbol{A}} \leq \|\boldsymbol{e}^{(j)}\|_{\boldsymbol{A}}$. In the (mostly hypothetical) case that the surviving nodes already had their parts of the solution $\boldsymbol{x}$ of the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, the LI method would reconstruct the solution of the system on the replacement node f [Agu+16]. In general, the LI method will reconstruct an iterate that is different from the iterate prior to the node failure.

The second interpolation/restart method [Agu+16] needs to *globally*, i.e., on all nodes, solve a least squares problem during the reconstruction phase and is therefore referred to as the *least squares interpolation* (LSI) method:

$$\begin{aligned} \boldsymbol{x}_i^{(\mathrm{LSI})} &:= \boldsymbol{x}_i^{(j)} \quad \forall i \neq f, \\ \boldsymbol{x}_f^{(\mathrm{LSI})} &:= \arg\min_{\boldsymbol{x}_f^{(j)}} \left\| \left(\boldsymbol{b} - \sum_{i \neq f} \boldsymbol{A}_{I,I_i} \boldsymbol{x}_i^{(j)} \right) - \boldsymbol{A}_{I,I_f} \boldsymbol{x}_f^{(j)} \right\|, \end{aligned}$$

where the superscript (LSI) refers to vectors reconstructed using the LSI method. This approach is more general than the LI method since the submatrix $\boldsymbol{A}_{I_f,I_f}$ is not required to be non-singular and only $\boldsymbol{A}$ must have full rank. This significantly increases the communication cost due to the involved global least squares problem but is beneficial for iterative linear solvers that operate on matrices $\boldsymbol{A}$ which are not SPD. However, this higher generality is not relevant for the PCG solver. Thus, we focus on the LI method from now on.

For the LSI method, it holds that $\|\boldsymbol{r}^{(\mathrm{LSI})}\|_2 \leq \|\boldsymbol{r}^{(j)}\|_2$ [Agu+16]. As with the LI method, the correct solution of the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ would be generated if the surviving nodes already contained their parts of the solution. In both cases, the reconstructed iterate guarantees

that the error of the process is monotonically decreasing when measured with the $\boldsymbol{A}$-norm. However, since they are interpolation/restart methods which are agnostic to the iterative linear solver being used, both the LI and LSI methods can only approximate the iterate but are not able to reconstruct the complete state of the solver.

Finally, the ESR approach, proposed by Chen in [Che11], is able to reconstruct all vectors required for continuing without any loss of information at the iteration the node failure happened. During the failure-free PCG iterations, only negligible or—depending on the linear system to be solved—no additional communication is necessary for achieving this. The only overhead required for the recovery of the state of the solver is slightly higher local memory requirements than the non-resilient standard variants and the solution of a small linear system.

At iteration j of the PCG algorithm, the matrix-vector product $\boldsymbol{A}\boldsymbol{p}^{(j)}$ has to be computed (cf. Alg. 1, lines 3 and 5). To illustrate the fundamental concepts, let us first consider the hypothetical case that $\boldsymbol{A}\boldsymbol{p}^{(j)}$ is computed by a dense matrix-vector multiplication kernel. Since we assume a row-wise distribution of $\boldsymbol{A}$, the whole vector $\boldsymbol{p}^{(j)}$ needs to be communicated to all nodes (e.g., by an `Allgather` operation) in this scenario. In the standard non-resilient PCG solver, all but the own block $\boldsymbol{p}_i^{(j)}$ can be dropped on node i after the product has been computed.

In the resilient algorithm, we can also drop most of $\boldsymbol{p}^{(j)}$ on each node after the matrix-vector multiplication. The crucial difference to standard non-resilient PCG is that now each node also stores the block of one of its neighbors in addition to its own block. In particular, this means that there is a redundant copy of every block of $\boldsymbol{p}^{(j)}$ on another node aside from its owner in a round-robin fashion, i.e., the redundant copy of $\boldsymbol{p}_i^{(j)}$ is always kept on node $d_i := i - 1$, or on node $d_0 := N - 1$ if $i = 0$. Hence, in case of a node failure, this redundant copy can be sent to the replacement node in order to completely recover the most recent search direction. We need the two most recent search directions [Che11, Section 5.2], we can keep redundant copies of each block of both $\boldsymbol{p}^{(j-1)}$ and $\boldsymbol{p}^{(j)}$ in a similar manner and, thus, are able to recover the two most recent search directions after a node failure. Therefore, the local memory overhead is $2n/N$ vector elements per node and $2n$ vector elements in total.

In the more realistic case of sparse $\boldsymbol{A}$, it is more efficient to compute $\boldsymbol{A}\boldsymbol{p}^{(j)}$ with a *sparse matrix-vector multiplication* (SpMV) kernel. All the necessary redundant data will be present on another node under the condition that all columns of matrices $\boldsymbol{A}_{I\setminus I_j, I_j}, j \in \{1..N\}$ are non-zero vectors [Che11]. If this condition is not fulfilled, as it is frequently the case, it is necessary to send all the entries of a node, and not only the ones required for the SpMV operation, to some other node. As a consequence, complete redundancy of the vector incurs additional communication and memory during the normal operation of the CG algorithm, and it is desirable to minimize the number of additional entries. For our

simulations, we use the strategy proposed in [Che11], where node i sends the totality of its contents to node $d_i := (i+1) \bmod N$. The procedure to reconstruct the state of the PCG solver is outlined in Alg. 2.

Lines 7 and 8 of Alg. 2 are solving the following equation:

$$\boldsymbol{A}_{I_f,I_f}\boldsymbol{x}_f^{(j)} = \boldsymbol{b}_f - \boldsymbol{r}_f^{(j)} - \boldsymbol{A}_{I_f,I\setminus I_f}\boldsymbol{x}_{I\setminus I_f}^{(j)}, \tag{5.4}$$

where $\boldsymbol{A}_{I_f,I_f}$ has full rank. Thus, by comparing Eq. (5.4) to Eq. (5.3), one can see that the ESR approach can be interpreted as a refinement of the LI method proposed in [Lan+08] and [Agu+16]. However, due to the subtraction of the residual $\boldsymbol{r}_f^{(j)}$, the ESR method not only *approximates* but *exactly reconstructs* the lost part $\boldsymbol{x}_f^{(j)}$ of the iterate. For solving the linear system in Eq. (5.4) locally on the replacement node f, the only communication involved is the gathering of $\boldsymbol{x}_{I\setminus I_f}^{(j)}$. The lost part $\boldsymbol{r}_f^{(j)}$ of the residual $\boldsymbol{r}^{(j)}$ has to be restored before the linear system can be solved.

5.2.2 Experiments

Our experiments run in the REPEAL2 machine of the University of Vienna, in a single NUMA node of 16 CPUs. Simulations are run on 16 processes. We use PETSc [Bal+18; Bal+97] as a framework to perform the experiments. This library was extended with implementations of the ESR and the LI algorithms.

As test cases, we selected the SPD matrices from the SuiteSparse Matrix Collection [DH11] summarized in Tab. 5.1. The right-hand sides of the linear systems are generated randomly using different seeds, normalized relative to the largest entry of the matrix.

We simulate node failures. The rank of the node that fails and the iteration when the failure happens are determined before the program starts. Once the iteration is reached, the reconstruction algorithm is run, representing the work that a replacement node would be required to perform in this event. A real-world application of these algorithms would require the ability to detect node failures, and to produce a replacement node in the event. This functionality is available [Bla+13; Mes17], but we are not using it in this work.

We experimentally investigate the following scenarios:

- Jacobi and block Jacobi preconditioners.
- Nodes with rank 0, 4, 8 and 12 failing.
- Node failures introduced at iterations corresponding to 10%, 30%, 50%, 70% and 90% of the iteration count for the corresponding fault-free case.
- Three seeds to generate the right-hand side.

We also perform fault-free runs for the used random seeds.

The convergence criterion is a residual norm reduction by a factor of 10^5.

Table 5.1: SPD matrices from [DH11] used in the experiments. In experiments with matrix audikw_1 we use only two seeds, instead of three, to generate right-hand sides for the solver.

Name	**Type of problem**	**Size n**	**Non-zeros**
1138_bus	Power network	1138	4054
nos3	Structural	2541	7361
offshore	Electromagnetics	259789	4242673
parabolic_fem	Fluid dynamics	525825	3674625
audikw_1	Structural	943695	77651847
G3_circuit	Circuit simulation	1585478	7660826

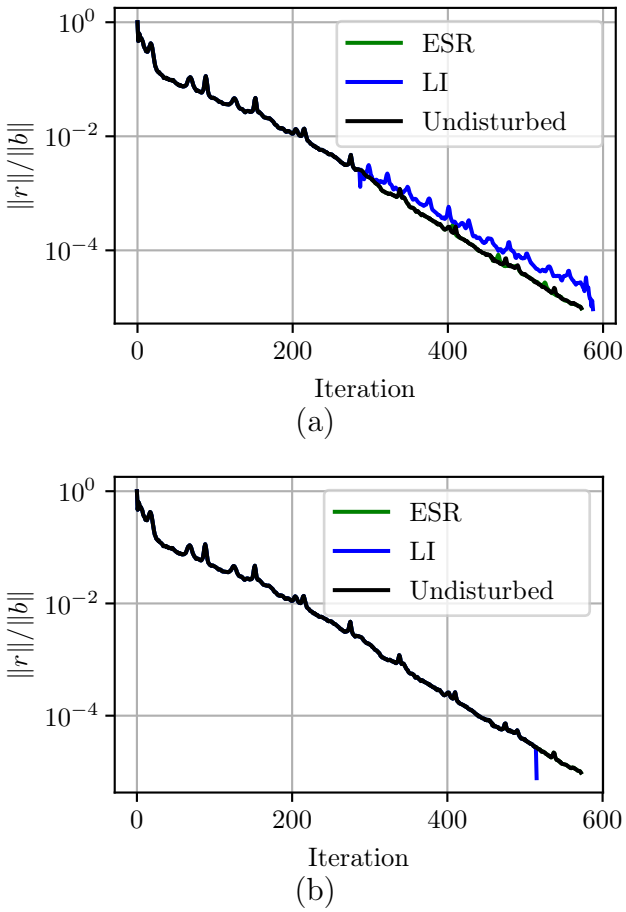


Figure 5.2: Residual norm history for the `offshore` matrix with the Jacobi preconditioner and the node failure happening at 50% (a) and 90% (b) of progress of the solution process. The results for ESR and LI overlap before the node failure. In (b), the residual norm drop of LI after the reconstruction makes it converge immediately in the event of a failure.

Fig. 5.3 show the relative overhead of ESR and LI compared for selected matrices of Tab. 5.1. We make some remarks. First, the relative overhead is, in general, smaller for the ESR method, usually hovering around zero. ESR also has considerably smaller variance, making the cost of the reconstruction process more predictable when compared to LI. Second, the overhead for the LI method tends to be smaller if the node failure happens close to the end of the process. In some cases, the overhead is even negative (cf.Fig. 5.2(b)). The explanation for this is that the residual norm always dips after the interpolation step of LI, as proven in [Agu+16] and as illustrated in Fig. 5.2, sometimes even fulfilling the residual criterion for convergence immediately.

Our results are summarized in Tab. 5.2 and Tab. 5.3, showing the maximum, mean and minimum relative overheads for the Jacobi and block Jacobi preconditioners respectively.

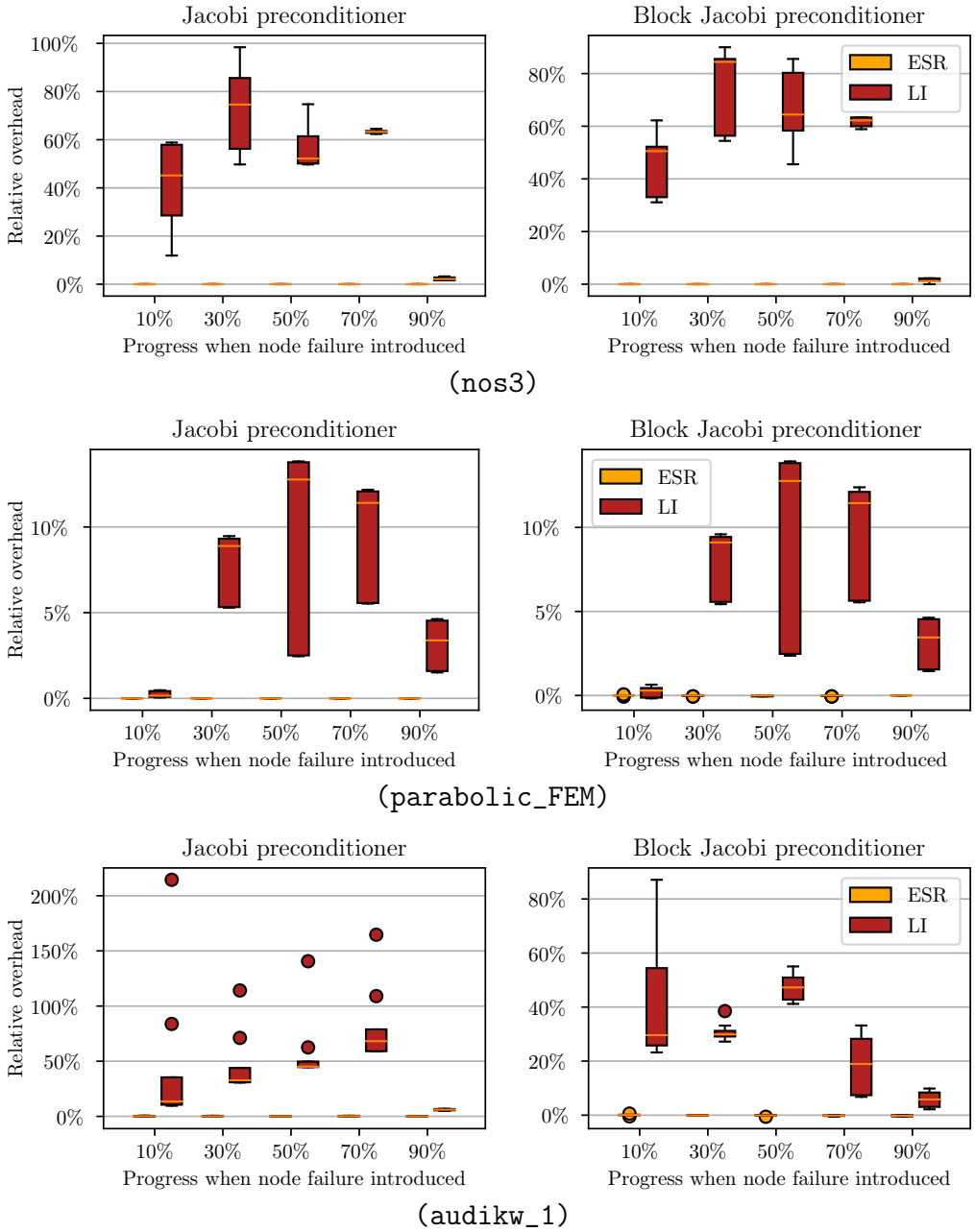


Figure 5.3: Relative overhead bar plot for selected matrices. The boxes contain points between Q1 and Q3. The whiskers extend to the farthest point within 1.5*(Q3-Q1) above and below the boxes. Outliers are represented with points.

Table 5.2: Aggregated results for the relative overhead for the ESR and LI algorithms for the Jacobi preconditioner

		1138	AudiKW 1	G3 Circuit	NOS3	Offshore	Parabolic FEM
Iterations if no failure (Avg)		934.0	6058.5	2009.3	186.0	1292.7	2019.3
ESR	Max	2.04%	0.74%	0.00%	0.00%	0.61%	0.00%
	Mean	0.25%	0.15%	0.00%	0.00%	0.01%	0.00%
	Min	0.00%	−0.35 %	0.00%	0.00%	−0.24 %	0.00%
LI	Max	75.16%	214.46 %	14.99%	98.39%	13.84%	13.83%
	Mean	31.28%	48.25%	4.90%	47.79%	4.32%	6.13%
	Min	10.72%	4.94%	−0.63 %	1.62%	0.15%	0.05%

Table 5.3: Aggregated results for the relative overhead for the ESR and LI algorithms for the block Jacobi preconditioner

		1138	AudiKW 1	G3 Circuit	NOS3	Offshore	Parabolic FEM
Iterations if no failure (Avg)		656.3	2133.0	669.3	90.0	579.0	1772.7
ESR	Max	4.86%	0.58%	0.00%	0.00%	1.24%	0.06%
	Mean	1.51%	−0.10 %	0.00%	0.00%	0.16%	−0.01 %
	Min	−0.31 %	−0.59 %	0.00%	0.00%	0.00%	−0.06 %
LI	Max	82.72%	87.10%	20.03%	90.00%	7.68%	13.93%
	Mean	51.31%	28.85%	3.51%	50.30%	1.18%	6.18%
	Min	10.11%	2.18%	−9.97 %	0.00%	−9.95 %	−0.17 %

5.3 Summary

In this chapter we introduced the exact state reconstruction (ESR) method for the preconditioned conjugate gradients solver, first developed by Chen [Che11], it is the foundation of the work of this thesis. We introduced the basic blocks of the method: an augmented form of the matrix-vector multiplication that allows us to inexpensively transfer redundant data, and the reconstruction algorithm itself, that uses said redundant data to efficiently recover the variables of the target algorithm as they were before the node failure.

Additionally, we presented some initial experiments with the method. We compared the ESR method to the linear interpolation strategies of Agullo et al. [Agu+16], which aim to complete the missing part of the iterand from the system matrix and the entries of the iterand in the surviving nodes. Our experiments showed basically zero overhead in terms of the number of iterations for a solver that recovered from a node failure compared to an unaffected solver, whereas linear interpolation strategies can incur in significant overheads in number of iterations.

In the next chapter, we present our extensions to ESR, including the ability to work in the event of the failure of multiple nodes, to recover without the use of spare nodes in the cluster, and the reduction of the runtime overhead in failure-free cases.

Chapter 6

Extensions of the exact-state-reconstruction approach

This chapter presents our extensions to the exact state reconstruction (ESR) method. The later was explained in detail in Chapter 5.

The proposed extensions are (1) resilience against multiple node failures, (2) recovery without the use of spare nodes and (3) reduction of the frequency of storage of redundant information to minimize the runtime overhead.

6.1 Multiple node failures

We analyze and extend an exact state reconstruction (ESR) approach. In this chapter, we show how to improve the fault tolerance of the PCG algorithm based on the ESR approach. In particular, we support recovery from simultaneous or overlapping failures of several nodes for general sparsity patterns of the system matrix, which cannot be handled by Chen's method [Che11]. For this purpose, we refine the strategy for how to store redundant information across nodes. We analyze and implement our new method and perform numerical experiments with large sparse matrices from real-world applications on 128 nodes of the Vienna Scientific Cluster (VSC). For recovering from three simultaneous node failures we observe average runtime overheads between only 2.8% and 55.0%. The overhead of the improved resilience depends on the sparsity pattern of the system matrix.

This section present work published in [Pac+19].

6.1.1 Problem statement and assumptions

We consider the parallel execution of the PCG solver on N compute nodes of a distributed-memory parallel computer, which communicate over an interconnection network. Each compute node consists of m processors such that the solver is executed on $M := N \times m$

processors in total. Each processor shares its memory with all other processors on the same compute node.

In the event of a *single processor failure*, exactly one processor on one compute node fails, but the shared memory on this node stays intact. Hence, no data is lost and the remaining processors on the node can take over the workload of the failed processor. The same holds for *multiple processor failures* where at least one processor per compute node survives. In this section, we consider a different event: a *node failure*, where a compute node fails as a whole and the data in the memory of the affected node is lost.

Node failures may occur for several reasons: for example, all m processors of a node fail, the shared memory of a node gets corrupted, or a node loses its connection to the interconnection network. In case of a *single-node failure*, exactly one compute node fails at a time. If more than one node fails at a time, we talk about a *multiple-nodes failure*. As discussed in Section 1.4.3, nodes that continue working after a node failure and keep all their data are called *surviving nodes*. A node that becomes unavailable after a node failure is referred to as a *failed node*, and a node that takes the place of a failed node in the recovery process is called a *replacement node* (which could be a spare node or one of the surviving nodes).

Failure detection and node replacement

We assume the availability of a parallel runtime environment that provides functionality comparable to state-of-the-art implementations of the industry-standard *Message Passing Interface* (MPI) [Mes15]. This particularly comprises efficient collective communication capabilities like the `AllReduce` function of MPI. Beyond that, we assume that the underlying runtime environment supports some basic fault-tolerance features. A prototypical example is the MPI extension *User Level Failure Mitigation* (ULFM) [Bla+13; Mes17] which supports the detection of node failures, the prevention of indefinitely blocking synchronizations or communications, the notification of the surviving nodes that a failure has occurred and which nodes have failed, and a mechanism for providing replacement nodes.

Data distribution

Analogously to [Che11], we assume that the fundamental problem-defining static input data, i.e., the system matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, the right-hand-side vector $\boldsymbol{b} \in \mathbb{R}^n$, and the preconditioner $\boldsymbol{P} \in \mathbb{R}^{n \times n}$ (cf. Section 3.1), can be retrieved from reliable external storage (e.g., from a checkpoint prior to entering the linear solver, cf. the *ABFT&PeriodicCkpt* algorithm [Bos+14; Bos+15; HR15], see Section 2.2), and thus does not have to be reconstructed after a node failure.

In accordance with typical distributions of sparse matrices in widely used high-perfor-

mance numerical libraries like PETSc [Bal+97], we consider a block-row data distribution of all matrices and vectors among the N nodes of the parallel computer. More precisely, every node owns blocks of n/N contiguous rows (if $n = cN$ with $c \in \mathbb{N}$, otherwise some nodes own $\lfloor n/N \rfloor$ and others $\lceil n/N \rceil$ rows) of both the matrices $\boldsymbol{A}$ and $\boldsymbol{P}$ as well as each of the vectors $\boldsymbol{b}$, $\boldsymbol{x}$, and all other vectors maintained by the PCG solver (cf. Section 3.1). On one node, the owned block rows, which are stored in the shared memory of the node, are evenly distributed among the m processors, i.e., each processor owns (approximately) n/M rows of each of the matrices and vectors. Globally used scalars are replicated on all N nodes.

Since each node owns block rows of all matrices and vectors, a node failure affects a part of each matrix and vector. Block rows of dynamic data which were owned by the failed node are lost and need to be reconstructed on the replacement node (cf. Section 5).

6.1.2 Extending to multiple node failures

Originally, the ESR method, as described in Section 5, considers exactly one node failure at a time, i.e., it is assumed that the reconstruction process finishes before another node failure occurs (cf. Section 5). For coping with up to $\phi < N$ uniformly distributed node failures that may overlap in time, we need to keep ϕ redundant copies of each block of the two most recent search directions $\boldsymbol{p}^{(j-1)}$ and $\boldsymbol{p}^{(j)}$ on ϕ different compute nodes [PLG18]. In the following, we design a resilient algorithm based on this idea in detail and analyze its communication overhead.

Tolerating multiple node failures

To keep ϕ redundant copies of each block of the two most recent search direction vectors, a similar strategy can be pursued as in the special case $\phi = 1$. We use the notation presented in Eq. (5.2) to describe entry sets. Let (S_i, m_i) be a multiset with the multiplicity

$$\begin{aligned} m_i \colon S_i &\to \mathbb{N}_0 \\ s &\mapsto \text{number of nodes } s \text{ is sent to} \\ &\quad \text{during the computation of } \boldsymbol{A}\boldsymbol{p}^{(j)}. \end{aligned} \tag{6.1}$$

Note that we assume here—as it is common for SpMV—that s is only sent to nodes with corresponding non-zero entries in their rows of $\boldsymbol{A}$. Hence, the number of nodes s is sent to depends on the sparsity pattern of $\boldsymbol{A}$. Comparing the definition of the multiplicity m in Eq. (6.1) with the definition of R_i^c in Eq. (5.2), it follows that

$$R_i^c = \{s \in S_i \mid m_i(s) = 0\}. \tag{6.2}$$

For supporting up to ϕ simultaneous (or overlapping) node failures, we need to store at least ϕ redundant copies of each element of $\boldsymbol{p}_i^{(j)}$, $i \in \{1, 2, \ldots, N\}$, on ϕ different nodes other than node i. Let

$$d_{ik} := \begin{cases} \left(i + \left\lceil \frac{k}{2} \right\rceil\right) \bmod N, & \text{if } k \text{ odd} \\ \left(i - \frac{k}{2}\right) \bmod N, & \text{if } k \text{ even} \end{cases} \tag{6.3}$$

denote the k^{th} destination node for redundant copies of entries owned by node i, and let $g_i(s)$ denote the number of sets $S_{id_{ik}}$ with $s \in S_{id_{ik}}$ for all $k \in \{1, 2, \ldots, \phi\}$. Then, the required redundancy for tolerating up to ϕ simultaneous node failures can be achieved by sending

$$R_{ik}^c := \{s \in S_i \mid s \notin S_{id_{ik}} \ \wedge \ m_i(s) - g_i(s) \leq \phi - k\} \tag{6.4}$$

to node d_{ik} for all $i \in \{1, 2, \ldots, N\}$ and $k \in \{1, 2, \ldots, \phi\}$, that is, by sending entries that are not already being sent to $S_{id_{ik}}$ for the regular SpMV, and which still have not reached the desired redundancy in the cluster, to d_{ik}. Note that the sets R_{ik}^c are of minimal size such that the required number ϕ of redundant copies of each search direction vector element is ensured. It holds that $|R_{i1}^c| \geq |R_{i2}^c| \geq \cdots \geq |R_{i\phi}^c|$.

The strategy proposed in Eq. (6.3) for selecting the nodes to keep the redundant copies of $\boldsymbol{p}_i^{(j-1)}$ and $\boldsymbol{p}_i^{(j)}$ is a reasonably good heuristic for minimizing communication overheads during SpMV if we assume that the entries of the system matrix $\boldsymbol{A}$ are mostly clustered around the diagonal (since it then is likely that there are some elements which have to be sent anyway from node i to node d_{ik} and, thus, there is no extra latency for establishing a new connection; see Section 6.1.3 for a more detailed discussion). For matrices with very different sparsity patterns, strategies different from Eq. (6.3) may be preferable. A comprehensive analysis of the interaction between a given sparsity pattern and the optimal choice of the "backup nodes" is work in progress.

When the backup strategy based on Eq. (6.3) and Eq. (6.4) is employed, we have $\phi + 1$ copies of each element of $\boldsymbol{p}_i^{(j-1)}$ and $\boldsymbol{p}_i^{(j)}$ on $\phi + 1$ different nodes (including node i that owns the block) and the two most recent search directions $\boldsymbol{p}^{(j-1)}$ and $\boldsymbol{p}^{(j)}$ can be fully recovered after a simultaneous or overlapping failure of up to ϕ arbitrary nodes. If the node failures do not happen simultaneously but are overlapping in time, i.e., more node failures occur during the reconstruction phase, the reconstruction process must be restarted after each node failure (an efficient implementation can of course skip steps that have already been performed and are not affected by the subsequent node failures).

We consider $\psi \leq \phi$ node failures. Let $f_1, f_2, \ldots, f_\psi$ denote the nodes that fail. Then, we can define $I_f := I_{f_1} \cup I_{f_2} \cup \cdots \cup I_{f_\psi}$ and use a similar reconstruction procedure as in the case of a single-node failure. Some of the reconstruction steps of Alg. 2 can be performed locally on each of the replacement nodes $f_1, f_2, \ldots, f_\psi$. However, for computing the matrix-vector

products and solving the linear systems in lines 5 to 8 of Alg. 2, additional communication between the ψ replacement nodes is necessary. In Section 6.1.2, we show analytically that the capability of tolerating up to ϕ node failures may incur increased communication cost. Hence, ϕ should be chosen only as large as necessary for handling the expected number of simultaneous or overlapping node failures on a given parallel computer.

Analysis

Communication time usually is the main cost for a parallel algorithm, dominating the computation time [Bal+14]. For analyzing the communication overhead of sending the additional elements of the sets R_{ik}^c as defined in Eq. (6.4), we adopt a latency-bandwidth communication model [Cha+07] (using respectively λ and μ to represent the latency and inverse bandwidth) and assume that the communication cost solely depends on latency $\lambda_{ik} > 0$—which may vary for different sending nodes i and corresponding receiving nodes d_{ik} according to Eq. (6.3)—and cost $\mu > 0$ per vector element. We further assume that each node is able to send and receive exactly one element at a time. If $S_{id_{ik}} \neq \emptyset$, the additional elements of R_{ik}^c are sent together with the elements of $S_{id_{ik}}$, which have to be sent anyway during computing the sparse matrix-vector product $\boldsymbol{A}\boldsymbol{p}^{(j)}$.

If this is the case for all pairs of nodes for a fixed $k \in \{1, 2, \ldots, \phi\}$, which we refer to as communication round k, no extra latency cost applies in that round, and the overhead is $\max_i |R_{ik}^c|\mu$. In contrast, if $\forall i \in \{1, 2, \ldots, N\}\colon S_{id_{ik}} = \emptyset$ in communication round $k \in \{1, 2, \ldots, \phi\}$, extra latency λ_{ik} is incurred for all pairs of nodes, and the overhead is $\max_i(\lambda_{ik} + |R_{ik}^c|\mu) \leq \max_i \lambda_{ik} + \max_i |R_{ik}^c|\mu$. Hence, in communication round $k \in \{1, 2, \ldots, \phi\}$, it holds for the communication overhead $\mathcal{O}$ that

$$\mathcal{O} \leq \max_i \left(\lambda_{ik} + |R_{ik}^c|\,\mu\right) \leq \max_i \lambda_{ik} + \max_i |R_{ik}^c|\,\mu,$$

where $i \in \{1, 2, \ldots, N\}$ and $\max_i |R_{ik}^c|\mu = 0$ if and only if $\forall i\colon |R_{ik}^c| = 0$, i.e., there are at least $\phi - k + 1$ redundant copies of all elements of $\boldsymbol{p}^{(j)}$ due to the sparsity pattern of $\boldsymbol{A}$. For all ϕ communication rounds, it follows that

$$\begin{aligned}\mathcal{O} &\leq \sum_{k=1}^{\phi} \max_i \left(\lambda_{ik} + |R_{ik}^c|\,\mu\right) \leq \sum_{k=1}^{\phi} \left(\max_i \lambda_{ik} + \max_i |R_{ik}^c|\,\mu\right) \\ &= \sum_{k=1}^{\phi} \underbrace{\max_i \lambda_{ik}}_{\leq \lambda_{\max}} + \sum_{k=1}^{\phi} \underbrace{\max_i |R_{ik}^c|}_{\leq \left\lceil \frac{n}{N} \right\rceil} \mu \leq \phi\lambda_{\max} + \phi \left\lceil \frac{n}{N} \right\rceil \mu,\end{aligned}$$

where $\lambda_{\max} \coloneqq \max_{i,k} \lambda_{ik}$. Hence, the communication overhead $\mathcal{O}$ for keeping ϕ redundant

copies of all elements of $\boldsymbol{p}^{(j)}$ has an upper bound $\phi\left(\lambda_{\max} + \left\lceil \frac{n}{N} \right\rceil \mu\right)$. The actual communication overhead within that interval entirely depends on the sparsity pattern of $\boldsymbol{A}$, which determines the elements in R_{ik}^c.

The memory overhead to hold the redundant information, which we denote $\mathcal{M}$, measured in floating-point numbers, is two times the sum of all the additional entries sent from every node to each of its destination nodes (for the case of PCG, we store two vectors redundantly). It depends strongly on the sparsity pattern of the matrix and the selection of destination nodes. We provide an expression for it, along an upper bound for the worst case, in a scenario with uniformly distributed rows among the nodes:

$$\mathcal{M} = \sum_{i=1}^{N} \sum_{k=1}^{\phi} 2 \underbrace{\left|R_{ik}^c\right|}_{\leq \left\lceil \frac{n}{N} \right\rceil} \leq 2n\phi. \tag{6.5}$$

That is to say, in the worst case, we need enough space to replicate each vector ϕ additional times. Of course, this is only the case if we attempt to send all redundant information to nodes that would have not received any entries from the source node anyway. In our scenarios with banded matrices, the nodes with ranks close to the source node would likely already receive some entries from it for the computation of the SpMV.

6.1.3 Influence of the sparsity pattern

In this section, we take a closer look at the implications of our theoretical analysis. As we showed in Section 6.1.2, there exist matrices with sparsity patterns which lead to *zero* communication overhead during the failure-free execution of the PCG solver, i.e., no extra communication is necessary for distributing ϕ redundant copies of all elements of the search direction vector $\boldsymbol{p}^{(j)}$ during the computation of the sparse matrix-vector product $\boldsymbol{A}\boldsymbol{p}^{(j)}$. On the other hand, other matrix sparsity patterns may lead to significant communication overhead during the SpMV operation.

Independent of the particular strategy for selecting the backup nodes, it is clearly beneficial for having a low communication overhead if $m_i(s) \geq \phi$ (cf. Eq. (6.1)) holds for most or even all $s \in S_i$ and $i \in \{1, 2, \ldots, N\}$ or, in other words, if most or all elements of $\boldsymbol{p}^{(j)}$ anyway—i.e., due to the sparsity pattern of $\boldsymbol{A}$—have to be communicated to at least ϕ different nodes during the computation of the product $\boldsymbol{A}\boldsymbol{p}^{(j)}$.

If this is not the case, a possibly considerable number of extra vector elements has to be transferred during the SpMV operation. Since the number of extra elements to be sent is determined by the sparsity pattern of $\boldsymbol{A}$, communication cost can then only be reduced by avoiding extra latencies, i.e., by sending the extra elements to nodes where also other elements have to be sent to, thus avoiding the need to start sending additional messages between nodes. In general, our strategy defined by Eq. (6.3) and Eq. (6.4) performs well

if $\boldsymbol{A}$ is not *too* sparse within a bandwidth of $\lceil \phi n/(2N) \rceil$ around the diagonal (but can still be very sparse overall). More formally, if for all $i \in \{1, 2, \ldots, N\}$ and $k \in \{1, 2, \ldots, \phi\}$ at least one element in each submatrix $\boldsymbol{A}_{I_{d_{ik}}, I_i}$ is not equal to zero, no cost is incurred for additional latency.

6.1.4 Implementation

Up to this point, we have analyzed the algorithms in theoretical terms. We now describe how the reconstruction is realized in a real-life, finite-precision machine.

We implement Chen's ESR algorithm [Che11] and our novel extensions as described in Section 5 and Section 6.1.2 using the PETSc framework [Bal+97; Bal+18]. PETSc provides the CG solver and linear algebra operations, and it also manages communication between nodes. We use operations offered by the framework to transfer the information required for the recovery from node failures. We use a block Jacobi as a preconditioner during the regular operation of the solver, solving the preconditioner blocks exactly.

To impart fault tolerance, the SpMV is augmented to transfer the additional data required to obtain the desired level of data redundancy (cf. Section 5.1.1). In PETSc, the SpMV operation is realized with a generalized scatter: A node determines, from the non-zero entries in its matrix rows, what vector components it requires from its neighbors to perform the SpMV product. With this information, PETSc collectively creates a *communication context* for the generalized scatter operation, defining what entries of a distributed vector are communicated, and where they must be communicated to.

In our experiments, we simulate node failures. Instead of taking down nodes and producing replacements during the reconstruction phase, a node will perform the operations and communication required to restore the solver state. The reconstruction process requires sending the surviving entries of the search direction vector to replacement nodes. In our experiments, this is achieved by reversing the communication that takes place during the matrix-vector product. PETSc already provides this functionality. However, reversing the communication that occurs during the matrix-vector product is not a well-defined operation. To see this, imagine a communication context that dictates that the entry in position i_0 of a vector, located in some node A, must be transferred to positions i_1, in node B, and i_2, in node C. In the reverse communication process, entries in positions i_1 and i_2 will hold candidates for the value of that entry to be transferred to position i_0. In the absence of node-failures, both candidates will be the same value, because they are copies of the original entry in i_0, and the operation is then well defined, but, in the event of a node failure, it is possible that one of these entries is lost and the resulting candidates are different, conflicting values. Therefore, in such cases, the reverse communication process used in the reconstruction could be non-deterministic. We cope with this issue by keeping the search directions in the nodes simulating a node failure. If this information is stored,

communication with the reversed context is deterministic, because there would not be a conflict between the candidates.

This problem arises because we use the reverse of a communication context intended for SpMV. In a custom implementation, which we use in later sections of this thesis, a tailored communication context can be produced after the node failure takes place, which avoids this problem altogether by selecting the entries that we need for the reconstruction.

Working with our modifications to PETSc, the linear system arising in line 8 of Alg. 2 is solved using a PCG solver assembled with global operations. In particular, the matrix-vector products of the submatrix $\boldsymbol{A}_{I_f,I_f}$ and the subvector $\boldsymbol{x}_f^{(j)}$ are performed by multiplying the entire matrix $\boldsymbol{A}$ with a modified vector $\boldsymbol{x}^{(j)}$, whose appropriate entries were set to zero. The desired $\boldsymbol{A}_{I_f,I_f}\boldsymbol{x}_f^{(j)}$ is a subvector of the result of this global operation. This is less efficient than working with the actual submatrix of $\boldsymbol{A}$, but some of the changes we introduced to increase redundancy conflict with PETSc' ability to create submatrices. The cost to reconstruct the solution, however, remains very small compared to the overall runtime. The CG solver for the subsystem uses a block Jacobi preconditioner, with blocks matching the process' index set. We use an approximate solver based on ILU factorization for the blocks.

Avoiding loss of orthogonality

For this section it is useful to distinguish between the *solver residual*, that is, vector $\boldsymbol{r}^{(j)}$ from Alg. 1, and the vector $\boldsymbol{b} - A\boldsymbol{x}^{(j)}$. These vectors are, in general, not equal in a finite-precision machine.

The CG algorithm in floating-point arithmetic undergoes loss of orthogonality, where round-off error accumulates and the conjugacy of the search directions is lost as the solver progresses. Consequently, in regular PCG the solver residual and the vector $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$ will differ slightly after convergence. Because we work with finite precision, and because we solve the local linear system in the reconstruction process (line 8 of Alg. 2) iteratively, our algorithm only reconstructs an approximation of the solver state before the node failures take place, thus potentially further contributing to this loss of orthogonality: Consequently, the ESR solver residual after convergence can be larger than the solver residual of PCG. The largest source of deviations is the solution of the local linear system of line 8 of Alg. 2. The loss of orthogonality relative to regular PCG can be controlled with the tolerance of this linear system.

To compare the accuracies of ESR and PCG in this regard, we define the *relative residual difference* metric:

$$\Delta_{\text{ESR}} = \frac{\|\boldsymbol{r}_{\text{ESR}}\|_2 - \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{\text{ESR}}\|_2}{\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{\text{ESR}}\|_2}$$
$$\Delta_{\text{PCG}} = \frac{\|\boldsymbol{r}_{\text{PCG}}\|_2 - \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{\text{PCG}}\|_2}{\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{\text{PCG}}\|_2}. \tag{6.6}$$

Here, $\boldsymbol{r}_{\text{ESR}}$ and $\boldsymbol{r}_{\text{PCG}}$ are the solver residual vectors of ESR with reconstruction and reference PCG respectively after convergence, and $\boldsymbol{x}_{\text{ESR}}$ and $\boldsymbol{x}_{\text{PCG}}$ are the corresponding iterands.

A side-by-side comparison of the ESR method and regular PCG can show that the former is as accurate as the latter. In Section 6.1.5 we show that the effects of using finite-precision arithmetic can be made negligible. Since node failures are uncommon and reconstruction is relatively cheap to perform, we can set the tolerance for the local system to a very small value, so that ESR converges, while the reconstruction overhead remains low.

6.1.5 Numerical experiments

Now we summarize our experimental setup and discuss our results.

Experimental setup

We use SPD matrices from the SuiteSparse Matrix Collection [DH11] as test problems, selecting problems from different application areas. Their properties are summarized in Table 6.1. We select medium (M1, M2) and large (M3-M8) size problems. The latter are among the largest available SPD matrices in the SuiteSparse Matrix Collection. There are problems with different numbers of non-zeros. Large matrices with relatively few non-zeros are common, but problems with more non-zero entries are more expensive to compute and would benefit the most from resilience schemes to protect the resource investment.

Our experiments are run on 128 nodes of the VSC3 system of the Vienna Scientific Cluster. Although our algorithm is well suited for multiple processes per node, we use only one process per node in our experiments. The argument for this decision is twofold: Firstly, from the point of view of resilience, the number of processes per node (cf. Section 6.1.1) makes no difference since the redundant vector elements always have to be stored on a different node (not just in the memory of another process). Secondly, the runtimes obtained in our experiments are based on simulations of node failures without ULFM (cf. Secs. 6.1.1 and 6.1.4). Hence, the relative runtime differences are more significant than the absolute runtimes (and, thus, improvements of the absolute runtimes due to multiple processes per node). Experiments for a matrix are run on the same set of nodes of VSC3. The system's topology is a fat tree. We use the following libraries:

Table 6.1: SPD matrices from [DH11] used in the experiments. n: Problem size. $\#NZ$: Number of non-zero entries. The matrices are ordered by increasing number of non-zero entries, with a larger ID indicating a larger number of non-zeros.

Name	**Id**	**Problem type**	n	$\#NZ$
parabolic_fem	M1	Fluid dynamics	525 825	3 674 625
offshore	M2	Electromagnetics	259 789	4 242 673
G3_circuit	M3	Circuit simulation	1 585 478	7 660 826
thermal2	M4	Thermal	1 228 045	8 580 313
Emilia_923	M5	Structural	923 136	40 373 538
Geo_1438	M6	Structural	1 437 960	60 236 322
Serena	M7	Structural	1 391 349	64 131 971
audikw_1	M8	Structural	943 695	77 651 847

- Intel MPI 5.1.3
- PETSc 3.10.4
- Intel MKL 2018.3

. We use the Intel C compiler version 18.0.5 with compiler flags `-O3`, `-march=native` and `-mtune=native`. We terminate the solver once the relative residual norm has been reduced by a factor of 10^8. The local linear system used during the reconstruction is terminated once its residual norm is reduced by a factor of 10^{14}.

We measure the runtime of the solver for several problem settings. Node failures are introduced once for each simulation, with either one, three or eight simultaneous node failures taking place at either 20%, 50% or 80% of the solver progress (measured in number of iterations). These failures are placed in contiguous ranks. Simultaneous node failures can well be caused by a faulty switch, therefore it seems like a realistic assumption that they are clustered: The node failures are introduced in neighbouring ranks either at the beginning or in the center of the vector, starting from rank 0 or 64 respectively. Additionally, we have results for runs without failures with either one, three or eight redundant copies. Each measurement in this test constellation is repeated at least 5 times.

Experimental results

Our experimental results are summarized in Tab. 6.3. Statistics for node failures are computed over at least 15 values: at least 5 measurements for node failures introduced at either 20%, 50% and 80% of the solver's progress.

As expected, a larger number of redundant copies leads to larger overheads. In general, the reconstruction time remains small, with larger relative reconstruction costs for matrices

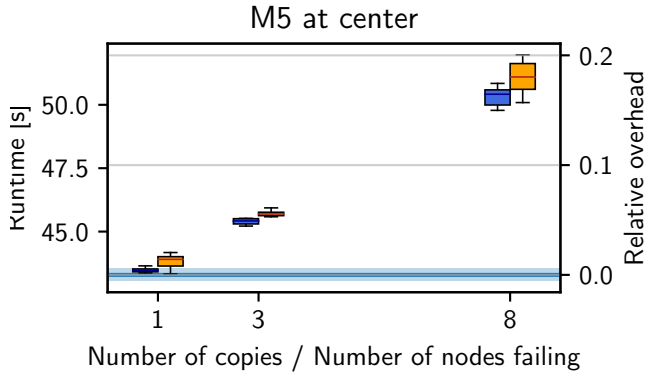


Figure 6.1: Runtimes and relative overhead for the matrix M5 of Tab. 6.1, for node failures introduced close to the center of the vector. The blue line at the bottom of the figure represents the reference time obtained solving the system using PETSc without modifications, with respect to which the relative overhead is measured, and the band around it extends for one standard deviation in each direction. The x-axis indicates the number of copies the resilient solver holds. Boxes include points in the interquartile range, and whiskers extend up to 1.5 times the width of the interquartile range. Blue boxes (to the left of a group) represent runs with the resilient solver without node failures. Orange boxes (to the right of a group) represent runs with node failures. In experiments with node failures, we introduce as many simultaneous failures as the solver can tolerate (one, three or eight failures respectively.) Orange boxes include experiments for failures introduced at 20%, 50% or 80% progress of the solver.

with smaller runtimes, where the absolute cost is consequently smaller.

The overall relative overhead with node failures, shown in the last three columns of Tab. 6.3, corresponds to the sum of the relative overhead of the undisturbed case plus the relative runtime cost of the reconstruction. These columns roughly match the sum, as expected, with some variation arising from the variation in runtime of running the code in a real machine, and from differences in the number of iterations caused by the reconstruction (cf. [PLG18]).

Tab. 6.3 also shows that the location of the node failure does affect the reconstruction cost, as the runtime is, in general, different for node failures at the start (lower indexes) or the center (middle indexes) of the vectors. These differences come from different diagonal linear systems in the reconstruction: Submatrices formed from the index sets of different failed nodes have different properties, and they will not converge at the same rate with an ILU preconditioner, thus affecting the reconstruction time.

Our experiments show that our algorithm is very efficient for matrices with many nonzeros contained in a band close to the diagonal. Relatively denser matrices also take longer to reach convergence because of the longer time required for the matrix-vector product, so it makes a lot of sense to protect the time investment in the solution process with a resilience technique like the one presented in this paper.

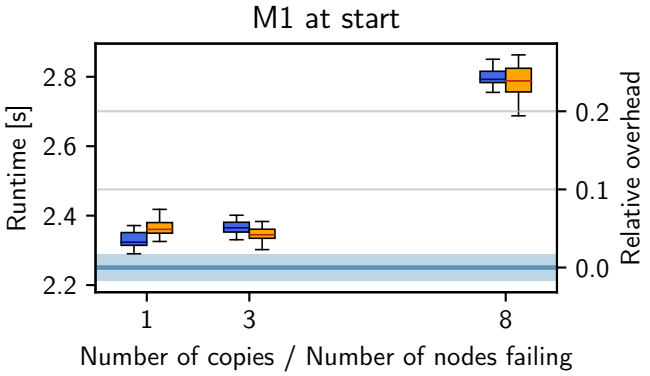


Figure 6.2: Runtimes and relative overheads for matrix M1 with node failures occurring close to the start (lower indices) of the vectors. This figure uses the same conventions as Fig. 6.1, and showcases a situation, for three redundant copies, where the solver converges faster after performing reconstruction after a node failure due to the reduction of the number of iterations until convergence.

Fig. 6.1 is an example of the runtimes and overheads obtained with our novel resilient algorithm for the matrix M5. For this matrix, the state reconstruction operation takes very little time: The runtimes for cases with failures, (orange boxes) are very close to the failure-free cases (blue boxes). In this case, the overhead for the method comes predominantly from the additional communication required to maintain redundant data.

In Fig. 6.2 the boxes corresponding to three redundant copies indicate a *smaller* runtime for simulations with node failures than for the failure-free case. As mentioned before, this can happen, since the number of iterations required after reconstruction can be a smaller than in the failure-free run (cf. [PLG18]).

Fig. 6.3 shows a test case where the overhead required to keep redundant data increases superlinearly with the number of node failures tolerated, As explained in Section 6.1.3, the growth of the overhead with the number of node failures tolerated strongly depends on the sparsity pattern of $\boldsymbol{A}$. The matrix M8 contains many non-zeros in a band around the diagonal in the middle indexes. As expected from the analysis of Section 6.1.3, this is a particularly favorable case for our method: It can resist three simultaneous node failures with an overhead of around 2.5%, and eight node failures with an overhead of around 10%.

In general, the iteration at which the node failures are introduced has little influence on the runtime of the solver. Fig. 6.4 illustrates this for M5. We observed the same behaviour for the other test cases.

In Section 6.1.4, we discuss the potential loss of orthogonality that occurs when performing the reconstruction. Tab. 6.2 shows that the relative residual difference for our method is comparable to the one for the reference results, even for its maximum value among all experiments for a given matrix. The deviations for both methods are tiny in comparison to the reduction of the residual norm by a factor of 10^8 from the solver.

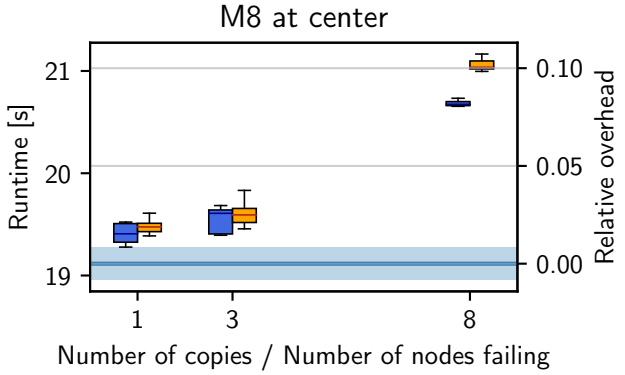


Figure 6.3: Runtimes and relative overheads for matrix M8 with node failures occurring close to the start of the vectors. This figure uses the same conventions as Fig. 6.1, and shows superlinear increase of the overhead with respect to the number of redundant copies held.

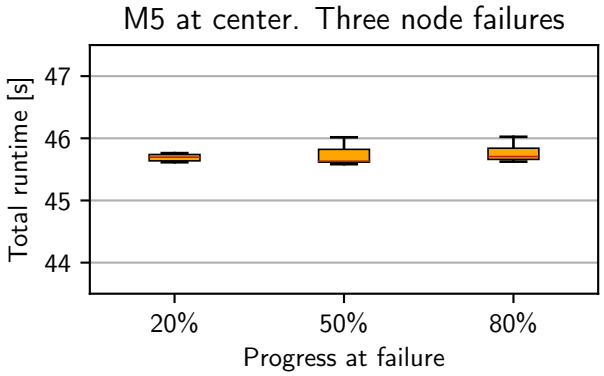


Figure 6.4: Runtime for matrix M5 when introducing the failure of three nodes at the center (middle indexes) of the vectors and at different iterations along the progress of the solver. The boxes contain runtimes in the interquartile range, and the whiskers extend up to 1.5 times the width of the interquartile range.

Table 6.2: Evaluation of the metric of Eq. (6.6). The first column shows the maximum value of the relative residual deviation for all experiments with node failures for a matrix. The second column shows the relative residual deviation for the reference run.

ID	**max** $\Delta_{\mathbf{ESR}}$	$\Delta_{\mathbf{PCG}}$
M1	3.46×10^{-7}	-1.63×10^{-7}
M2	2.24×10^{-7}	1.86×10^{-7}
M3	1.57×10^{-7}	1.57×10^{-7}
M4	1.96×10^{-7}	9.06×10^{-8}
M5	3.59×10^{-5}	1.81×10^{-6}
M6	8.80×10^{-8}	-3.31×10^{-8}
M7	1.32×10^{-7}	-7.24×10^{-8}
M8	2.64×10^{-3}	1.49×10^{-3}

Table 6.3: Summarized experimental results. Values are aggregated for experiments with different times (iteration numbers) when the failures are introduced. Matrices are ordered by descending reference runtime, with the ID number of a matrix growing with increasing number of non-zeros. t_0: average reference time for the reference runs with regular (non-fault tolerant) PCG. ϕ: number of redundant copies for the search direction. ψ: number of simultaneous node failures introduced. The column "relative overhead undisturbed" shows the mean overhead of modified ESR-capable PCG, with a given number of redundant copies ϕ, with respect to the reference time t_0 if no reconstruction takes place. The columns marked "relative reconstruction time" indicate the time that it takes to reconstruct the state of the solver, expressed as a percentage of the reference time t_0, plus/minus a standard deviation. The columns marked "relative overhead with failures" show the mean, plus/minus a standard deviation, of the overall relative overhead, that is, the relative overhead for the total time until convergence, when node failures occur and the state of the solver is reconstructed, with respect to the reference runtime t_0.

ID	t_0 **[s]**	**Relative overhead undisturbed [%]**			**Failure location**	**Relative reconstruction time [%]**			**Overhead with failures [%]**		
		$\phi = 1$	$\phi = 3$	$\phi = 8$		$\psi = \phi = 1$	$\psi = \phi = 3$	$\psi = \phi = 8$	$\psi = \phi = 1$	$\psi = \phi = 3$	$\psi = \phi = 8$
M5	43.29	0.4	5.1	16.3	start	0.2 ± 0.1	0.3 ± 0.1	0.6 ± 0.1	1.2 ± 0.5	6.0 ± 0.3	19.6 ± 0.7
					center	0.0 ± 0.1	0.1 ± 0.1	0.4 ± 0.1	1.2 ± 0.6	5.6 ± 0.3	20.5 ± 6.0
M8	19.12	1.5	2.2	8.2	start	1.4 ± 0.1	3.5 ± 0.2	10.7 ± 0.2	3.8 ± 1.0	5.6 ± 0.7	19.9 ± 0.9
					center	0.4 ± 0.1	0.8 ± 0.1	1.4 ± 0.1	1.9 ± 0.4	2.8 ± 1.0	10.4 ± 1.1
M6	11.70	0.3	3.1	15.1	start	1.3 ± 0.1	2.4 ± 0.2	5.3 ± 0.2	1.4 ± 0.3	5.9 ± 0.4	21.2 ± 1.2
					center	0.3 ± 0.1	0.6 ± 0.1	1.6 ± 0.1	0.7 ± 0.4	4.0 ± 0.3	18.3 ± 0.7
M7	6.48	0.2	4.3	13.6	start	2.6 ± 0.1	8.2 ± 0.3	23.4 ± 0.6	2.9 ± 0.3	13.7 ± 0.6	39.6 ± 1.4
					center	1.2 ± 0.1	1.5 ± 0.1	21.1 ± 0.7	1.2 ± 0.2	7.1 ± 0.6	36.7 ± 1.7
M4	6.31	8.2	22.8	65.6	start	2.0 ± 0.1	3.1 ± 0.1	4.9 ± 0.3	9.6 ± 0.8	26.4 ± 1.1	66.0 ± 7.9
					center	1.3 ± 0.1	3.8 ± 0.2	5.2 ± 0.4	9.2 ± 2.6	35.7 ± 2.2	63.4 ± 9.3
M1	2.25	3.6	5.1	24.5	start	0.3 ± 0.1	0.3 ± 0.1	0.4 ± 0.1	5.1 ± 1.2	4.3 ± 0.9	23.8 ± 2.1
					center	0.3 ± 0.1	0.3 ± 0.1	0.4 ± 0.1	5.0 ± 1.0	5.1 ± 1.3	25.4 ± 1.8
M2	1.58	8.0	8.1	21.1	start	4.0 ± 0.2	7.0 ± 0.3	9.7 ± 0.4	7.3 ± 4.6	15.6 ± 2.3	30.1 ± 5.5
					center	2.1 ± 0.3	3.5 ± 0.2	4.4 ± 0.3	10.0 ± 4.6	14.2 ± 2.6	23.8 ± 3.0
M3	1.16	5.0	24.1	91.3	start	0.6 ± 0.1	1.0 ± 0.1	1.9 ± 0.1	6.3 ± 1.7	24.8 ± 1.8	93.6 ± 6.0
					center	8.0 ± 0.5	32.2 ± 1.2	58.1 ± 1.4	14.8 ± 2.0	55.0 ± 3.2	147.6 ± 5.0

6.2 Recovery without the use of spare nodes

In this Section, we present the work published in [PPG19]. We discuss a technique to perform the reconstruction of the PCG state described in [Pac+19] without the need for replacement nodes being in standby while the solver progresses. From an initial number of nodes N before the node failure, if a node fails, the task would have to be completed with $N-1$ nodes. The objective is to complete the reconstruction and continue until the problem is solved using these $N-1$ nodes.

6.2.1 Problem definition and contributions

We consider the scenario that PCG runs on a computer cluster that is vulnerable to node failures. In the reconstruction phase, all the vectors of the solver (iterand, search direction, residual) are reconstructed in order to recover the state they were in before the node failures took place. The techniques proposed in [Pac+19] assume the availability of spare nodes. Some frameworks already consider the need for keeping a pool of spare nodes [Gam+14]. However, maintaining such a pool of inactive nodes, or, as a user, requesting them while launching an application, is a considerable extra cost.

We make the following contributions:

1. We propose several strategies for the redistribution of data after the reconstruction of the state of a PCG solver, such that the solver can finish with the set of surviving nodes (Section 6.2.2),

2. we evaluate one of these strategies experimentally for the case of a single node failure (Section 6.2.3) and

3. we identify a trade-off between load balancing and convergence speed through the choice of a preconditioner after the reconstruction phase.

6.2.2 Data redistribution

If spare nodes are available for the recovery phase, the data contained in each node afterwards will be identical to the data before the node failure. In this situation, there is no need to think about load balancing in the recovery phase, since we assume that this problem is addressed as the matrix is partitioned, before being passed to the solver. If we want to recover the solver without spare nodes, however, we have to be mindful of this matter.

To motivate the need for a redistribution strategy, imagine the situation where one of the surviving nodes reconstructs and keeps the data and index set of the lost node, in addition to its own. If we attempt to continue the solver iterations, with this node now

also being responsible for the recovered data, it is well possible that the time cost of each iteration doubles due to load imbalance. In this section, we propose strategies to perform a redistribution of reconstructed data.

Redistribution strategies

We work with block-row distributed matrices in all stages of the algorithms. Thus, we can express the changes of ownership of the vector entries –as a subset of the surviving nodes take charge of the data of the lost nodes– with a permutation matrix $\boldsymbol{T}$ (designated so to avoid confusion with the preconditioner operator $\boldsymbol{P}$) and the size of the index set of each node. Both before and after a node failure and the recovery process, nodes will contain information for disjoint sets of elements with contiguous indices.

We assume that the number of non-zeros per row is more-or-less equal for all rows of the matrix. Under this assumption, it is enough to have a balanced number of rows in each processor to have a balanced load in the system. Widely used scientific frameworks, such as PETSc [Bal+97], distribute the rows of a matrix uniformly by default. A more sophisticated approach remains as an open research direction.

At any rate, we require the submatrix $\boldsymbol{A}_{I_f,I_f}$ to be in the reconstructing node.

Row permutation After a node failure occurs, the rows of the matrix corresponding to the affected nodes are loaded by the reconstructing node, then distributed among the surviving nodes. The redistribution operations are realized with `MPI_Scatterv` calls.

The distribution of the rows will obey a permutation matrix $\boldsymbol{T}$. The new system matrix then becomes a row permutation of the original system matrix: $\boldsymbol{A_r} = \boldsymbol{TA}$. In order to obtain the result of the product $\boldsymbol{Ax}$ working with the modified matrix $\boldsymbol{A_r}$, it is necessary to permute the vector as well:

$$\boldsymbol{Ax} = \boldsymbol{T}^{-1}(\boldsymbol{TA})x = \boldsymbol{T}^{\intercal}\boldsymbol{A_r}x. \tag{6.7}$$

The second equality of Eq. (6.7) holds because, being a permutation matrix, $\boldsymbol{T}$ is orthonormal.

After a node failure, it is necessary to permute the vectors every time a matrix-vector product is performed, which entails additional communication, and an overhead on every iteration. Even though the matrix $\boldsymbol{TA}$ is not necessarily symmetric and positive definite (SPD), together with the vector permutation it makes the SpMV equivalent to multiplying with a SPD matrix.

This approach is conceptually the simplest one. It is easy to implement, but the permutation required after every step is an overhead that accumulates with each iteration until convergence. Therefore, the overhead will be less significant if the reconstruction happens

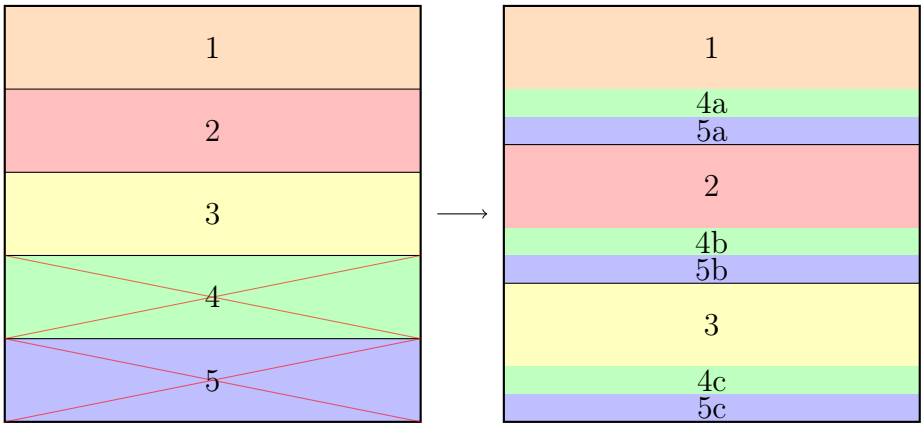


Figure 6.5: Representation of the redistribution of rows after a node failure, using the row permutation strategy. Regions separated by lines represent the rows assigned to a rank. Contiguous blocks of rows of the matrix, here represented with different colors, are initially assigned to different ranks. Nodes 4 and 5 fail simultaneously. During the reconstruction, two of the remaining nodes (to make up for the two lost nodes) read in the matrix rows and perform the reconstruction. After this is complete, the matrix rows are redistributed as indicated in this figure: Rows corresponding to the lost nodes are distributed uniformly among the surviving nodes. A permutation matrix $\boldsymbol{T}$ exists that transforms the matrix to the left, $\boldsymbol{A}$, into the matrix to the right, $\boldsymbol{TA}$, fitting in three nodes.

when the solver is close to the solution already.

Row and column permutation The rows are permuted according to $\boldsymbol{T}$, but we additionally reorder the rows of the matrix in the reconstruction process, thus producing the matrix $\boldsymbol{A_{rc}} = \boldsymbol{TAT}^\intercal$. This is illustrated in Fig 6.6.

By performing column permutation on the reconstructed matrix, solver vectors can remain in a permuted representation while the solver operates:

$$\boldsymbol{y} = \boldsymbol{Ax}, \qquad \boldsymbol{Ty} = (\boldsymbol{TAT}^\intercal)\boldsymbol{Tx} = \boldsymbol{A_{rc}Tx}. \tag{6.8}$$

The matrix $\boldsymbol{A_{rc}}$ is symmetric and positive definite (see Appendix A, Lemma 1).

With this strategy, we have to perform the additional column permutation of the matrix once during the reconstruction process, but the operation of the solver can continue without the necessity of permuting the solver vectors on every iteration. At the end of the solution process, the iterand is permuted back to the original order.

Just as with the column permutation case, the redistribution is performed using the function `MPI_Scatterv`.

Conserve the original order of the rows Local surviving data and recovered data is distributed in such a way that the original order of the rows is kept, thus avoiding the

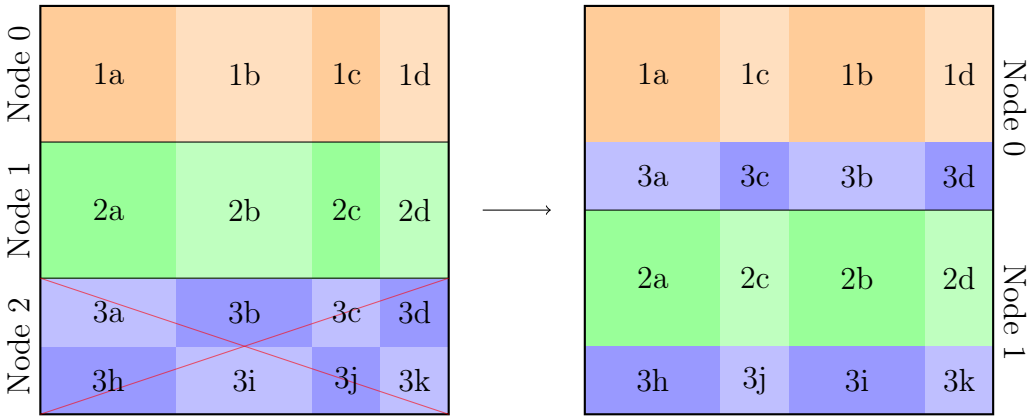


Figure 6.6: Illustration of the row-column redistribution. On the left and right, the distribution of rows and columns of the matrix among the nodes, before and after a node failure, respectively. In a solver initially working with three nodes, node 2 is affected by a node failure. Its rows are redistributed among nodes 0 and 1 defining a permutation $\boldsymbol{T}$. This requires communication between nodes. The columns of the matrix are then reordered according to $\boldsymbol{T}$.

need for additional permutations on every iteration.

With this approach, every rank has to communicate with its neighbors to transmit part of its rows and entries. The redistribution is realized with left and right `MPI_Sendrecv` operations.

Reloading the entire matrix The matrix is reloaded using the routines of the linear algebra package used by our implementation, just as it was done at the start of the program, but working with the surviving nodes. Notice, however, that in order to perform the reconstruction, one of the nodes must hold the submatrix $\boldsymbol{A}_{I_f,I_f}$ at some point: This approach does not eliminate the need to communicate matrix information between the nodes during the recovery phase.

Recovery of the preconditioner

Besides the matrix, a valid preconditioner must be produced, both for the reconstruction process and for the continuation of the iterations of PCG. For reconstructing, we need the operator $\boldsymbol{P}_{I_f,I_f}$ as it was used up to the time of the node failure. Assuming a constant preconditioner, it is possible to keep its data in reliable storage, and retrieve it during the reconstruction stage. As an alternative, it is also possible to recompute $\boldsymbol{P}_{I_f,I_f}$. Both of these schemes are easy to carry out for Jacobi and block Jacobi preconditioners.

After the redistribution, we can either recompute a preconditioner for the new matrix, or redistribute the preconditioner itself. Care must be taken while setting this up. For illustration, consider the following matrix:

$$\bar{\boldsymbol{A}} = \left(\begin{array}{cccccc} 2 & 1 & & & & \\ 1 & 2 & & & & \\ \hline & & 2 & 1 & & \\ & & 1 & 2 & & \\ \hline & & & & 2 & 1 \\ & & & & 1 & 2 \end{array}\right).$$

Suppose that we solve a linear linear system for this matrix with the PCG method. The rows of the $\bar{\boldsymbol{A}}$ are divided into three processes, each receiving a consecutive pair of rows. The block Jacobi preconditioner would work very well with this matrix (the solver would converge after the first iteration). Suppose now that we perform row-column redistribution for the node responsible for the last pair of rows. The permuted matrix is now

$$\bar{\boldsymbol{A}}_{rc} = \left(\begin{array}{cccccc} 2 & 1 & & & & \\ 1 & 2 & & & & \\ & & 2 & & & 1 \\ \hline & & & 2 & 1 & \\ & & & 1 & 2 & \\ & & 1 & & & 2 \end{array}\right).$$

Now that we are working on two nodes, not all of the non-zeros fit in local blocks (leading to non-local operations when applying the preconditioner), and a block Jacobi preconditioner would not work as well as before the matrix was permuted (in this little example, the preconditioner is no longer exactly the inverse of the matrix).

We may try to produce a new block Jacobi preconditioner using one of the following approaches:

1. We can recompute the preconditioner for $\bar{\boldsymbol{A}}_{rc}$, but we might end up with something that does not work as well as it does on $\bar{\boldsymbol{A}}$, and might cause an increased number of iterations when compared to the failure-free solver.

2. If the preconditioner itself is redistributed, then it will be mathematically the same as it was before the node failure. However, its sparsity pattern in the permuted representation will change and, as illustrated by the changes of sparsity pattern from $\bar{\boldsymbol{A}}$ to $\bar{\boldsymbol{A}}_{rc}$, it is possible that it will require communication between ranks, since there are non-zeros outside of the diagonal block.

Both of these situations are undesirable. However, these obstacles do not hinder convergence.

The Jacobi (diagonal) preconditioner is not affected in these situations, since its sparsity pattern and performance do not change after permutation. However, in general, it will not

improve the condition of a problem as much as block Jacobi. In the experiments in Section 6.2.3, we use both block Jacobi and Jacobi preconditioners.

We can also ask whether it is worth it to sacrifice a balanced load distribution (by maintaining the reconstructed data in the reconstructing node for the rest of the solution process) for the sake of keeping an optimized preconditioner. A better strategy for the recovery of the preconditioner in more general cases remains future work.

Recovery without replacement nodes based on row-column permutation

In this section, we focus on recovery from node failures without replacement nodes using row and column permutations.

The recovery algorithm without a reconstruction node is outlined in Alg. 3. The reconstruction stage, from Line 2 to Line 10, performs the same reconstruction task as the algorithms described in Section 5.1.2, where the lost static data is retrieved in a reconstructing node, and there, the missing dynamic data is reconstructed. Line 13 represents the definition of the permutation $\boldsymbol{T}$ of rows and columns, thus, the way the recovered matrix will be redistributed among the surviving nodes.

The redistribution stage encompasses Lines 15, 16 and 17 of Alg 3. To scatter matrices and vectors, rows of the matrix and entries of the vectors are divided into blocks of uniform size, keeping the original order of rows and entries. These blocks are scattered among the surviving nodes. The received blocks are then attached after the local rows of the matrix, or after the local elements of the vectors. The scattering, of course, requires global communication.

After the scattering is complete, each processor reorders the columns of its local matrix according to the permutation $\boldsymbol{T}$. This is a local operation.

Once entries of the new matrix are in place, it is necessary to reconstruct the communication structures for the SpMV. This requires global communication.

Finally, the preconditioner is recomputed for the new, reordered system matrix. Since we work with block Jacobi, with blocks that do not cross the boundaries of the processes' index sets, this is a local operation.

6.2.3 Experiments

In this section, we summarize our experimental evaluations.

Implementation

Experiments are run with our own implementation of resilient PCG, programmed by us from scratch. In previous work [Pac+19], we performed an experimental evaluation of ESR using PETSc [Abh+18]. PETSc is a well known and very optimized library. Operations

Algorithm 3 ESR phase without a spare node for the PCG method using redistribution based on row-column permutation

1: **if** Working in a reconstruction node **then**
2: Retrieve the static data $\boldsymbol{A}_{I_f,I}$, and $\boldsymbol{b}_f$
3: Retrieve or recompute $\boldsymbol{P}_{I_f,I}$
4: Gather $\boldsymbol{r}^{(j)}_{I\setminus I_f}$ and $\boldsymbol{x}^{(j)}_{I\setminus I_f}$
5: Retrieve the redundant copies of $\beta^{(j-1)}$, $\boldsymbol{p}^{(j-1)}_f$, and $\boldsymbol{p}^{(j)}_f$
6: Compute $\boldsymbol{z}^{(j)}_f \coloneqq \boldsymbol{p}^{(j)}_f - \beta^{(j-1)}\boldsymbol{p}^{(j-1)}_f$
7: Compute $\boldsymbol{v} \coloneqq \boldsymbol{z}^{(j)}_f - \boldsymbol{P}_{I_f,I\setminus I_f}\boldsymbol{r}^{(j)}_{I\setminus I_f}$
8: Solve $\boldsymbol{P}_{I_f,I_f}\boldsymbol{r}^{(j)}_f = \boldsymbol{v}$ for $\boldsymbol{r}^{(j)}_f$
9: Compute $\boldsymbol{w} \coloneqq \boldsymbol{b}_f - \boldsymbol{r}^{(j)}_f - \boldsymbol{A}_{I_f,I\setminus I_f}\boldsymbol{x}^{(j)}_{I\setminus I_f}$
10: Solve $\boldsymbol{A}_{I_f,I_f}\boldsymbol{x}^{(j)}_f = \boldsymbol{w}$ for $\boldsymbol{x}^{(j)}_f$
11: **end if**
12: Create communicator for surviving nodes
13: Define permutation $\boldsymbol{T}$
14: **if** Working on a surviving node **then**
15: Scatter recovered rows of $\boldsymbol{A}_{I_f,I}$. Produces $\boldsymbol{TA}$
16: Scatter recovered $\boldsymbol{x}^{(j)}_f$, $\boldsymbol{r}^{(j)}_f$, $\boldsymbol{z}^{(j)}_f$, $\boldsymbol{p}^{(j)}_f$ and $\boldsymbol{b}_f$ acc. to $\boldsymbol{T}$
17: Permute columns of $\boldsymbol{TA}$. Produces $\boldsymbol{TAT}^{-1}$.
18: Recompute preconditioner $\boldsymbol{P}$ for system matrix $\boldsymbol{TAT}^{-1}$
19: **end if**

on sparse matrices are not as well optimized in our implementation, and thus we observe larger overheads for the resilience and recovery. However, our framework is sufficient to show the feasibility of the strategies in this work, and to compare recovery with and without a replacement node. Furthermore, it has been written for the purpose of implementation of our algorithms from the ground up, providing us with more flexibility than a PETSc-based implementation.

Some linear algebra functionality is provided by GSL [Gou09]. Matrices and right-hand-side vectors are stored using the PETSc binary format [Abh+18; Bal+97]. This format allows us to read only the relevant rows of a matrix. Matrices are stored in memory in compressed row (*CRS*) and compressed column storage formats (*CCS*). The solver itself operates using CCS format, but, as they are read from the binary file in the reconstruction stage, they are stored in CRS format. The CRS format also makes it easier to transfer matrix rows between processes in the redistribution stage. After the matrix data is transferred, local CCS matrices are formed after concatenation of the surviving data and the received rows of the reconstructed matrix. The permutation of columns required in Line 17 of Alg. 3 is then performed on a CCS-format matrix.

In our implementation, node failures are simulated instead of having the program stopped in the affected nodes. How node failures are represented depends on whether we are working with replacement nodes or not. A node failure that is repaired using a replacement node will be simulated by the corresponding node erasing its local data, performing the reconstruction and continuing the operation of PCG. In the case without replacement nodes, we represent a failing node by excluding the node from the communicator and having it return from all function calls. One of the neighbors of the lost nodes, now in the surviving communicator, will perform the reconstruction, the data is then redistributed among the nodes that are still active and the solver continues. Even though we do not work with real node failures, there are no fundamental obstacles towards a framework that can do this. The MPI extension ULFM [Mes17; Bla+13] provides functions to query the state of the ranks of a communicator and to create a new communicator composed of surviving nodes only. This should make it possible to produce a complete implementation of node-failure-tolerant PCG based on ESR.

Test problems and setup

We will now describe the test problems and solver setup used to run our experiments. We run experiments on problems from the SuiteSparse Matrix collection [DH11]. The problems are selected from the small to middle-sized SPD matrices available there, trying to pick from a wide array of application areas. Their properties are summarized in Table 6.4.

We use Jacobi and block Jacobi preconditioners. As mentioned in Section 6.2.2, such a preconditioner can be recomputed after the redistribution operation and maintain the

Table 6.4: Sample problems used in this section. The Name column is their ID in the SuiteSparse Matrix Collection. The ID column is a shorthand used when presenting the results. Size and NNZ are the number of row/columns and the number of non-zeros of the matrix, respectively.

Name	ID	Size	NNZ	Domain
BenElechi1	M9	245 874	13 150 496	2D/3D
pwtk	M10	217 918	11 524 432	Structural
boneS01	M11	127 224	5 516 602	Model reduction
offshore	M12	259 789	4 242 673	Electromagnetics
cfd2	M13	123 440	3 085 406	CFD
qa8fm	M14	66 127	1 660 579	Acoustics
2cubes_sphere	M15	101 492	1 647 264	Electromagnetics
G2_circuit	M16	150 102	726 674	Circuit sim.

condition of the permuted matrix. The solver runs with a relative tolerance of 10^{-8}, that is, we consider it converged when the relative residual norm ($\|\boldsymbol{r}\|_2/\|\boldsymbol{b}\|_2$) has decreased by a factor of 10^8. The inner solver used for the solution of the linear systems of lines 8 and 10 of Alg. 3 has a tighter relative tolerance of 10^{-12}.

To insure comparability of the results, static data of the lost nodes (the corresponding matrix rows and right-hand-side entries) is loaded from disk for both cases: with and without spare nodes.

The experiments are run on 32 nodes of the Hydra cluster of the Parallel Computing group at TU Wien. Hydra consists of 36 nodes (Dual 16-core Intel Xeon Gold 6130 processors at 2.1GHz), with dual-lane Intel OmniPath network interconnect. We run a single process per node. This is sufficient for our purposes, since this setup performs the communication between nodes that we are interested in. All experiments for a given problem matrix are run in the same set of nodes of the cluster.

We use the following libraries:

- OpenMPI 3.1.3
- GSL 2.5

. We use GCC 8.3.0, with optimization flags `-O3`.

Even though 32 nodes is a small number, it means that, for a fixed-size problem, a node failure in our setup causes an larger fraction of the data to be lost than in a larger number of nodes, which makes this a very relevant study case. The cost of reconstruction in ESR is comparable to a single iteration of the solver [PLG18; Pac+19]. The communication operations required for the row-column redistribution of the matrix and vectors are in the order of magnitude of an iteration of PCG (`MPI_Scatterv` on data the size of a global

vector, as well as with the number of non-zeros of the reconstructed rows of the matrix). We also load only the required data during the reconstruction phase by using binary, random access file formats. Thus, the methods described here scale well for very large machines.

We run experiments without any node failure, as well as experiments with node failures simulating recovery with and without a replacement node. In the latter case, we use the row-column redistribution strategy described in Section 6.2.2. If present, node failures are introduced after the solver has completed 50% of the number of iterations that PCG requires to reach convergence without node failures. We run separate experiments for node failures introduced in nodes with ranks 8, 16 and 24. Experiments are repeated nine times for each setting.

6.2.4 Experimental results

The results for a block Jacobi preconditioner with blocks of size 10 are summarized in Table. 6.5.

Both approaches have some overhead compared to the solution time achieved without fault-tolerance and in the absence of node failures, designated t_0 in Table 6.5. This could likely be reduced by further optimizing our framework. In [Pac+19], better overheads are achieved when using PETSc, a highly optimized library that provides some of the linear algebra functionality required for ESR.

We find that the time spent in the recovery phase as a fraction of the total solver time is relatively small. For most of our problems the fraction is less than 10% for the row-columns recovery without replacement nodes. For problems that take a very short time to solve, the recovery time reaches 87.5% of the total solution time for the M14 matrix. This happens because the time to retrieve data from the hard disk is itself large in comparison to the solver time. However, the solution time for M14 and M15, as well as their recovery time, takes just a fraction of a second.

The solution time with and without node failures is quite comparable. This time is dominated by the access to the hard disks to retrieve the binary matrix file, which is performed in both cases and, along with the time required to set up the communication schedule for the linear system, overshadows the cost of redistributing the information among the nodes for the recovery without replacement nodes.

We notice that the solver time can differ considerably between the recovery with and without spare nodes, and that these changes are also reflected in the number of iterations until convergence. For matrices M12-M16, we can argue that the solve time, including the recovery phase, are roughly the same. For M9, M10 and M11, reconstruction without a replacement node causes significant overhead compared with the in-place reconstruction with a replacement node. As discussed in Section 6.2.2, the way the preconditioner is built in the recovery phase can have an effect on the solver, and Table 6.5 shows that the

differences can be very visible.

To confirm that the differences are caused by the use of block Jacobi as a preconditioner after the recovery process, we perform the same experiments with the Jacobi preconditioner, which, as discussed in Section 6.2.2, should not change with the reordering of the matrix. The results are presented in Table 6.6.

Working with a Jacobi preconditioner, the number of iterations after recovery with a replacement node, and after recovery without a replacement node, are almost the same (as expected for the ESR approach). Differences are caused by the reordering of the data, and the fact that floating-point arithmetic is non-commutative. Just as with block Jacobi, the recovery time either goes up to about 10% of the solution time for most cases or it is just a fraction of a second if the problem takes very little time to be solved. The table shows that, when using a preconditioner more suitable for our recovery strategy, the overhead with respect to ESR with spare nodes is much smaller. In our tests, the maximum relative difference of the average solver runtime of ESR without spare nodes, in relation to the one of ESR with spare nodes, is 9.1% (which happens for M14 in Table 6.6, if process 16 fails) with the average overall being 3.5%.

The results presented in Tables 6.5 and 6.6 show a very minor influence of the location of the lost node on the recovery time.

Fig. 6.7 shows the relative overhead of the two approaches compared to the standard, non-resilient PCG solver: with and without spare nodes. The figure shows that the overall overhead for the two approaches decreases with increasing runtime. The overhead increases faster for the recovery without a replacement node for larger, more time-consuming problems. As explained in Section 6.2.3, we expect that the further we optimize our implementations, the smaller this overhead becomes.

Fig. 6.8 shows a similar plot, now for the Jacobi preconditioner. Without changes in the behaviour of the preconditioner, the curves behave in the same way. The recovery without a replacement node still has a slightly larger overhead, particularly visible for problems that are solved more quickly, caused by the overhead of disk access and the communication between nodes for redistribution.

In Fig. 6.9 we compare the relative overhead of a solver that encounters a node failure and recovers without the use of spare nodes, and that of a solver that recovers using spare nodes. This represents the relative cost of not using spare nodes for the reconstruction. In Figures 6.7 and 6.8 we can see that the runtimes for matrices M14 and M15 are very small, in the order of hundreds of a second, and thus we can ignore their large relative runtime overheads. For this reason, we exclude them from Fig. 6.9.

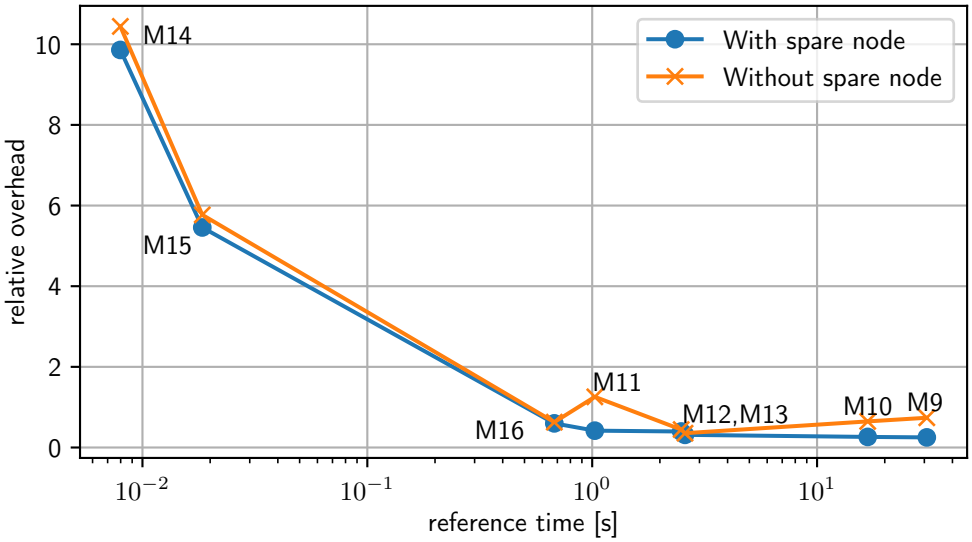


Figure 6.7: Relative overhead $(t_s - t_0)/t_0$ for matrices sorted by increasing reference time t_0 (cf. caption of Table 6.5) using a block Jacobi preconditioner with block size 10. Values for each matrix are the average over all failed node indexes and repetitions. As expected, the overhead for recovery without spare nodes (orange) is above the recovery with a replacement node (blue). The overhead without spare nodes is, at points, non-negligible. This does not have to do with the recovery process itself, and is better for a more suitable preconditioner. Overheads for very small runtimes tend to be large. This is caused by I/O (disk access) and due to the fact that our code is not yet well optimized. The relative overhead over non-resilient PCG is not the focus of this work. For more information, refer to [PLG18; Pac+19].

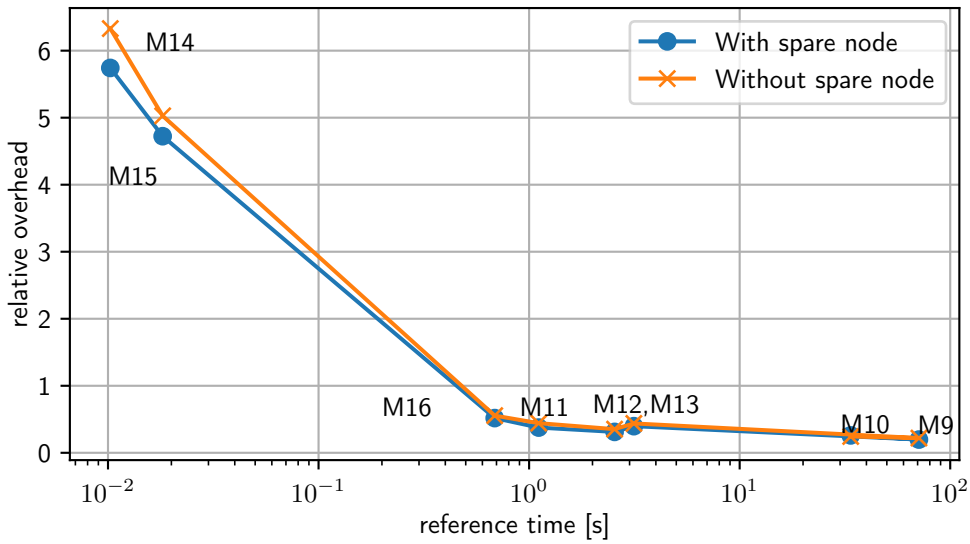


Figure 6.8: Relative overhead $(t_s - t_0)/t_0$ for matrices sorted by increasing reference time t_0 (cf. caption of Table 6.5) using a Jacobi (diagonal) preconditioner. Values for each matrix are the average over all failed node indexes and repetitions. Jacobi is a more suitable preconditioner for our approach (cf. Section 6.2.2), so, in contrast to Fig. 6.7, the curves are now close together, indicating small differences between the two approaches.

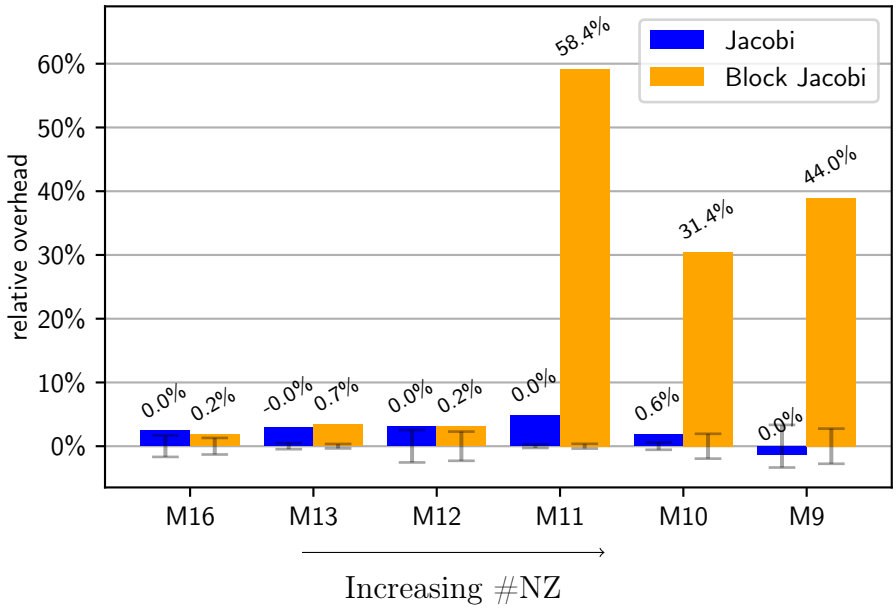


Figure 6.9: Mean relative runtime overhead of the solution without the use of spare nodes, including the ESR recovery process, with respect to the corresponding solver using spare nodes. Both the Jacobi and block Jacobi preconditioners are presented. The relative excess in iterations of the solver without spare nodes with respect to the solver with spare nodes is presented as a percentage above each bar. The standard deviation of the runtime of the solver with spare nodes is marked as a black error bar around zero.

Table 6.5: Runtimes for the test problems of Table 6.4 with a block Jacobi preconditioner with a block size of 10. t_0 refers to the average time to convergence for a PCG solver without ESR resilience in a failure-free run. i_f is the index of the node that fails. It_{conv} is the number of iterations it takes the solver to converge to the relative tolerance. t_r is the time spent in the recovery phase, and t_s is the time used for the solver, including the recovery phase. Values after the $\pm$ symbol are the standard deviation of the measurements.

ID	t_0 [s]	i_f	Recovery with spare node				Recovery without spare node			
			It_{conv}	t_s [s]	t_r [s]	t_r/t_s [%]	It_{conv}	t_s [s]	t_r [s]	t_r/t_s [%]
M9	30.52	8	27760	38.57 $\pm$ 0.74	0.85 $\pm$ 0.01	2.2	40678	54.17 $\pm$ 0.40	0.87 $\pm$ 0.01	1.6
		16	27758	37.85 $\pm$ 0.97	0.89 $\pm$ 0.01	2.4	39389	52.43 $\pm$ 0.44	0.91 $\pm$ 0.01	1.7
		24	27758	38.65 $\pm$ 1.11	1.06 $\pm$ 0.03	2.7	39886	53.67 $\pm$ 0.64	1.07 $\pm$ 0.01	2.0
M10	17.00	8	16710	21.25 $\pm$ 0.49	0.79 $\pm$ 0.02	3.7	20814	26.20 $\pm$ 0.27	0.80 $\pm$ 0.01	3.0
		16	16704	21.24 $\pm$ 0.39	0.79 $\pm$ 0.01	3.7	21357	26.87 $\pm$ 0.24	0.81 $\pm$ 0.01	3.0
		24	16716	21.00 $\pm$ 0.31	0.72 $\pm$ 0.02	3.4	23704	29.67 $\pm$ 0.20	0.74 $\pm$ 0.02	2.5
M11	1.03	8	1858	1.45 $\pm$ 0.01	0.20 $\pm$ 0.01	13.6	3024	2.38 $\pm$ 0.01	0.21 $\pm$ 0.01	8.6
		16	1858	1.46 $\pm$ 0.01	0.20 $\pm$ 0.01	13.9	2657	2.11 $\pm$ 0.01	0.21 $\pm$ 0.01	10.2
		24	1858	1.45 $\pm$ 0.01	0.20 $\pm$ 0.01	13.5	3147	2.46 $\pm$ 0.01	0.20 $\pm$ 0.01	8.3
M12	2.59	8	1877	3.51 $\pm$ 0.02	0.44 $\pm$ 0.01	12.5	1883	3.63 $\pm$ 0.02	0.45 $\pm$ 0.01	12.5
		16	1873	3.33 $\pm$ 0.02	0.27 $\pm$ 0.01	8.1	1879	3.44 $\pm$ 0.02	0.29 $\pm$ 0.01	8.3
		24	1878	3.35 $\pm$ 0.02	0.28 $\pm$ 0.01	8.3	1879	3.45 $\pm$ 0.02	0.29 $\pm$ 0.01	8.5
M13	2.49	8	6145	3.48 $\pm$ 0.01	0.26 $\pm$ 0.01	7.3	6186	3.59 $\pm$ 0.02	0.26 $\pm$ 0.01	7.4
		16	6146	3.47 $\pm$ 0.02	0.25 $\pm$ 0.01	7.2	6191	3.60 $\pm$ 0.01	0.26 $\pm$ 0.01	7.2
		24	6141	3.47 $\pm$ 0.01	0.25 $\pm$ 0.01	7.2	6180	3.59 $\pm$ 0.02	0.26 $\pm$ 0.01	7.2
M14	0.008	8	33	0.086 $\pm$ 0.001	0.075 $\pm$ 0.001	87.1	33	0.090 $\pm$ 0.001	0.079 $\pm$ 0.001	87.3
		16	33	0.088 $\pm$ 0.001	0.077 $\pm$ 0.001	87.5	33	0.092 $\pm$ 0.001	0.081 $\pm$ 0.001	87.5
		24	33	0.086 $\pm$ 0.001	0.074 $\pm$ 0.001	87.1	33	0.090 $\pm$ 0.001	0.078 $\pm$ 0.001	86.9
M15	0.019	8	29	0.120 $\pm$ 0.001	0.095 $\pm$ 0.001	81.3	29	0.120 $\pm$ 0.001	0.100 $\pm$ 0.001	81.4
		16	29	0.120 $\pm$ 0.001	0.098 $\pm$ 0.001	81.7	29	0.130 $\pm$ 0.001	0.100 $\pm$ 0.001	81.7
		24	29	0.120 $\pm$ 0.001	0.098 $\pm$ 0.001	81.6	29	0.130 $\pm$ 0.001	0.100 $\pm$ 0.001	81.7
M16	0.68	8	1745	1.07 $\pm$ 0.01	0.13 $\pm$ 0.01	11.8	1750	1.10 $\pm$ 0.01	0.13 $\pm$ 0.01	12.1
		16	1745	1.09 $\pm$ 0.01	0.15 $\pm$ 0.01	14.2	1749	1.11 $\pm$ 0.01	0.16 $\pm$ 0.01	14.5
		24	1745	1.10 $\pm$ 0.01	0.16 $\pm$ 0.01	14.8	1747	1.12 $\pm$ 0.01	0.17 $\pm$ 0.01	15.3

Table 6.6: Runtimes for the test problems of Table 6.4 with a Jacobi preconditioner. The variables used in the header are the same as in Table 6.5.

ID	t_0 [s]	i_f	Recovery with spare node				Recovery without spare node			
			It_{conv}	t_s [s]	t_r [s]	t_r/t_s [%]	It_{conv}	t_s [s]	t_r [s]	t_r/t_s [%]
M9	33.88	8	33799	42.19 ± 0.84	0.75 ± 0.01	1.8	33802	42.17 ± 0.89	0.76 ± 0.01	1.8
		16	33795	42.98 ± 1.70	0.79 ± 0.01	1.8	33797	41.96 ± 0.93	0.80 ± 0.01	1.9
		24	33800	42.08 ± 0.37	0.94 ± 0.01	2.2	33803	42.27 ± 0.92	0.97 ± 0.01	2.3
M10	71.0	8	77239	85.17 ± 0.60	0.68 ± 0.01	0.8	78581	87.45 ± 0.20	0.70 ± 0.01	0.8
		16	77193	85.07 ± 0.55	0.69 ± 0.01	0.8	77272	86.20 ± 0.28	0.71 ± 0.01	0.8
		24	77229	86.13 ± 3.60	0.61 ± 0.01	0.7	77238	86.27 ± 0.26	0.64 ± 0.01	0.7
M11	1.11	8	2161	1.52 ± 0.01	0.16 ± 0.01	10.3	2162	1.61 ± 0.01	0.16 ± 0.01	10.3
		16	2162	1.53 ± 0.01	0.16 ± 0.01	10.7	2162	1.61 ± 0.01	0.17 ± 0.01	10.8
		24	2162	1.53 ± 0.01	0.16 ± 0.01	10.3	2162	1.59 ± 0.01	0.16 ± 0.01	10.3
M12	2.55	8	1953	3.45 ± 0.02	0.42 ± 0.01	12.0	1953	3.57 ± 0.02	0.43 ± 0.01	12.0
		16	1953	3.28 ± 0.02	0.24 ± 0.01	7.4	1953	3.37 ± 0.02	0.26 ± 0.01	7.6
		24	1953	3.29 ± 0.02	0.25 ± 0.01	7.7	1953	3.41 ± 0.03	0.27 ± 0.01	7.8
M13	3.14	8	8705	4.39 ± 0.02	0.22 ± 0.01	4.9	8705	4.51 ± 0.02	0.23 ± 0.01	5.0
		16	8704	4.39 ± 0.03	0.21 ± 0.01	4.8	8704	4.52 ± 0.02	0.22 ± 0.01	4.8
		24	8706	4.40 ± 0.02	0.21 ± 0.01	4.8	8702	4.54 ± 0.02	0.22 ± 0.01	4.8
M14	0.010	8	48	0.068 ± 0.001	0.054 ± 0.001	79.4	48	0.073 ± 0.001	0.059 ± 0.001	80.1
		16	48	0.071 ± 0.002	0.057 ± 0.001	79.8	48	0.078 ± 0.002	0.062 ± 0.001	80.5
		24	48	0.068 ± 0.001	0.054 ± 0.001	79.4	48	0.074 ± 0.001	0.059 ± 0.001	80.1
M15	0.018	8	30	0.10 ± 0.01	0.082 ± 0.010	79.6	30	0.11 ± 0.01	0.087 ± 0.010	80.0
		16	30	0.11 ± 0.01	0.084 ± 0.010	79.9	30	0.11 ± 0.01	0.088 ± 0.010	80.4
		24	30	0.10 ± 0.01	0.083 ± 0.010	79.7	30	0.11 ± 0.01	0.087 ± 0.010	80.1
M16	0.69	8	2027	1.02 ± 0.02	0.07 ± 0.01	6.9	2027	1.06 ± 0.01	0.08 ± 0.01	7.1
		16	2027	1.04 ± 0.01	0.10 ± 0.01	9.6	2027	1.06 ± 0.01	0.11 ± 0.01	9.9
		24	2027	1.06 ± 0.01	0.11 ± 0.01	10.2	2027	1.08 ± 0.01	0.11 ± 0.01	10.5

6.3 Reduction of the frequency at which redundant information is stored

In this section, we present work published in [Pac+20]. As a main contribution of that paper, we illustrate similarities between ESR and checkpoint/restart (CR), frame it as an instance of what we call *algorithm-based checkpoint/restart*, and describe how it can be restructured to enable decreased state-storage frequencies. We experimentally show that this approach reduces the runtime overhead in the absence of node failures. This is particularly beneficial in scenarios with multiple node failures, where the additional communication needs increase the overhead most drastically and, consequently, for which the runtime overhead reduction is greatest.

For the description of the methods in this section, we introduce the concept of *redundant copies*:

Redundant copies After the augmented SpMV (ASpMV) is executed, as discussed in Section 5.1.1, the redundant information of the search direction $\boldsymbol{p}^{(j)}$ is not explicitly available. This means, even though the information of the copies is present and spread in the cluster, after a node failure, the data must be gathered in a replacement node before we can work with $\boldsymbol{p}^{(j)}$ again. We introduce the concept of a *redundant copy*, designated with a prime symbol ($'$), as in $\boldsymbol{p'}^{(j)}$, to abstractly represent the redundant vector data in the cluster, in whatever storage scheme the framework utilizes. We do not specify the number of copies per vector entry that a redundant copy represents, since we do not need this information for further descriptions. The concept of redundant copies will be used to explain our algorithms in Section 6.3.3.

6.3.1 Function notation for the augmented SpMV

For the description of the algorithms in Section 6.3, we refer to the ordinary SpMV as $\boldsymbol{\varrho} \coloneqq \texttt{SpMV}(\boldsymbol{A}, \boldsymbol{p})$, a function that takes matrix $\boldsymbol{A}$ and vector $\boldsymbol{p}$ as inputs and returns their product, $\boldsymbol{\varrho}$. The augmented variant is represented as $\boldsymbol{\varrho} \coloneqq \texttt{ASpMV}(\boldsymbol{A}, \boldsymbol{p}, \phi, Q)$, where the function additionally takes the number of desired redundant copies ϕ, and the queue Q where it will push a new *redundant copy* for $\boldsymbol{p}$.

6.3.2 Problem definition

We work on the solution of sparse, symmetric, positive definite linear systems in a distributed-memory setting using the PCG method. The solver data is distributed, and the problem is solved in N nodes of a computer cluster. Disjoint subsets I_s of consecutive indices are distributed among the nodes, such that their union forms the set of all indices, I. Node s

is assigned the rows of the matrix, and entries of the vectors, corresponding to the indices in its subset I_s. This *block row distribution* is common, and is used in prominent libraries such as PETSc [Bal+18]. The scalars used by the solver are replicated in all nodes.

We consider a situation in which the computer cluster is unreliable, specifically, that it can suffer node failures, in which one or more nodes can fail simultaneously. In terms of the linear solver, these faults represent the loss of information owned by the affected nodes: blocks of vector entries, matrix rows, and the scalars also residing in the nodes' memory.

We look for strategies to recover from these events and still converge to the correct solution of the linear system. In this work, the cost metric is the overall runtime for the solver to converge. We assume the availability of sufficient memory to hold redundant information, and of spare nodes in the cluster.

6.3.3 ESR with periodic storage

We extend the ESR approach introduced in Section 5 to perform iterations with ASpMV with a reduced frequency, that is, not in every iteration, but two consecutive times every T iterations. We call T the *checkpointing interval* to keep with CR terminology, and we refer to the set of two iterations in which redundant information is stored as the *storage stage*. In the event of a node failure, the solver will return to the last time the search directions for two successive iterations were stored redundantly via ASpMV. It is then possible to reconstruct the state for the last of those iterations. We call the new approach *Exact state reconstruction with periodic storage* (ESRP), and contrast it with ESR which stores data in every iteration.

To describe ESRP, we introduce the concept of a *queue*, where the solver stores redundant copies (see start of Section 6.3). In the ESR algorithm, this queue has space for two positions: Every iteration, ASpMV will push a new redundant copy into the queue, and the oldest copy will be released. This queue thus contains the redundant copies of search directions for two successive iterations.

We now examine the same procedure in the case of ESRP. Suppose that the last redundant copies held in the queue are $\boldsymbol{p}'^{(j)}$ and $\boldsymbol{p}'^{(j+1)}$, that we performed some additional iterations using regular SpMV afterwards, and that then a node failure occurs. The search directions in the queue could be used to reconstruct the state for iteration $j+1$. However, it is possible that the node failure takes place after only one of the two iterations of a storage stage have been completed. Suppose that the solver has reached a storage stage at some iteration j, and the first call to `ASpMV` is performed. The redundant copy $\boldsymbol{p}'^{(j)}$ is then pushed to the queue. If a node failure happens at this point in time, before the redundant copy $\boldsymbol{p}'^{(j+1)}$ is created, the vector $\boldsymbol{p}^{(j)}$ that we can retrieve is not sufficient to perform the reconstruction shown in Alg. 2, since we would additionally need $\boldsymbol{p}^{(j+1)}$. For this reason, it is necessary to have a queue of not two, but three redundant copies of search directions,

such that, if this happens, the queue still contains entries of two successive search directions from a redundant storage period before.

In addition to the redundant copies created during the ASpMV, the solver needs to duplicate some local data at each node during the storage stage. As can be seen in Line 4 of Alg. 2, to reconstruct the vector $\boldsymbol{z}^{(j)}$, and subsequently the state of the solver for iteration j, the value of $\beta^{(j-1)}$ is needed. However, depending on when the node failure occurs, the scalar β may have changed since the last storage stage. It is therefore necessary to create a duplicate of the value of $\beta^{(j-1)}$. Similarly, the local entries of the residual $\boldsymbol{r}$, the preconditioned residual $\boldsymbol{z}$, the iterand $\boldsymbol{x}$ and the search direction $\boldsymbol{p}$ at iteration j must also be duplicated in all nodes, so that they can be used for the reconstruction process and so that the surviving nodes can reset their own parts of the solver state to match the state that is reconstructed at the replacement nodes. Entries of the vector $\boldsymbol{z}^{(j)}$ corresponding to surviving nodes are not used during the reconstruction, and could also be recomputed from $\boldsymbol{r}^{(j)}$ once the latter has been reconstructed. However, our solver stores a local copy instead of performing this operation. We mark these duplicate values with an asterisk: $\beta*$ is a scalar, and $\boldsymbol{r}*$, $\boldsymbol{p}*$, $\boldsymbol{z}*$ and $\boldsymbol{x}*$ are distributed vectors. There is no need to store the scalar α: It is not used during the reconstruction and it will be computed in Line 13 of Alg. 4 when the solver continues iterating. Since these copies are created locally by each node, they do not introduce any additional communication between nodes, and the runtime overhead they cause is therefore negligible. Note that since $\boldsymbol{r}*$, $\boldsymbol{p}*$, $\boldsymbol{z}*$ and $\boldsymbol{x}*$ are created by each node copying its own data, if a node fails, the copies contained in it are also lost. Therefore, these copies, by themselves, obviously cannot be used to reconstruct the state.

An example of the procedure follows: The solver has a queue Q with three positions to hold redundant copies. Initially, the queue is empty: $Q \coloneqq [\_, \_, \_]$. After T iterations, `ASpMV` will be called for the first time and will push the first redundant copy, thus Q contains $[\_, \_, \boldsymbol{p}'^{(T)}]$. The solver will also create a copy of the value of $\beta^{(T)}$, $\beta*$, on every node. At this point, it is not possible to recover from a node failure using ESRP. After another iteration, `ASpMV` pushes another redundant copy, Q becomes $[\_, \boldsymbol{p}'^{(T)}, \boldsymbol{p}'^{(T+1)}]$, and copies of the vectors $\boldsymbol{r}^{(T+1)}$, $\boldsymbol{z}^{(T+1)}$, $\boldsymbol{x}^{(T+1)}$ and $\boldsymbol{p}^{(T+1)}$ are made, respectively designated $\boldsymbol{r}*$, $\boldsymbol{z}*$, $\boldsymbol{x}*$ and $\boldsymbol{p}*$. This information can now be used to reconstruct the state for iteration $T+1$, and the storage stage ends. From there, we continue iterating using regular `SpMV`.

When the next storage stage is reached, after an additional T iterations, we use `ASpMV` again, and Q becomes $[\boldsymbol{p}'^{(T)}, \boldsymbol{p}'^{(T+1)}, \boldsymbol{p}'^{(2T)}]$, and again, a copy of $\beta^{(2T)}$ is created. The newest entry of Q cannot be used in conjunction with the previous ones to reconstruct the state of the solver. In the event of a node failure at this point, the state would be recovered for iteration $T+1$. At this time, we still need the value of $\beta^{(T)}$ to reconstruct the state, so we may not overwrite $\beta*$ yet, and will overwrite it in the next iteration instead. After

an additional successful iteration, the queue contains $[\boldsymbol{p'}^{(T+1)}, \boldsymbol{p'}^{(2T)}, \boldsymbol{p'}^{(2T+1)}]$, and copies for $\boldsymbol{r}^{(2T+1)}$, $\boldsymbol{z}^{(2T+1)}$, $\boldsymbol{x}^{(2T+1)}$ and $\boldsymbol{p}^{(2T+1)}$ are created. From that point on, it is possible to reconstruct for iteration $2T+1$, which concludes the second storage stage. The process is presented in Alg. 4 and graphically represented in Fig. 6.10.

Algorithm 4 Preconditioned conjugate gradients (PCG) method with periodic redundant storage (for ESRP). We use the function notation for ASpMV discussed in Sec. 6.3.1.

1: $\boldsymbol{r}^{(0)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(0)}, \boldsymbol{z}^{(0)} := \boldsymbol{P}\boldsymbol{r}^{(0)}, \boldsymbol{p}^{(0)} := \boldsymbol{z}^{(0)}, j := 0,$
$Q := [\_, \_, \_]$
2: **for** $j = 0, 1, \ldots,$ until convergence **do**
3: **if** $j \bmod T = 0$ and $j > 2$ **then**
4: $\boldsymbol{\varrho}_j := \texttt{ASpMV}(\boldsymbol{A}, \boldsymbol{p}^{(j)}, \phi, Q)$
5: $\beta{*}{*} = \beta^{(j)}$
6: **else if** $(j-1) \bmod T = 0$ and $j > 2$ **then**
7: $\boldsymbol{\varrho}_j := \texttt{ASpMV}(\boldsymbol{A}, \boldsymbol{p}^{(j)}, \phi, Q)$
8: $\boldsymbol{x}{*} = \boldsymbol{x}^{(j)}$, $\boldsymbol{r}{*} = \boldsymbol{r}^{(j)}$, $\boldsymbol{z}{*} = \boldsymbol{z}^{(j)}$, $\boldsymbol{p}{*} = \boldsymbol{p}^{(j)}$
9: $\beta{*} = \beta{*}{*}$
10: **else**
11: $\boldsymbol{\varrho}_j := \texttt{SpMV}(\boldsymbol{A}, \boldsymbol{p}^{(j)})$
12: **end if**
13: $\alpha^{(j)} := \boldsymbol{r}^{(j)\intercal}\boldsymbol{z}^{(j)} / \boldsymbol{p}^{(j)\intercal}\boldsymbol{\varrho}_j$
14: $\boldsymbol{x}^{(j+1)} := \boldsymbol{x}^{(j)} + \alpha^{(j)}\boldsymbol{p}^{(j)}$
15: $\boldsymbol{r}^{(j+1)} := \boldsymbol{r}^{(j)} - \alpha^{(j)}\boldsymbol{\varrho}_j$
16: $\boldsymbol{z}^{(j+1)} := \boldsymbol{P}\boldsymbol{r}^{(j+1)}$
17: $\beta^{(j)} := \boldsymbol{r}^{(j+1)\intercal}\boldsymbol{z}^{(j+1)} / \boldsymbol{r}^{(j)\intercal}\boldsymbol{z}^{(j)}$
18: $\boldsymbol{p}^{(j+1)} := \boldsymbol{z}^{(j+1)} + \beta^{(j)}\boldsymbol{p}^{(j)}$
19: **end for**

From Fig. 6.10 we can see which checkpointing intervals make sense for ESRP. For $T > 2$, we proceed as explained above. For $T = 2$, it no longer makes sense to use this approach, since redundant copies are created in every iteration. In this case, it is better to use ESR (Section 5). For $T = 1$, we can no longer talk about creating two successive copies. Again, this corresponds to regular ESR.

6.3.4 ESR/ESRP and checkpointing

The addition of a checkpointing interval to ESRP highlights its similarity to CR approaches. In both cases, we store the state of the solver after every checkpointing period, either explicitly, in the case of CR, or implicitly, by exploiting redundancy provided by the algorithm itself, in the case of ESRP. We claim, therefore, that ESRP is an *algorithm-based checkpoint/restart* strategy.

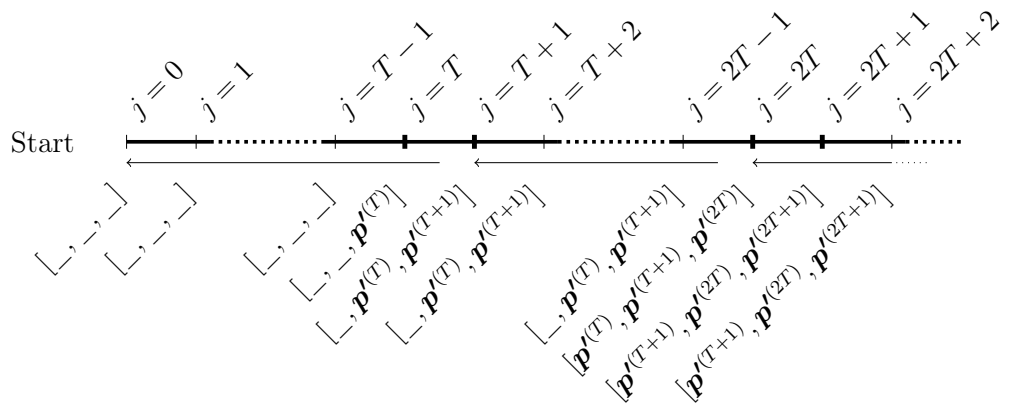


Figure 6.10: State of the redundancy queue for the search directions during the solution process. The lists below the line represent the state of the search direction queue, and which search directions are stored redundantly somewhere in the cluster. The thin arrows running leftwards show how far the solver has to revert in the event of a node failure.

As with CR, this method introduces a trade-off between the runtime overhead, which decreases with increasing T since we create redundant copies less frequently, and the cost of discarding the iterations performed since the last storage stage was reached (cf. Section 2.1).

In-memory checkpoint/restart

In our experiments in Section 6.3.5 we compare the runtime of our ESRP approach to an in-memory buddy checkpoint/restart strategy (IMCR), which will now be described in detail.

Similar to the description above, we assume a checkpoint interval of T: Once every T iterations, each node will create a checkpoint by sending a complete copy of the local parts of all vectors it owns to a neighboring node (the *buddy node*). In the event of a node failure, the replacement node will then simply retrieve its local vector parts from the buddy. As in the case of ESR, we assume that the static solver data can be retrieved from safe storage and does not need to be stored during the checkpointing. To extend this approach to support multiple node failures, it is sufficient to send the checkpointing data to multiple buddies. This is a form of *algorithm-based fault tolerance*, since the checkpointing and recovery strategies are specifically tailored to the PCG solver.

There are further similarities between this checkpointing strategy and ESR/ESRP: For example, the data that can be retrieved from a checkpointing buddy is the same data that is reconstructed during the recovery phase of ESR, and the strategy for choosing the buddies is the same as the strategy for determining the destinations of redundant elements in the augmented sparse matrix-vector product, as defined by Eq. 6.3. An important difference,

however, is that ESR mainly adds on to existing communication, while the checkpointing strategy introduces a completely new round of communication in each storage iteration.

6.3.5 Experimental evaluation

Implementation

We implement our algorithms using our own framework, written in C. In this way we achieve the highest flexibility for our purpose. Elementary linear algebra functionality is provided by GSL [Gou09], but parallelization of linear algebra operations, in particular of the matrix-vector product, is in-house code. Communication between nodes is realized with MPI. The framework is modularly structured, such that different strategies to achieve data redundancy, and to perform reconstruction and recovery, can be used. In particular, we can simulate ESRP, as well as in-memory CR.

We simulate one node failure event for each run of PCG. The ranks of the affected nodes and the iteration at which the failure should occur are passed as parameters to our framework. Once the marked iteration is reached, the nodes set to fail zero-out all their vector entries, as well as the scalars they contain, thus simulating the loss of all of their dynamic data (their components of vectors $\boldsymbol{x}$, $\boldsymbol{r}$, $\boldsymbol{z}$, $\boldsymbol{p}$, as well as the scalars β and α). This is also the initial state of a replacement node, which starts without knowledge of the state of the node it is replacing. For the sake of ease of implementation, nodes simulating a node failure will also act as the replacements. After a simulated node failure, the replacement nodes start the recovery process, collecting information from their neighbors and reconstructing the lost data.

In a real-world scenario, the replacement nodes would also have to reload the rows of the system matrix and the preconditioner, and the entries of the right-hand-side vector that they own; however, we have decided not to include this step in the measurement of the runtime overheads during our experiments. There are two main reasons for this decision. Firstly, this overhead depends too strongly on the individual use case for us to be able to make any generalized statements about it: Not only is it influenced by the matrix size and file system properties, but matrices might be stored in different file formats (e.g. plain text or binary), or the solver might be working with a matrixless representation altogether. This changes the loading time considerably. Secondly, the reloading step is the same for both the ESR and CR versions we investigate. Therefore, we would not gain any valuable information about differences in the behaviour of these strategies from examining the time required for the reloading of static data.

Beyond node-failure simulation Since we only simulate node failures, our framework does not capture all of the events that would take place in the case of a real incident,

nor all of the steps that are necessary to recover from one. We assume that, in a realistic application, there would be some middleware available to take care of these additional tasks. At any rate, we can describe the operations necessary to perform the recovery using ESR and CR that are not modeled in our framework.

The first of these tasks is detecting a node failure: A prerequisite for recovery for both ESR and CR is that there is a mechanism in place that will notice if one of the nodes becomes unresponsive. It is reasonable to assume that this cost would be the roughly the same for both approaches in a framework that is well-optimized for this purpose.

A second task currently not modeled is determining which node has failed. The surviving nodes need this information to decide what data must be transferred to the replacement node, and the replacement node needs it to know which rows of the matrix, the preconditioner, and which entries of the right-hand-side vector to load.

A third task would be to setup the cluster to continue working. For ESR as as well as the CR strategy described in this section, this involves providing a replacement node to take the place of the lost node and setting up a new communicator to continue iterating. In the case of ESR, it is also possible to proceed without a replacement node as discussed in Section 6.2. More generally speaking, depending on how exactly CR is implemented, it could make use of spare nodes, enabling the application to keep using most of the nodes already allocated to it, but making it necessary to identify the lost node just as in the case of ESR; or the whole application could be restarted on newly-allocated nodes, although this is likely to be more costly that identifying the lost nodes, particularly at greater scales [Cha+15; TH14; Hor+20].

All in all, we expect the costs of the events that our framework is not modeling to be comparable between ESR and CR.

Although presently not in the MPI standard, there is ongoing work on tools to deal with node failures. The *User-Level Fault Mitigation* (ULFM) library [Bla+13; Mes17] offers functions for detection of node failures and identification of the affected nodes. In conjunction with standard MPI, it is possible to create a new communicator on which the solver can continue working.

Experimental setup

Our experiments are run on the VSC3 machine of the Vienna Scientific Cluster. We use 128 nodes, with one process per node. (One process is sufficient to examine the overheads for resilience, since the redundant data has to be sent to different nodes in any case.) This machine has a fat-tree topology.

We use the following libraries: Intel C compiler 18.0.5, Intel MPI version 2018 update 4 and GSL 2.4.

If no protective measures are taken, node failures can cause the loss of all the com-

Table 6.7: Test matrices from [DH11]

Matrix	**Problem type**	**Problem size**	**#NZ**
Emilia_923	Structural	923 136	40 373 538
audikw_1	Structural	943 695	77 651 847

putation invested in the solution of a linear system. However, they are a relatively rare occurrence. With the incidence estimations of [HR15] (mean time between failures of 9 hours for 100 000 nodes, and 53 minutes for 1 000 000 nodes), a linear solver might be affected by at most a few of such events during its runtime. Thus, we consider that examining the behavior of the solver when a single node-failure event strikes at some point during its operation is a useful exercise.

We use a block Jacobi preconditioner, with non-overlapping blocks and all rows of a block belonging to a single node. The blocks are uniformly sized and we use as few of them as possible, with a maximum block size of 10. This preconditioner is used both for the linear system of the problem we are solving, and for the inner systems of the reconstruction (Lines 6 and 8 of Alg 2). The solver has converged once the relative residual $\|r\|_2/\|b\|_2$ is below 10^{-8}. The relative residual for the inner system for the reconstruction must reach 10^{-14} for convergence.

Our test problems are SPD matrices from the SuiteSparse Matrix Collection [DH11] (see Table 6.7). They were selected based on their size, to allow for comparisons with related work in [Pac+19]. With these matrices, we set up the test constellation as follows:

- Two recovery strategies: ESRP and in-memory CR.
- Checkpoint interval of 20, 50 and 100 iterations, plus an interval of 1 for ESRP, representing the previously existing ESR method.
- Resilience with 1, 3 and 8 redundant copies.
- Reference runs, runs with resilience but without node failures, and node failures introduced in contiguous blocks starting in ranks 0 and 64, with as many node failures as the solver can tolerate with the number of available copies.

We introduce a node failure in the interval between checkpoints that contains the iteration $C/2$, where C is the number of iterations that a failure-free solver needs to converge. Within this interval, the node failure is introduced two iterations before its end, thus representing a worst-case scenario in which most of the progress since the start of the interval is lost. Experiments are repeated at least five times for every setting in this test constellation.

The use of contiguous blocks of ranks for the node failures is justified by considering that multiple-nodes failures would most likely come from, for example, a switch fault, affecting a branch of the fat-tree and, consequently, a contiguous block of ranks.

Experimental results

We measure the runtime t of the solver to reach convergence to evaluate our algorithms. We define the reference time, t_0, as the median time for runs of a reference, non-resilient PCG solver. The metric that we present in our results is the relative overhead over this reference time: relative overhead $= (t - t_0)/t_0$. Our results are summarized in Tables 6.8 and 6.9, and shown in Fig. 6.11 and 6.12. Note that in all of our experiments the measured total runtime is less than a minute and, thus, well below an estimated mean time between failures of approximately an hour up to several hours in large-scale applications [HR15]. Since we have to recover from a node failure in that short runtime, the overhead due to this recovery is likely more severe than in a real-world application with less frequent recoveries. Therefore, we test a scenario that is *less favorable* for our approach. Reasonably low overheads in our experiments can hence be considered a proof of concept.

For the used test matrices, overheads of ESRP are usually much smaller than the ones of ESR (Section 5), and this advantage is more pronounced for larger numbers of redundant copies. In the case of ESRP, the columns for reconstruction overhead in Tables 6.8 and 6.9 show the cost of gathering the information and of recomputing the lost data. In the case of IMCR, these columns show the cost of communicating checkpointed data to the replacement nodes. For both test matrices, our experiments show a reconstruction overhead of basically zero for IMCR. This suggests that in our experimental setup the communication cost is considerably smaller than the computation cost. Keeping in mind the common understanding that communication tends to become more expensive than computation, especially in the large scale, this observation indicates that the experimental setup currently available to us possibly leads to underestimating the communication cost and does not allow for a solid comparison with IMCR. Consequently, we decided to depict experimental measurements for IMCR less prominently in Figs. 6.11 and 6.12 and leave more representative experiments at larger scales for future work.

The cluster that we use introduces a certain amount of variation to our measurements. We repeat each experiment to reduce the standard deviation of the tests, and we present their median. However, there are cases in which this standard deviation for the reference runtimes is not reduced below the overhead. Table 6.8 shows an instance of this: The median reference time for ESRP for three redundant copies is larger for a period of 50 than for a period of 20. In general, we expect larger checkpoint intervals to produce smaller overheads in the failure-free case, but these results can be affected by the variation in runtime from external factors. In this case, the overhead is so close to zero that it is

overshadowed by the noise from the machine.

Table 6.8 shows that, for the matrix `Emilia_923`, the failure-free overhead for ESRP is lower than for IMCR, down to about half of the corresponding value for IMCR in some settings. For ESRP, reducing the frequency at which redundant copies are stored visibly reduces the overhead, especially in cases with the failure of multiple nodes. (Tables 6.8 and 6.9). As for matrix `audikw_1`, Table 6.9 shows that overheads for the failure-free cases for ESRP and IMCR are close, with some advantage for ESRP in cases with multiple-node failures.

In the case of ESRP, the ranks of the lost nodes determine the submatrix $\boldsymbol{A}_{I_f,I_f}$ (where f represents the set of indices affected by the node failure), and thus which inner linear system will be solved during the reconstruction in Line 8 of Alg. 2, and how fast this can be done. This cost is also influenced by the performance of the preconditioner used for the inner system. As a consequence, the recovery times for ESRP change for different matrices and for different sets of lost nodes for the same matrix. In contrast, the recovery cost in IMCR is the cost of transferring checkpointed vectors to the replacement nodes, and it is more or less independent of the location of the failure. Currently, our experiments use a simple block Jacobi preconditioner. We believe that ESRP would greatly benefit from more appropriate preconditioners, and an investigation of this aspect is future work.

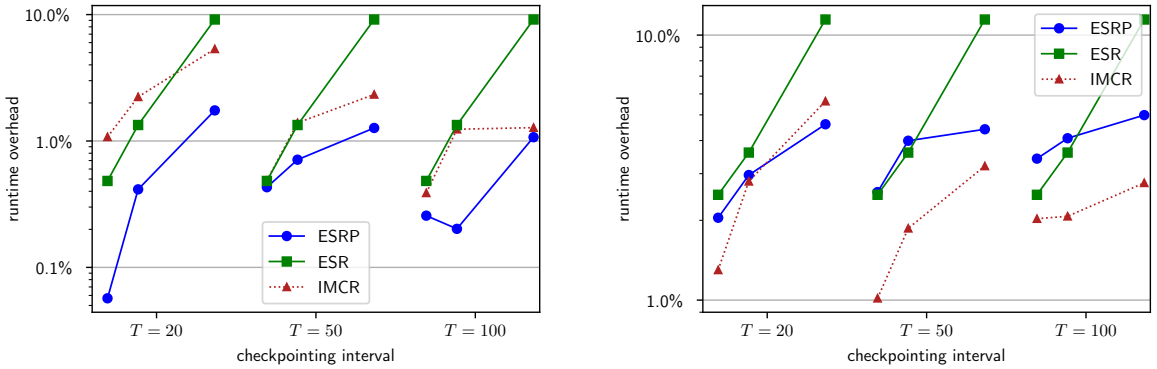
Whether ESRP or IMCR is a better strategy for resilience depends on the probability of node failures happening. If the probability is low, a method with a low overhead in the undisturbed case would be preferable, even if the reconstruction cost is higher, since it is unlikely that the runtime overhead must be incurred. Conversely, a method with a lower recovery cost is preferable if the probability of encountering a node failure is higher.

Table 6.8: Results for matrix `Emilia_923`. Reference time $t_0 = 14.66$ s. The reference case takes $C = 10279$ iterations to reach convergence. The strategies shown are ESR with periodic storage (ESRP) and In-memory buddy CR (IMCR). T: Checkpointing interval, measured in iterations. ϕ: Number of supported node failures. ψ: Number of introduced node failures. All overheads are relative to t_0. *Failure-free overhead*: runtime overhead of runs with resilience, but without introduced node failures. The Location column indicates where the failures are introduced. Rows marked with *start* and *center* have node failures introduced in blocks starting in ranks 0 and 64, respectively. *Overhead with node failures*: overall overheads for runs with an event where as many nodes fail simultaneously as the solver can tolerate. *Reconstruction overhead*: overhead for the reconstruction operations (collecting data in the replacement nodes and reconstructing the state for ESRP, and sending the checkpointed data to the replacement node in IMCR). All results are the median of at least five repeated experiments for the corresponding setting. In all cases, node failures are introduced two iterations before a checkpoint for the interval containing the iteration $C/2$, thus representing a worst-case scenario. The overhead for wasted iterations for $T > 1$ is not explicitly shown. It can be approximated by subtracting the reconstruction overhead from the overall overhead since, in general, the reconstruction does not change the trajectory of the solver after restarting.

		Failure-free overhead [%]				Overhead with node failures [%]			Reconstruction overhead [%]		
Strategy	T	$\phi = 1$	$\phi = 3$	$\phi = 8$	Location	$\phi = \psi = 1$	$\phi = \psi = 3$	$\phi = \psi = 8$	$\phi = \psi = 1$	$\phi = \psi = 3$	$\phi = \psi = 8$
ESRP	1	0.4830	1.3375	9.1421	Start	2.8222	3.6966	11.5376	2.3745	2.0782	3.5541
					Center	2.4125	3.4152	10.6912	1.9389	2.1925	2.8038
	20	0.0569	0.4144	1.7454	Start	2.0299	2.9246	4.6422	2.3871	2.1060	3.5559
					Center	2.0605	3.0033	4.3514	1.1207	2.1979	2.8377
	50	0.4307	0.7111	1.2672	Start	2.6826	4.9636	4.9943	1.5942	2.9031	3.6114
					Center	2.5405	3.6839	3.7626	1.1161	2.1949	2.8485
	100	0.2566	0.2018	1.0712	Start	3.5246	3.9890	5.4503	1.5980	2.9106	3.6123
					Center	3.1925	4.2150	4.1014	1.9495	2.1910	2.8275
IMCR	20	1.0769	2.2326	5.3460	Start	0.9414	2.8330	5.6569	0.0014	0.0012	0.0023
					Center	1.5076	2.3171	5.6024	0.0010	0.0010	0.0016
	50	0.4860	1.3918	2.3332	Start	1.2451	2.0502	3.2020	0.0014	0.0012	0.0023
					Center	1.0133	1.6810	3.2857	0.0010	0.0012	0.0018
	100	0.3881	1.2383	1.2749	Start	2.3488	2.1359	2.1681	0.0014	0.0013	0.0023
					Center	1.7076	1.9167	3.4778	0.0010	0.0011	0.0018

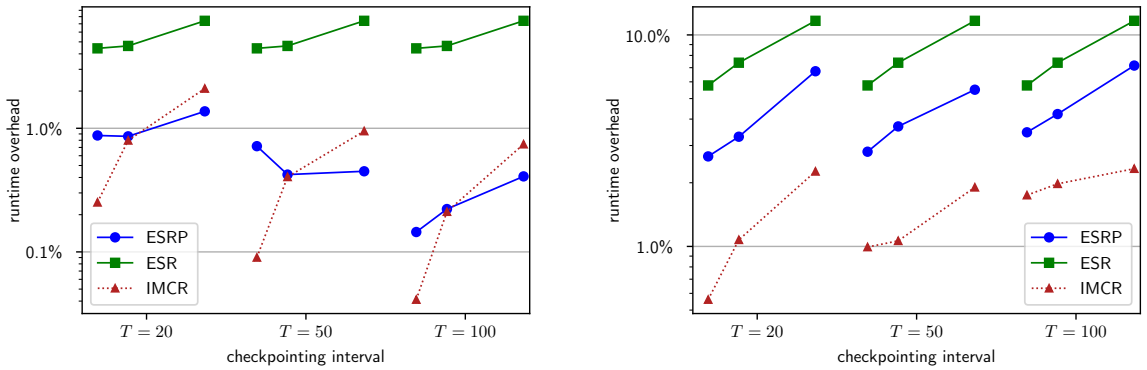
Table 6.9: Results for matrix `audikw_1`. Reference time $t_0 = 23.22$ s. The reference case takes $C = 5543$ iterations to reach convergence. Symbols and terms are explained in the caption of Table 6.8

		Failure-free overhead [%]				Overhead with node failures [%]			Reconstruction overhead [%]		
Strategy	T	$\phi = 1$	$\phi = 3$	$\phi = 8$	Location	$\phi = \psi = 1$	$\phi = \psi = 3$	$\phi = \psi = 8$	$\phi = \psi = 1$	$\phi = \psi = 3$	$\phi = \psi = 8$
ESRP	1	4.4326	4.6371	7.4038	Start	5.5299	8.0317	13.2384	1.2895	2.5640	5.7166
					Center	5.8069	6.1523	10.3589	1.3325	1.5115	2.2443
	20	0.8746	0.8606	1.3713	Start	2.8660	3.5503	7.4569	1.8275	2.5364	5.6828
					Center	2.4897	2.6095	3.6964	1.3433	1.5319	2.2716
	50	0.7166	0.4222	0.4489	Start	3.3570	4.0505	7.1400	1.8198	2.6823	5.7107
					Center	2.4215	2.9246	3.3996	1.3426	1.5288	2.2480
	100	0.1448	0.2221	0.4082	Start	3.3412	4.7635	8.3100	1.3066	2.5310	5.7064
					Center	3.5648	3.3805	4.2566	1.3446	1.5202	2.2511
IMCR	20	0.2515	0.7975	2.0963	Start	0.6075	1.0509	2.2102	0.0008	0.0009	0.0014
					Center	0.5135	1.0979	2.3148	0.0006	0.0008	0.0012
	50	0.0901	0.4041	0.9489	Start	1.0000	1.0403	1.7710	0.0007	0.0009	0.0014
					Center	0.9862	1.9529	1.9225	0.0006	0.0007	0.0012
	100	0.0411	0.2114	0.7430	Start	1.7560	1.9292	2.2807	0.0008	0.0009	0.0014
					Center	1.7383	2.2017	2.5487	0.0006	0.0008	0.0012



(a) Failure-free solver (b) Node failures introduced

Figure 6.11: Median relative runtime overhead for the experiments with the matrix `Emilia_923`. In each plot, experiments for a checkpoint interval T are clustered together. In each cluster, there are three lines, representing experiments with ESRP, ESR and in-memory CR (IMCR). In each line, the three markers, from left to right, represent experiments with 1, 3, or 8 redundant copies, and also 1, 3 or 8 simultaneous node failures for both figures, ESR results are the same for all checkpointing intervals within each plot because they are equivalent to ESRP results with $T = 1$, and are displayed along data for ESRP and IMCR for comparison. The markers represent the median for results in all locations (cf. Tab 6.8) and repetitions.



(a) Failure-free solver (b) Node failures introduced

Figure 6.12: Median relative runtime overhead for ESRP for the experiments with the matrix `audikw_1`. See the caption of Fig. 6.11 for details on the structure of the plot.

Accuracy of the experiments

In general, when working with PCG without residual replacement, there is some drift between the vector $\boldsymbol{r}$ and the vector $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$ [VY00]. In ESRP, solving the inner system with an iterative solver, and performing the reconstruction in floating-point arithmetic, cause the reconstructed vector $\boldsymbol{r}$ not to be exactly equal to its state before the node failure. To evaluate the accuracy of ESRP, we compute the vector $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(End)}$, where the

superscript *End* represents the vectors' state after convergence, and use it to compute the *residual drift* metric:

$$\frac{\|\boldsymbol{r}^{(End)}\|_2 - \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(End)}\|_2}{\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(End)}\|_2}. \tag{6.9}$$

A more positive value of the residual drift indicates a smaller value of $\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(End)}\|_2$ and, thus, a more accurate result. This metric is not used to determine convergence of the solver; for this, we use the relative residual as described in Section 6.3.5. The residual drift is computed only after the solver has converged. We use this metric to ensure that ESRP is not generally less accurate than PCG.

PCG and ESRP experiments without node failures produce the same value for the metric since they always follow exactly the same trajectory. With node failures, the residual drift depends on the selection of affected nodes and on the iteration when the node failure occurs; in this case, we present the minimum and median for all experiments. Accuracy results are summarized in Tab. 6.10. In the median, ESRP with node failures does not differ significantly from PCG. As for the minimum value, the results for the matrix `Emilia_923` show little accuracy loss with respect to PCG, but for the matrix `audikw_1`, there is a drift of close to 15.5%, where PCG has a drift of close to 8%. We do not consider this to be a significant issue. The slight advantage for ESRP in the median case for `audikw_1`is explained by the fact that it reconstructs the iterand from the residual vector, thus making the residual and iterand consistent with each other.

Table 6.10: Residual drift observed in the experiments. *Reference*: Residual drift for all failure-free cases. *Median*: Median residual drift over all experiments with node failures. *Minimum*: Minimum residual drift over all experiments with node failures, representing the greatest loss of accuracy during the reconstruction in ESRP.

Matrix	Reference	Median	Minimum
`Emilia_923`	-4.43×10^{-2}	-4.74×10^{-2}	-5.63×10^{-2}
`audikw_1`	-7.98×10^{-2}	-6.67×10^{-2}	-1.55×10^{-1}

6.4 Summary

This chapter presented our extensions to the ESR method that we introduced in Chapter 5. They are used with the conjugate gradients method as a target algorithm. These extensions are:

1. ESR resilience against the simultaneous failure of multiple nodes. This expands the

scope of the ESR method to a wider set of failure scenarios. Our experiments showed what we consider to be very low overheads for even extreme numbers of nodes failing simultaneously, with the overhead for undisturbed runs with resilience against the simultaneous failure of three nodes ranging between 2.2% and 24.1%.

2. enabling ESR to recover from a node failure without the use of spare nodes by redistributing the ownership of vector entries. Our experiments showed that this approach produces relatively small overheads, with a maximum of around 12%, except for problems that require a very short time to converge to the solution. Care must be taken when working with a preconditioner. For example, the changes in entry ownership among the nodes can cause the blocks of a block Jacobi preconditioner not to be contained in a single node anymore, and lead to some overhead.

3. modifications necessary for the ESR method to store redundant data less frequently, thus reducing the runtime overhead in failure-free scenarios. In our experiments, we compared ESR with increased period between redundant information storage, which we call ESRP, with in-memory checkpoint/restart (IMCR). ESRP produced lower overheads for cases without node failures, but this advantage disappeared when node failures were introduced. The reason is the more expensive reconstruction operation of ESRP. Hence, ESRP would be a better fit in situations with a low likelihood of node failures happening, in which a lower investment for maintaining redundant information for resilience would be desired, even if the cost of the reconstruction is somewhat larger than with alternative methods.

In this chapter, we illustrated our extensions with the conjugate gradients method as the target algorithm. The application to these extensions to other iterative algorithms is straightforward, but the more difficult problem of producing a exact state reconstruction strategy for the new target algorithm remains. In the next chapter, we present our meta-algorithm to produce ESR resilience strategies against node failures for linear algebra algorithms based on finite-term recurrences, which enables us to move beyond the conjugate gradients method that we have worked with so far.

Chapter 7

A generic meta-algorithm for producing ESR methods

In this chapter, we present work published in [PEG20]. We work on resilience for iterative methods in linear algebra that satisfy the conditions that we present later in Section 7.1.3, in a distributed-memory setting. The algorithms we work with involve matrix-vector products, and the matrices and vectors involved are held in the memory of a computer cluster in a block-row distribution.

We concern ourselves with situations where the computer cluster can suffer node failures, i.e., events in which its nodes stop working and the information regarding the iterative algorithm that is contained in them becomes inaccessible, such as blocks of vector entries, matrix rows, and the scalars residing in the nodes' memory.

We look for a generic way to derive exact state reconstruction (ESR) strategies for iterative linear algebra algorithms. These strategies reconstruct the information lost during a node failure in a reconstruction node, and, in exact arithmetic, recover the state of the iterative algorithm exactly. We assume that the static data that defines the problem is securely stored and can be retrieved during the reconstruction.

In this work, we assume the availability of a spare node to act as a replacement node, and that only one node can fail at a time. However, generalizing the methods described in this section to deal with the simultaneous failure of multiple nodes (cf. Section 6.1) and to work without spare nodes (cf. Section 6.2) is possible.

The main contribution of this work is a generic procedure for the derivation of exact state reconstruction strategies for iterative linear algebra methods satisfying the conditions presented in section 7.1.3. Our approach is based on the dependency graph of the iterative algorithm, and moves backwards to recover the lost entries of its vectors. We also propose ways to efficiently solve for variables in frequently occurring algorithmic patterns in these iterative methods.

In the recovery process after a node failure it is often necessary to solve a local linear

system, using a diagonal submatrix of the system matrix, to recover the missing entries of a given vector. Some of the iterative algorithms that we consider in this work can deal with general, non-symmetric matrices, and it is therefore possible that the submatrices required during the reconstruction are rank deficient. Based on a *communication-avoiding rank-revealing QR* (CARRQR) decomposition, we provide a strategy for finding an alternative submatrix that can then be used to find the missing entries.

7.1 Generic derivation of ESR

In this section, we present our generalization of the ESR method, applied to iterative methods where an SpMV operation is used.

7.1.1 Drawing a dependency graph

The value of a variable of an iterative algorithm depends on previous values of other variables. Thus, it is possible to create a dependency graph for the iterative algorithm, extending all the way back to its initialization. Nodes in this graph represent both a variable and the operation used to compute it.

While finding the path backwards in the graph, we can assume that all input vectors in the SpMV are known. If the iterative algorithm operates with a finite-term recurrence, it is only necessary to redundantly store the last few of them. For example, the conjugate gradients method (see Section 7.2.1) is a two-term recurrence, and we must only redundantly store the last two search directions.

Let us consider a graph node m whose corresponding variable is used as an argument in the computation of the variable of node n. In this case, the edge will be directed from m to n. We identify the following annotations for the edge:

- The operation is reversible: If n and all nodes with an edge pointing to n are known, outside of m, it is possible to compute m.

- The operation is not reversible: The node m cannot be computed, even if n and all other nodes pointing to n are known. For example, it is not possible to determine the entries of a vector from its norm.

- A relationship between variables that is not explicit in the graph. This is usually the relationship between the residual and the iterand.

These annotations are displayed in Figure 7.1.

Figure 7.1: Edge annotations for the construction of the dependency graph. (a): Dependency can be reverted. (b): Dependency cannot be reverted. (c): Reversible relation not explicit in the algorithm (between the iterand and the residual, for example).

7.1.2 Derivation of an ESR algorithm

To produce an ESR algorithm that can be used in a reconstruction node, we follow these steps:

1. Identify all relevant variables, considering vectors and matrices as single variables.
2. Identify what variables form the state of the solver. We need enough information to be able to continue with the solution process as an undisturbed solver would.
3. Identify, in each line of the algorithm, which variables can be reconstructed if all other variables were present.
4. Construct the graph from the algorithm. Add relations that are not explicit in the algorithm, such as between the iterand and the residual of a linear solver.
5. From the last vector in the graph that is involved in a SpMV with the system matrix, start traversing the graph in breadth-first order:
   (a) Node n corresponds to an assignment in the algorithm. This assignment defines an equation. For each variable m in the equation (outside of the one corresponding to n), which corresponds to a node with an edge pointing to n, determine if it is possible to find it with the variables known already. See Section 7.3 for some frequently occurring algorithmic patterns in these assignments.
   (b) If the variable can be computed, mark it as known and add it to the list for the graph traversal.
6. Stop once the complete state of the solver can be reconstructed.

At this point, we have a path that enables us to reconstruct the state of the solver, such that it follows the same trajectory as an undisturbed solver. The equations obtained while solving for the variables in this workflow are the individual steps in the ESR algorithm. This reconstruction is performed in a single reconstruction node (since we are assuming that only a single node can fail at a time).

The generic meta-algorithm, used to produce ESR algorithms, is abridged in Alg. 5.

Algorithm 5 State reconstruction meta-algorithm from $\boldsymbol{w}^{(j)}$ as the last input vector to the ASpMV

1: $Q := \{\boldsymbol{w}^{(j)}\}$, Nodes for vectors involved in the ASpMV are marked as known.
2: **repeat**
3: $n := \texttt{pop}(Q)$
4: **for** all nodes m with incoming connections to n **do**
5: **if** m is not marked as known **and**
6: m can be computed from n **then**
7: Mark m as known.
8: Push m into Q.
9: **end if**
10: **end for**
11: **until** All nodes for a complete state are marked as known

7.1.3 Scope of the method

A ESR algorithm can be derived for an iterative algorithm in linear algebra if the latter fulfills the following conditions:

- The iterative algorithm performs a finite-term recurrence. This enables us to reconstruct the state from a bounded amount of data.
- The iterative algorithm involves a matrix-vector product, such that redundant copies the input vector can be produced with low memory and runtime overhead. For this purpose we employ the *augmented sparse matrix-vector product* (ASpMV) discussed in Section 5.1.1.

A complete characterization of the class of iterative algorithms for which the meta-algorithm is applicable is topic of future work.

7.1.4 Performance of the resulting ESR method

Since ASpMV transfers more information than the amount required to compute the matrix-vector product, its use entails an overhead compared to regular SpMV. This overhead depends on various aspects such as the sparsity pattern of the matrix, the topology of the computer cluster, and the selection of which nodes receive and redundantly store a given entry. Sending these additional entries, however, does not require additional MPI messages, or new collective-communication rounds. It can be performed with the MPI operation `Alltoallw`, the same as the regular SpMV. Thus, the significant cost of starting a new communication round is avoided.

In Section 6.3 we show how to apply the ASpMV with a reduced frequency (only every T iterations) and thus reduce the runtime overhead caused by holding redundant copies

to basically zero, at the price of having to revert up to T iterations in the event of a node failure. This is analogous to a checkpoint/restart method with a checkpointing interval T. During failure-free operation, our methods transfer less than the information communicated when using an in-memory checkpoint/restart approach, which transfers the entirety of the local part of the state of the iterative method. However, the state reconstruction of our method, which takes place in the event of a node failure, is more expensive than the recovery stage of checkpoint/restart, which must only retrieve backed-up data. Most of the overhead for the reconstruction in ESR is caused by the solution of a local linear system. We expect to reduce this overhead with the technique presented in Sections 7.3.1 and 7.3.2, which is based on a CARRQR decomposition.

7.2 Case studies

We discuss how to generate ESR strategies for some example target algorithms.

7.2.1 Case study: The conjugate gradients method

In this example, we show how the ESR algorithm for the conjugate gradients method (CG) is produced. For simplicity in the explanations, we only present the non-preconditioned CG method, although this strategy is also applicable to preconditioned CG. CG solves a linear system of the form $\boldsymbol{Ax} = \boldsymbol{b}$, where the matrix $\boldsymbol{A}$ is symmetric, positive definite (SPD).

The CG algorithm is presented in Alg. 6. In this algorithm, we can identify all the vectors ($\boldsymbol{x}$, $\boldsymbol{r}$, $\boldsymbol{p}$ and $\boldsymbol{Ap}$) and scalars ($\boldsymbol{r}^\intercal\boldsymbol{r}$, α and β) that we would declare in the program for the solver. This, of course, is not the only way to select these variables.

We can draw a subgraph for each of the lines of Alg. 6. We present two examples, in Fig. 7.2 and Fig. 7.3 for Lines 3 and 7, respectively, of Alg. 6. It is not necessary to consider all the vectors of the program simultaneously. For example, $\boldsymbol{Ap}$ can be found from $\boldsymbol{p}$, so we do not place the former in our graph. We will not consider the scalars in our graphs either, for reasons explained in Section 7.3.3. However, if we formed the graph with nodes for all variables, we would obtain the same ESR method.

Observing the previously presented conventions, and following the operations specified in the algorithm, a dependency graph for the solver is drawn. Here, it is presented in Fig. 7.4, and the steps required to reconstruct the state are described in the figure's caption. The resulting ESR algorithm for CG is presented in Alg. 7, including the addition of the necessary steps for retrieving the required static data and entries from the surviving nodes. After we have found a way to recover the state, we can determine more carefully how to use the scalars. It turns out that, from the scalars, we only require $\beta^{(j-1)}$ to be able to

Algorithm 6 Conjugate gradients (CG) method [Saa03, Alg. 6.18]

1: $\boldsymbol{x}^{(0)}$ arbitrary, $\boldsymbol{r}^{(0)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(0)}, \boldsymbol{p}^{(0)} := \boldsymbol{r}^{(0)}$
2: **for** $j = 0, 1, \ldots,$ until convergence **do**
3: $\quad \alpha^{(j)} := \boldsymbol{r}^{(j)\intercal}\boldsymbol{r}^{(j)} / \boldsymbol{p}^{(j)\intercal}\boldsymbol{A}\boldsymbol{p}^{(j)}$
4: $\quad \boldsymbol{x}^{(j+1)} := \boldsymbol{x}^{(j)} + \alpha^{(j)}\boldsymbol{p}^{(j)}$
5: $\quad \boldsymbol{r}^{(j+1)} := \boldsymbol{r}^{(j)} - \alpha^{(j)}\boldsymbol{A}\boldsymbol{p}^{(j)}$
6: $\quad \beta^{(j)} := \boldsymbol{r}^{(j+1)\intercal}\boldsymbol{r}^{(j+1)} / \boldsymbol{r}^{(j)\intercal}\boldsymbol{r}^{(j)}$
7: $\quad \boldsymbol{p}^{(j+1)} := \boldsymbol{r}^{(j+1)} + \beta^{(j)}\boldsymbol{p}^{(j)}$
8: **end for**

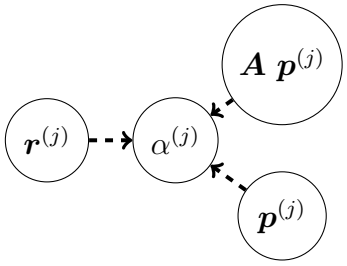


Figure 7.2: Subgraph for the formula $\alpha^{(j)} := \boldsymbol{r}^{(j)\intercal}\boldsymbol{r}^{(j)} / \boldsymbol{p}^{(j)\intercal}\boldsymbol{A}\boldsymbol{p}^{(j)}$. Arrows go towards the $\alpha^{(j)}$ node because that is the variable being computed. Dashed lines are used to indicate what variables cannot be computed if all of the others are known.

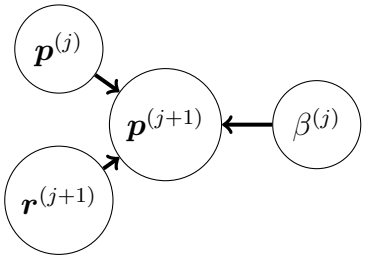


Figure 7.3: Subgraph for the formula $\boldsymbol{p}^{(j+1)} := \boldsymbol{r}^{(j+1)} + \beta^{(j)}\boldsymbol{p}^{(j)}$. Arrows go towards the $\boldsymbol{p}^{(j+1)}$ node because that is the variable being computed. Bold lines are used to indicate what variables can be computed if all of the others are known.

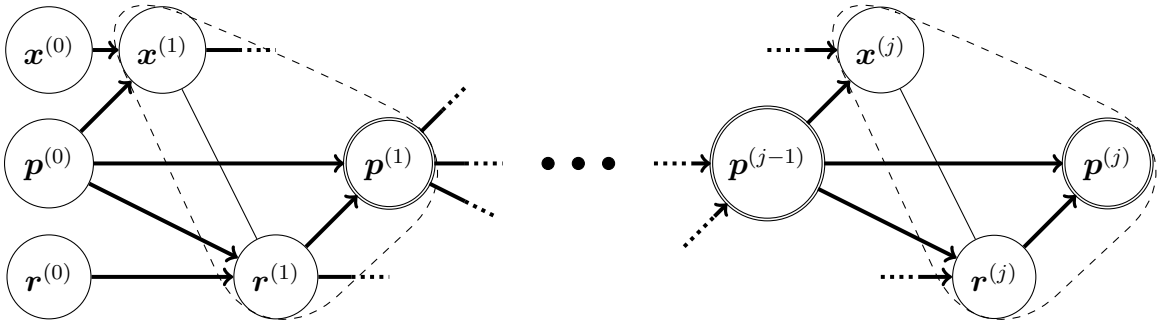


Figure 7.4: Dependency graph for the CG method. 1. The traversal starts in the last vector involved in the SpMV. Here, it is $\boldsymbol{p}^{(j)}$. 2. Starting the breadth-first traversal with $\boldsymbol{r}^{(j)}$, we check if $\boldsymbol{r}^{(j)}$ can be computed. Since we assume that $\boldsymbol{p}^{(j-1)}$ is also known, $\boldsymbol{r}^{(j)}$ can be computed. There are no further unknown nodes connected to $\boldsymbol{p}^{(j)}$. 3. The traversal continues from $\boldsymbol{r}^{(j)}$. From this node, both $\boldsymbol{x}^{(j)}$, through the residual relation, and $\boldsymbol{r}^{(j-1)}$ can be found. At this point, we have the complete state of the solver, surrounded by a dashed line: If we know $\boldsymbol{x}^{(j)}$, $\boldsymbol{r}^{(j)}$ and $\boldsymbol{p}^{(j)}$, the solver can continue in the same trajectory as before the node failure. $\boldsymbol{p}^{(j-1)}$ is still required to complete the reconstruction.

reconstruct $\boldsymbol{r}^{(j)}$ from the equation of Line 7 in Alg. 6.

Algorithm 7 ESR reconstruction phase for the CG method on a replacement node.

1: Retrieve the static data $\boldsymbol{A}_{I_f,I}$, and $\boldsymbol{b}_f$
2: Gather $\boldsymbol{r}^{(j)}_{I\setminus I_f}$ and $\boldsymbol{x}^{(j)}_{I\setminus I_f}$
3: Retrieve the redundant copies of $\beta^{(j-1)}$, $\boldsymbol{p}_f^{(j-1)}$, and $\boldsymbol{p}_f^{(j)}$
4: Compute $\boldsymbol{r}_f^{(j)} := \boldsymbol{p}_f^{(j)} - \beta^{(j-1)}\boldsymbol{p}_f^{(j-1)}$
5: Compute $\boldsymbol{w} := \boldsymbol{b}_f - \boldsymbol{r}_f^{(j)} - \boldsymbol{A}_{I_f,I\setminus I_f}\boldsymbol{x}^{(j)}_{I\setminus I_f}$
6: Solve $\boldsymbol{A}_{I_f,I_f}\boldsymbol{x}_f^{(j)} = \boldsymbol{w}$ for $\boldsymbol{x}_f^{(j)}$

7.2.2 Case study: The BiCGStab method

We generate an ESR reconstruction algorithm for the BiCGStab solver. This is a Krylov subspace method to solve the linear system, $\boldsymbol{Ax} = \boldsymbol{b}$, where, in contrast to CG, there are no restrictions on $\boldsymbol{A}$ beyond not being singular. BiCGStab is shown in Alg. 8.

From the listing we can produce a dependency graph, presented in Fig. 7.5. The steps followed to reconstruct the state are presented in the caption of this figure. Notice that the solver does not require the vector $\boldsymbol{s}^{(j)}$ to continue in the same trajectory as an undisturbed solver. In the graph of Fig. 7.5, it is sufficient to reconstruct $\boldsymbol{p}^{(j+1)}$, $\boldsymbol{p}^{(j)}$, $\boldsymbol{r}^{(j+1)}$ and $\boldsymbol{x}^{(j+1)}$ to compute a new $\boldsymbol{s}^{(j+1)}$. Because BiCGStab operates with general matrix, a diagonal submatrix, as the one we use for the case of CG, could be singular and not usable to

Algorithm 8 BiCGStab algorithm [Saa03, Alg. 7.7]

1: $\boldsymbol{x}^{(0)}$ arbitrary, $\boldsymbol{r}^{(0)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(0)}$, $\boldsymbol{r}^{(0)*}$ arbitrary, $\boldsymbol{p}^{(0)} := \boldsymbol{r}^{(0)}$
2: **for** $j = 0, 1, \ldots,$ until convergence **do**
3: $\quad \alpha^{(j)} := \boldsymbol{r}^{(j)\intercal}\boldsymbol{r}^{(0)*} / \boldsymbol{r}^{(0)*\intercal}\boldsymbol{A}\boldsymbol{p}^{(j)}$
4: $\quad \boldsymbol{s}^{(j)} := \boldsymbol{r}^{(j)} - \alpha^{(j)}\boldsymbol{A}\boldsymbol{p}^{(j)}$
5: $\quad \omega^{(j)} := \boldsymbol{s}^{(j)\intercal}\boldsymbol{A}\boldsymbol{s}^{(j)} / (\boldsymbol{A}\boldsymbol{s}^{(j)})^{\intercal}(\boldsymbol{A}\boldsymbol{s}^{(j)})$
6: $\quad \boldsymbol{x}^{(j+1)} := \boldsymbol{x}^{(j)} + \alpha^{(j)}\boldsymbol{p}^{(j)} + \omega^{(j)}\boldsymbol{s}^{(j)}$
7: $\quad \boldsymbol{r}^{(j+1)} := \boldsymbol{s}^{(j)} - \omega^{(j)}\boldsymbol{A}\boldsymbol{s}^{(j)}$
8: $\quad \beta^{(j)} := \frac{\alpha^{(j)}}{\omega^{(j)}}\, \boldsymbol{r}^{(j+1)\intercal}\boldsymbol{r}^{(0)*} / \boldsymbol{r}^{(j)\intercal}\boldsymbol{r}^{(0)*}$
9: $\quad \boldsymbol{p}^{(j+1)} := \boldsymbol{r}^{(j+1)} + \beta^{(j)}(\boldsymbol{p}^{(j)} - \omega^{(j)}\boldsymbol{A}\boldsymbol{p}^{(j)})$
10: **end for**

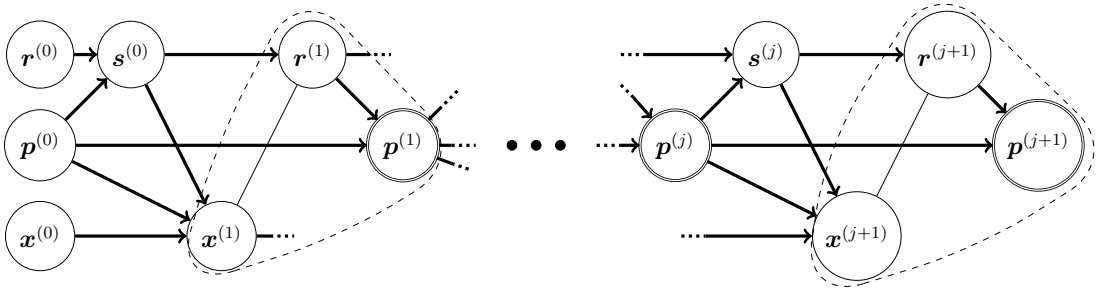


Figure 7.5: Dependency graph for the BiCGStab method. 1. The traversal starts in the last vector involved in the SpMV: $\boldsymbol{p}^{(j+1)}$. 2. Starting the breadth-first traversal, we check if $\boldsymbol{r}^{(j+1)}$ can be computed. Since $\boldsymbol{p}^{(j)}$ is assumed to be known, this is the case. 3. Continuing the traversal, we find that $\boldsymbol{s}^{(j)}$ and $\boldsymbol{x}^{(j+1)}$ can be computed, the latter by using the residual relation. At this point, the state has been reconstructed and the solver can continue in the same trajectory as the undisturbed solver.

perform the reconstruction. Instead, we find an index set I_r such that the submatrix $\boldsymbol{A}_{I_r,I_f}$ is well conditioned and can be used for reconstruction of the state. We discuss how to select I_r in Section 7.3.1.

From this, we can write down the ESR reconstruction algorithm for BiCGStab, presented in Alg. 9.

Algorithm 9 ESR reconstruction phase for the BiCGStab method on a replacement node.

1: Retrieve the static data $\boldsymbol{A}_{I_f,I}$, $\boldsymbol{r}_f^{(0)*}$ and $\boldsymbol{b}_f$
2: Gather $\boldsymbol{r}_{I\setminus I_f}^{(j)}$, $\boldsymbol{x}_{I\setminus I_f}^{(j)}$ and $\boldsymbol{p}_{I\setminus I_f}^{(j-1)}$
3: Retrieve the redundant copies of $\beta^{(j-1)}$, $\omega^{(j-1)}$, $\boldsymbol{p}_f^{(j-1)}$, and $\boldsymbol{p}_f^{(j)}$
4: Compute $\boldsymbol{\varrho}_{I_f}^{(j-1)} := \boldsymbol{A}_{I_f,I}\boldsymbol{p}^{(j-1)}$
5: Compute $\boldsymbol{r}_f^{(j)} := \boldsymbol{p}_f^{(j)} - \beta^{(j-1)}(\boldsymbol{p}_f^{(j-1)} - \omega^{(j-1)}\boldsymbol{\varrho}_f^{(j-1)})$
6: Find I_r. See Section 7.3.1.
7: Collect rows of $\boldsymbol{A}_{I_r,I}$, and entries of $\boldsymbol{b}_{I_r}$ and $\boldsymbol{r}_{I_r}^{(j)}$ in the replacement node.
8: Compute $\boldsymbol{w} := \boldsymbol{b}_{I_r} - \boldsymbol{r}_{I_r}^{(j)} - \boldsymbol{A}_{I_r,I\setminus I_f}\boldsymbol{x}_{I\setminus I_f}^{(j)}$
9: Solve $\boldsymbol{A}_{I_r,I_f}\boldsymbol{x}_f^{(j)} = \boldsymbol{w}$ for $\boldsymbol{x}_f^{(j)}$

7.2.3 Case study: The Lanczos algorithm

The Lanczos algorithm is used to find the k most dominant eigenvalues and eigenvectors of a symmetric matrix $\boldsymbol{A} \in \mathbb{R}^{|I|\times|I|}$. The algorithm finds a sequence of scalars that form a symmetric tridiagonal matrix $\boldsymbol{T}_k \in \mathbb{R}^{k\times k}$:

$$\boldsymbol{T}_k = \begin{pmatrix} \alpha^{(1)} & \beta^{(2)} & & & \\ \beta^{(2)} & \alpha^{(2)} & \beta^{(3)} & & \\ & & \ddots & & \\ & & \beta^{(k-1)} & \alpha^{(k-1)} & \beta^{(k)} \\ & & & \beta^{(k)} & \alpha^{(k)} \end{pmatrix}, \tag{7.1}$$

as well as the column vectors for an orthogonal matrix $\boldsymbol{V}_k$, such that $\boldsymbol{T}_k = \boldsymbol{V}_k^\mathsf{T}\boldsymbol{A}\boldsymbol{V}_k$. The eigenvalues of $\boldsymbol{T}_k$ approximate the dominating eigenvalues of $\boldsymbol{A}$, and if $\boldsymbol{y}$ is an eigenvector of $\boldsymbol{T}_k$, then $\boldsymbol{V}_k\boldsymbol{y}$ approximates an eigenvector of $\boldsymbol{A}$.

The Lanczos algorithm is presented in Alg. 10. From it, we build the dependency graph, ignoring the scalars and intermediate vectors ($\boldsymbol{A}\boldsymbol{v}^{(j)}$). The graph is presented in Fig. 7.6, and the steps followed to perform the reconstruction are presented in its caption.

Algorithm 10 Lanczos algorithm [Saa03, Alg. 6.15]

$\boldsymbol{v}^{(1)}$ arbitrary, with $\|\boldsymbol{v}^{(1)}\|_2 = 1$, $\beta^{(1)} := 0$, $\boldsymbol{v}^{(0)} = \boldsymbol{0}$
for $j = 1, 2, \ldots, k$ **do**
    $\boldsymbol{w}^{(j)} := \boldsymbol{A}\boldsymbol{v}^{(j)} - \beta^{(j)}\boldsymbol{v}^{(j-1)}$
    $\alpha^{(j)} := \boldsymbol{w}^{(j)\intercal}\boldsymbol{v}^{(j)}$
    $\boldsymbol{w}^{(j)} := \boldsymbol{w}^{(j)} - \alpha^{(j)}\boldsymbol{v}^{(j)}$
    $\beta^{(j+1)} := \|\boldsymbol{w}^{(j)}\|_2$. If $\beta^{(j+1)} = 0$, stop.
    $\boldsymbol{v}^{(j+1)} := \boldsymbol{w}^{(j)}/\beta^{(j+1)}$
end for

Figure 7.6: Dependency graph for the Lanczos method. 1. The traversal starts in the last vector involved in the SpMV. Here, it is $\boldsymbol{v}^{(j)}$. 2. Starting the breadth-first traversal with $\boldsymbol{w}^{(j-1)}$, Since we assume that $\boldsymbol{v}^{(j)}$ is known, $\boldsymbol{w}^{(j-1)}$ can be computed. If we also redundantly store $\boldsymbol{v}^{(j-1)}$, we have the complete state of the solver and can continue iterating in the same trajectory as the unaffected solver.

The scalars α and β must be replicated in all nodes in order to perform the operations in Lines 3, 5 and 7 of Alg. 10. We focus on the case where $k \ll |I|$, where $|I|$ is the number of entries of the vectors and therefore also the size of the problem. Consequently, we can assume that the number of scalars remains small, and it is reasonable to maintain copies of all of them in every node.

Algorithm 11 ESR reconstruction phase for the Lanczos algorithm on a replacement node.

1: Retrieve the static data: $\boldsymbol{A}_{I_f,I}$
2: Gather $\boldsymbol{w}_{I\setminus I_f}^{(j-1)}$
3: Retrieve the redundant copies of $\beta^{(j)}$, $\boldsymbol{v}_f^{(j-1)}$, and $\boldsymbol{v}_f^{(j)}$

The resulting ESR algorithm is presented in Alg. 11. After a node failure, the algorithm can recover the last $\boldsymbol{w}$ vector from the last $\boldsymbol{v}$ vector. It can then continue computing the remaining values of $\boldsymbol{T}_k$ if we retrieve the $\boldsymbol{v}$ vectors from the last two iterations. In the end, the matrix $\boldsymbol{T}_k$ is found and it can be used to approximate the eigenvalues of $\boldsymbol{A}$.

Although the Lanczos algorithm is a two-step recurrence, all of the $\boldsymbol{v}$ vectors are required for eigenvalue computations. After a node failure, the rows I_f of the matrix $\boldsymbol{V}_k$ are lost. Therefore, recovering from such a failure in a way that enables eigenvalue computations also requires provision of redundancy for all $\boldsymbol{v}$ vectors, and not only the last two. This problem is beyond the scope of our work.

7.3 Handling algorithmic patterns during derivation of an ESR strategy

When building an ESR algorithm for an iterative algorithm, certain operations are common. In this section, we explain how to deal with them.

7.3.1 Reconstructing vectors involved in matrix-vector products

Some formulas in an algorithm have the form $\boldsymbol{z} := \boldsymbol{w} + \boldsymbol{A}\boldsymbol{x}$, where $\boldsymbol{z}$ and $\boldsymbol{w}$ are known vectors, $\boldsymbol{A}$ is a matrix, and we try to reconstruct the vector $\boldsymbol{x}$. This is equivalent to solving the linear system $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ where $\boldsymbol{y} := \boldsymbol{z} - \boldsymbol{w}$. As an instance of this case, in the ESR algorithm for CG, we use the residual relation $\boldsymbol{r}^{(j)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(j)}$ for the reconstruction of the iterand (Alg. 7, Line 6). After a node failure, the surviving parts of $\boldsymbol{x}$ will still be available, so it is not necessary to solve the full linear system. Instead, we can solve only for the lost entries. If I represents the ordered set of all indices of the system, and I_F is the ordered set of the indices of all entries lost after a node failure event, we can introduce

an ordered set I_r, with $|I_r| = |I_F|$, and then we write this matrix-vector product as

$$\begin{pmatrix} \boldsymbol{y}_{I\setminus I_r} \\ \boldsymbol{y}_{I_r} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}_{I\setminus I_r, I\setminus I_F} & \boldsymbol{A}_{I\setminus I_r, I_F} \\ \boldsymbol{A}_{I_r, I\setminus I_F} & \boldsymbol{A}_{I_r, I_F} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_{I\setminus I_F} \\ \boldsymbol{x}_{I_F} \end{pmatrix}.$$

Thus, we can write an equation for the lost entries of $\boldsymbol{x}$:

$$\boldsymbol{A}_{I_r, I_F} \boldsymbol{x}_{I_F} = \boldsymbol{y}_{I_r} - \boldsymbol{A}_{I_r, I\setminus I_F} \boldsymbol{x}_{I\setminus I_F}. \tag{7.2}$$

Selection of I_r

For a SPD matrix $\boldsymbol{A}$, we can pick $I_r = I_F$ because the submatrix $\boldsymbol{A}_{I_F, I_F}$ is non-singular (cf. Appendix A, Corollary 1). This has the advantage of being local: It does not require matrix information from other nodes to define the submatrix.

For more general matrices, it might happen that $\boldsymbol{A}_{I_F, I_F}$ is non-invertible and the linear system of Eq. (7.2) cannot be solved. In this event, we need to pick I_r such that the submatrix is non-singular. According to Corollary 2 in Appendix A, enough information exists in the matrix to solve the system of Eq. (7.2). If $I_r \neq I_f$, this requires row information from other nodes to form the subsystem, and thus requires communication.

The selection of I_r intends to form a matrix of the *most linearly independent* row vectors of $\boldsymbol{A}_{I, I_f}$, that is, a selection that minimizes the condition number of the resulting matrix $\boldsymbol{A}_{I_r, I_f}$. We propose to perform this selection using the *communication-avoiding rank-revealing QR factorization* (CARRQR) [Dem+15]. CARRQR minimizes communication by employing a technique called tournament pivoting. In order to identify the $|I_f|$ most linearly independent rows of $\boldsymbol{A}_{I, I_f}$, the algorithm partitions $\boldsymbol{A}^T_{I, I_f}$ into blocks of $2|I_f|$ columns. Tournament pivoting then proceeds in two steps. The first step applies a rank revealing QR factorization, e.g., QR factorization with column pivoting (QRCP), to each block to reveal and permute the $|I_f|$ most linearly independent columns to the left. The second step involves a reduction operation that combines these $|I_f|$ leftmost columns with the $|I_f|$ leftmost columns of its neighbor to form new blocks of $2|I_f|$ columns. These steps are repeated until, after a final rank revealing QR factorization, there are only $|I_f|$ columns left. The algorithm produces a permutation matrix $\boldsymbol{P}_r$ such that the most linearly independent rows are located in the upper square submatrix of $\boldsymbol{P}_r^T \boldsymbol{A}_{I, I_f}$.

In a distributed memory environment, traditional rank revealing algorithms like, e.g., QRCP are sub-optimal with respect to the number of messages exchanged. CARRQR minimizes communication at the cost of worse bounds for a quantity involved in the characterization of a rank revealing factorization. In [Dem+15], Demmel et al. show that these bounds are usually very pessimistic, and that the algorithm is competitive in practice. CARRQR performs three times more flops than QRCP but, on a system where communication is the bottleneck, the former is expected to be significantly faster. We consider

CARRQR as a first approximation to the solution of this problem, and illustrate its use to improve the condition number of a matrix subblock with experiments in Section 7.3.2.

Completing the linear system with rows from other nodes

It is also possible to first perform a singular value decomposition (SVD) in the replacement node, and then obtain a minimal number of rows from other nodes. Also, because of Corollary 2, there must exist a subset of indices I_c, with $I_c \subset I$ and $I_c \cap I_F = \emptyset$, that provides the additional information to solve the system. Of course, we want our selection of I_c to minimize some objective, such as the amount of communication required, which depends on the nodes that own the selected rows.

A required step would be to determine if the submatrix $\boldsymbol{A}_{I_F,I_F}$ is singular. Thus, we perform a rank-revealing QR factorization. If the subsystem has full rank, we can proceed with the solution. Otherwise, the system will have a rank $k < |I_F|$. We perform a SVD decomposition:

$$\boldsymbol{A}_{I_F,I_F} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\intercal = \hat{\boldsymbol{U}}\hat{\boldsymbol{\Sigma}}\hat{\boldsymbol{V}}^\intercal, \tag{7.3}$$

where $\hat{\boldsymbol{\Sigma}} \in \mathbb{R}^{k\times k}$ is the diagonal matrix of the non-zero singular values of $\boldsymbol{\Sigma}$ and $\hat{\boldsymbol{U}}, \hat{\boldsymbol{V}} \in \mathbb{R}^{|I_F|\times k}$ are the matrices of the corresponding left and right singular vectors.

We can define another submatrix for the index set $I_G \coloneqq I \setminus I_c \setminus I_f$ and decompose the corresponding linear system as in Eq. (7.4):

$$\begin{pmatrix} \boldsymbol{y}_{I_G} \\ \boldsymbol{y}_{I_c} \\ \boldsymbol{y}_{I_F} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}_{I_G,I_G} & \boldsymbol{A}_{I_G,I_c} & \boldsymbol{A}_{I_G,I_F} \\ \boldsymbol{A}_{I_c,I_G} & \boldsymbol{A}_{I_c,I_c} & \boldsymbol{A}_{I_c,I_F} \\ \boldsymbol{A}_{I_F,I_G} & \boldsymbol{A}_{I_F,I_c} & \boldsymbol{A}_{I_F,I_F} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_{I_G} \\ \boldsymbol{x}_{I_c} \\ \boldsymbol{x}_{I_F} \end{pmatrix}. \tag{7.4}$$

From this, we can write one more equation:

$$\boldsymbol{A}_{I_c,I_F}\boldsymbol{x}_{I_F} = \boldsymbol{y}_{I_c} - \boldsymbol{A}_{I_c,I_G}\boldsymbol{x}_{I_G} - \boldsymbol{A}_{I_c,I_c}\boldsymbol{x}_{I_c} \tag{7.5}$$

Eq. (7.5) provides $|I_c|$ additional restrictions on $\boldsymbol{x}_{I_c}$ that can be used to correct the rank deficiency of $\boldsymbol{A}_{I_F,I_F}$.

We define a vector $\boldsymbol{w}_{I_F} \coloneqq \boldsymbol{y}_{I_F} - \boldsymbol{A}_{I_F,I\setminus I_F}\boldsymbol{x}_{I\setminus I_F}$, set $I_r \coloneqq I_F$ and write, from Eq. (7.2), $\boldsymbol{A}_{I_F,I_F}\boldsymbol{x}_{I_F} = \boldsymbol{w}_{I_F}$. Then we can write

$$\hat{\boldsymbol{U}}\hat{\boldsymbol{\Sigma}}\hat{\boldsymbol{V}}^\intercal\boldsymbol{x}_{I_F} = \boldsymbol{w}_{I_F} \rightarrow \hat{\boldsymbol{\Sigma}}\hat{\boldsymbol{V}}^\intercal\boldsymbol{x}_{I_F} = \hat{\boldsymbol{U}}^\intercal\boldsymbol{w}_{I_F}.$$

We introduce the vector $\boldsymbol{v}_{I_c} \coloneqq \boldsymbol{y}_{I_c} - \boldsymbol{A}_{I_c,I_G}\boldsymbol{x}_{I_G} - \boldsymbol{A}_{I_c,I_c}\boldsymbol{x}_{I_c}$ and rewrite Eq. (7.5) as $\boldsymbol{A}_{I_c,I_F}\boldsymbol{x}_{I_F} = \boldsymbol{v}_{I_c}$. Then, provided that the rows of $\boldsymbol{A}_{I_c,I_F}$ are not in the span of the vectors

Table 7.1: Test matrices from [DH11]

Matrix	Problem type	Problem size
sherman5	CFD	3312
west1505	Chemical simulation	1505

Table 7.2: Condition number for submatrices

Matrix	$\mathbf{cond}(\boldsymbol{A}_{I_f,I_f})$	$\mathbf{cond}(\boldsymbol{A}_{I_r,I_f})$
sherman5	4540.0	469.2
west1505	∞	6.08×10^8

of $\hat{\boldsymbol{V}}$, we can instead solve the system

$$\begin{pmatrix} \hat{\boldsymbol{U}}^\intercal \boldsymbol{w}_{I_F} \\ \boldsymbol{v}_{I_c} \end{pmatrix} = \begin{pmatrix} \hat{\boldsymbol{\Sigma}}\hat{\boldsymbol{V}}^\intercal \\ \boldsymbol{A}_{I_c,I_F} \end{pmatrix} \boldsymbol{x}_{I_F} \tag{7.6}$$

and obtain the lost entries we are looking for.

The selection of columns from the surviving nodes, I_c, remains future work.

7.3.2 Experiments with I_r selection using tournament pivoting

We perform numerical experiments with small matrices to demonstrate how an appropriate selection of matrix rows, I_r, improves the conditioning of the subsystem if it is poorly conditioned, or creates a non-singular one if the corresponding submatrix is rank-deficient. As mentioned in Section 7.3.1, we employ CARRQR [Dem+15] for the selection of rows.

We work with two square matrices from the SuiteSparse Matrix Collection [DH11], presented in Table 7.1. We simulate eight nodes by uniformly dividing the rows of the matrix in eight contiguous blocks. We then examine the resulting diagonal submatrices. For the matrix `sherman5`, we pick I_f to correspond to the most poorly conditioned diagonal submatrix. For the matrix `west1505`, we pick I_f to correspond to the second submatrix, which does not have non-zero entries and is thus singular.

The resulting condition numbers are shown in Table 7.2.

For the matrix `sherman5`, the reduced condition number leads to fewer iterations if we solve the subsystem with an iterative solver. For the matrix `west1505`, the subsystem is no longer singular. We can find a solution and, therefore, perform the reconstruction.

7.3.3 Storage and recovery of scalars

Holding scalars in memory does not represent a large overhead in comparison to holding vectors, therefore, we propose to replicate them in all nodes and to maintain the necessary copies in all nodes. Often, these scalars are computed with `AllReduce` operations (as in Line 3 in Alg. 6), making these values available in all surviving nodes in the event of a node failure.

7.3.4 Recomputation of matrix-vector products

If the reconstruction of a vector involves a matrix-vector product (as in Line 4 of Alg. 9) it is possible to recompute only the entries in I_f in the reconstruction node $\boldsymbol{A}_{I_f,I}\boldsymbol{x}$ instead of the full product $\boldsymbol{Ax}$. The latter would have to be performed by all nodes, while the former would result in a smaller overhead during the reconstruction.

7.4 Summary

In this chapter, we presented our meta-algorithm to produce ESR resilience strategies for certain linear algebra algorithms. The meta-algorithm traverses the dependency graph of the target algorithm following a set of rules, and the path it follows represents the exact state reconstruction strategy. We used the meta-algorithm on the conjugate gradients method, BiCGStab and the Lanczos method as target algorithms, producing an ESR algorithm for each.

We need to solve a linear system for a submatrix of the system matrix during the reconstruction. So far, working with the preconditioned conjugate gradients method, we have used a diagonal submatrix, for which we can guarantee positive-definiteness and non-singularity. In the case of the BiCGStab method, the system matrix must not be positive definite and, as a consequence, a diagonal submatrix may be singular. We proposed a strategy to find a well-conditioned matrix to use during the reconstruction, based on a tournament-pivoting, rank-revealing QR decomposition of the rows belonging to the nodes lost in the node failure.

In the next and final chapter of this thesis, we discuss open questions and conclude this work.

Chapter 8

Conclusions

This thesis presents our work extending a class of methods, first proposed by Chen [Che11], to endow iterative algorithms in linear algebra with resilience against node failures. We call these methods *exact state reconstruction* (ESR), because, after a node failure, they attempt to return all relevant variables of the algorithm to the state they were in in the unaffected case. The strategy consists of two procedures: storing enough redundant information to reconstruct the state during the normal operation, and reconstructing the state of the iterative method, allowing it to converge to the correct solution, in the event that a node failure takes place. In the absence of node failures, these methods incur a small runtime overhead, even if the iterative algorithm is protected against the simultaneous failure of a large number of nodes.

In most of our work, we use the preconditioned conjugate gradients method (PCG) as our study case. We present an argument that favors the use of an exact state reconstruction method over naively restarting the PCG solver in the event of a node failure. The argument is that, since a Krylov solver uses the history of the solver to compute the vectors of the next iterations, its state, which contains information of its history, and which completely determines its behavior in future iterations, should be brought back as closely as possible to its pre-failure state. We also support this argument with experiments.

We present four extensions to this method, which we discuss next.

Resilience against multiple node failures In the first place, we describe and implement a strategy to make the original ESR method by Chen resilient against multiple node failures. The algorithms are implemented using PETSc, and experiments run on 128 nodes of the VSC3 machine of the Vienna Scientific Cluster, using eight large, positive-definite matrices from the SuiteSparse collection. Failure-free overheads are relatively small, with a maximum of 8% for resilience against a single node failure, 24.1% for resilience to three node failures, and 91.3% for resilience against eight simultaneous node failures. Although in the worst case, the overhead for this last scenario is large, keep in mind that the event

of the failure failure of eight nodes is also quite unlikely, and even in this scenario, the overhead for most of our test cases is much lower, hovering around 20%. Further work in this thesis shows a strategy to reduce this overhead.

Recovery without the use of spare nodes In second place, we present redistribution strategies to enable the reconstruction of the state without the use of spare nodes. The information lost to the node failure is reconstructed in a set of reconstructed nodes, and after that, the load is rebalanced among the nodes of the cluster. We run experiments on 32 nodes of the Hydra cluster of the TU Wien, this time using our own home brew code instead of PETSc to have more flexibility. Recovering after a node failure in this scenario causes overhead because of the reconstruction of the information as well as because of the reduced number of nodes working on the problem. However, a naive reconstruction of the preconditioner can also cause very significant overheads if it does not adapt well to the data redistribution. In our experiments, the relative overhead of the solver after a node failure for the case without spare nodes over the case with spare nodes is around 5%, but if we use a block Jacobi preconditioner, which can lose its block structure after reordering of the matrix, this overhead shoots up to 60% in some cases. Thus, we conclude that it is necessary to consider the preconditioner to obtain a strategy that does not incur large overheads. This could mean generating a new preconditioner efficiently, or modifying the previous preconditioner after a node failure occurs.

This work does not present strategies on the selection of an adequate preconditioner, or its adaption after the data redistribution caused by a node failure. This aspect remains an open question.

Reduction of the frequency for redundant data storage The third extension we introduce is the reduction of the frequency at which redundant data is stored. Since the more expensive operation of performing an augmented matrix-vector product, which stores redundant information for a potentially necessary reconstruction, is performed less frequently, the runtime overhead for the regular, failure-free operation of the resilient solver decreases. This approach, which we call ESRP, has a visibly smaller overhead when compared to ESR, and, in failure-free scenarios, also against an in-memory checkpoint/restart strategy that we implement for comparison. This indicates that ESRP is a good option if the likelihood of a node failure occurring during the lifetime of the program is low. However, when a node failure occurs, the recovery time is dominated by the solution of a local linear system for the lost nodes, which depends on the matrix itself. An open question is the behavior of these methods in the regime of extreme-scale problems, where the cost of communication becomes dominating, and thus could favor the smaller messages of ESRP when compared to in-memory checkpoint/restart.

Generic meta-algorithm for the generation of ESR strategies Finally, we presented a meta-algorithm to guide the production of new ESR strategies for linear algebra algorithms that use a finite-step recurrence. This meta-algorithms moves backwards in the dependency graph of the variables of the target algorithm, finding a path that enables reconstruction of the state from redundantly stored data.

We demonstrate how this meta-algorithm can be applied to two linear solvers, non-preconditioned conjugate gradients and BiCGStab, and an eigensolver, the Lanczos algorithm. The latter algorithm requires its complete history to find the eigenvectors of the problem but, as we apply it, our ESR strategies are intended to produce only enough information to allow the algorithm to continue iterating as an unaffected solver if a node failure take place, and because of this, we only find the eigenvalues, and not the eigenvectors.

Additionally, we propose guidelines on how to deal with commonly-occurring algorithmic patterns and, particularly, we present a strategy to obtain a well-conditioned linear subsystem, based on a communication avoiding, rank revealing QR decomposition, to recover variables in a reconstruction node.

The extensions developed in this work make resilience strategies based on the exact reconstruction of the state of the algorithm more flexible, and ease their application to a wider range of scientific applications.

As possible future research directions, we propose the implementation and benchmarking of these resilience strategies in a framework supporting real node failures, as well as the study of the influence of the sparsity pattern of the matrix and of the selection of buddy nodes on the overhead of maintaining resilience and of the reconstruction. An investigation of strategies to efficiently produce a new preconditioner when working without spare nodes, as discussed in Section 6.2, would also be valuable.

In this work, we employ redundancy of information to eventually perform reconstruction of the variables of an algorithm. Another potential research direction would be determining whether geometric properties, such as the conjugacy of the search directions in the conjugate gradients method, can be used to find the original values of the variables before the node failure took place without the need to store redundant information.

Bibliography

[Abh+18] Shrirang Abhyankar, Jed Brown, Emil M. Constantinescu, Debojyoti Ghosh, Barry F. Smith, and Hong Zhang. *PETSc/TS: A Modern Scalable ODE/DAE Solver Library*. 2018. arXiv: 1806.01437v1 [math.NA].

[Agu+13] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Jean Roman, and Mawussi Zounon. "Towards resilient parallel linear Krylov solvers: recover-restart strategies". PhD thesis. INRIA, 2013.

[Agu+16] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Jean Roman, and Mawussi Zounon. "Numerical recovery strategies for parallel resilient Krylov linear solvers". In: *Numerical Linear Algebra with Applications* 23.5 (2016), pp. 888–905.

[Agu+20a] Emmanuel Agullo, Mirco Altenbernd, Hartwig Anzt, Leonardo Bautista-Gomez, Tommaso Benacchio, Luca Bonaventura, Hans-Joachim Bungartz, Sanjay Chatterjee, Florina M. Ciorba, Nathan DeBardeleben, Daniel Drzisga, Sebastian Eibl, Christian Engelmann, Wilfried N. Gansterer, Luc Giraud, Dominik Göddeke, Marco Heisig, Fabienne Jezequel, Nils Kohl, Xiaoye Sherry Li, Romain Lion, Miriam Mehl, Paul Mycek, Michael Obersteiner, Enrique S. Quintana-Orti, Francesco Rizzi, Ulrich Ruede, Martin Schulz, Fred Fung, Robert Speck, Linda Stals, Keita Teranishi, Samuel Thibault, Dominik Thoennes, Andreas Wagner, and Barbara Wohlmuth. *Resiliency in Numerical Algorithm Design for Extreme Scale Simulations*. 2020. arXiv: 2010.13342 [cs.DC].

[Agu+20b] Emmanuel Agullo, Siegfried Cools, Emrullah Fatih Yetkin, Luc Giraud, Nick Schenkels, and Wim Vanroose. "On soft errors in the Conjugate Gradient method: sensitivity and robust numerical detection". In: *SIAM Journal on Scientific Computing* 42.6 (2020), pp. C335–C358.

[Avi+04] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[Baj+07] Michael A. Bajura, Younes Boulghassoul, Riaz Naseer, Sandeepan DasGupta, Arthur F. Witulski, Jeff Sondeen, Scott D. Stansberry, Jeffrey Draper, Lloyd W. Massengill, and John N. Damoulakis. "Models and algorithmic limits for an ECC-based approach to hardening sub-100-nm SRAMs". In: *IEEE Transactions on Nuclear Science* 54.4 (2007), pp. 935–945.

[Bal+14] Grey Ballard, Erin Carson, James Demmel, Mark Hoemmen, Nicholas Knight, and Oded Schwartz. "Communication lower bounds and optimal algorithms for numerical linear algebra". In: *Acta Numerica* 23 (2014), pp. 1–155.

[Bal+18] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. *PETSc Users Manual*. Tech. rep. ANL-95/11 - Revision 3.9. Argonne National Laboratory, 2018.

[Bal+97] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. "Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries". In: *Modern Software Tools for Scientific Computing*. Birkhäuser Boston, 1997, pp. 163–202.

[Bau05] Robert Baumann. "Soft errors in advanced computer systems". In: *IEEE Design & Test of Computers* 22.3 (2005), pp. 258–266.

[BBM82] Agnès Braun, Ernest Braun, and Stuart MacDonald. *Revolution in miniature: The history and impact of semiconductor electronics*. Cambridge University Press, 1982.

[BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial*. SIAM, 2000.

[Bla+13] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. "Post-failure recovery of MPI communication capability: Design and rationale". In: *The International Journal of High Performance Computing Applications* 27.3 (2013), pp. 244–254.

[Bor10] Shekhar Borkar. "The exascale challenge". In: *Proceedings of 2010 International Symposium on VLSI Design, Automation and Test*. IEEE. 2010, pp. 2–3.

[Bos+14] George Bosilca, Aurelien Bouteiller, Thomas Herault, Yves Robert, and Jack Dongarra. "Assessing the impact of ABFT and checkpoint composite strategies". In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE. 2014, pp. 679–688.

[Bos+15] George Bosilca, Aurelien Bouteiller, Thomas Herault, Yves Robert, and Jack Dongarra. "Composing resilience techniques: ABFT, periodic and incremental checkpointing". In: *International Journal of Networking and Computing* 5.1 (2015), pp. 2–25.

[BS08] Greg Bronevetsky and Bronis de Supinski. "Soft error vulnerability of iterative linear algebra methods". In: *Proceedings of the 22nd Annual International Conference on Supercomputing*. 2008, pp. 155–164.

[Cap+14] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. "Toward exascale resilience: 2014 update". In: *Supercomputing Frontiers and Innovations: an International Journal* 1.1 (2014), pp. 5–28.

[Cas+18] Ralph H. Castain, Joshua Hursey, Aurelien Bouteiller, and David Solt. "Pmix: process management for exascale environments". In: *Parallel Computing* 79 (2018), pp. 9–29.

[CGW19] Marc Casas, Wilfried N. Gansterer, and Elias Wimmer. "Resilient gossip-inspired all-reduce algorithms for high-performance computing: Potential, limitations, and open questions". In: *The International Journal of High Performance Computing Applications* 33.2 (2019), pp. 366–383.

[Cha+07] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. "Collective communication: theory, practice, and experience". In: *Concurrency and Computation: Practice and Experience* 19.13 (2007), pp. 1749–1783.

[Cha+15] Sourav Chakraborty, Hari Subramoni, Adam Moody, Akshay Venkatesh, Jonathan Perkins, and Dhabaleswar K. Panda. "Non-Blocking PMI Extensions for Fast MPI Startup". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015, pp. 131–140.

[Cha+20] Sourav Chakraborty, Ignacio Laguna, Murali Emani, Kathryn Mohror, Dhabaleswar K. Panda, Martin Schulz, and Hari Subramoni. "EReinit: Scalable and efficient fault-tolerance for bulk-synchronous MPI applications". In: *Concurrency and Computation: Practice and Experience* 32.3 (2020).

[Che11] Zizhong Chen. "Algorithm-based Recovery for Iterative Methods Without Checkpointing". In: *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. 2011, pp. 73–84.

[Con+08] Cristian Constantinescu, Ishwar Parulkar, Rick Harper, and Sarah Michalak. "Silent Data Corruption–Myth or reality?" In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE. 2008, pp. 108–109.

[Con08] Cristian Constantinescu. "Intermittent faults and effects on reliability of integrated circuits". In: *2008 Annual Reliability and Maintainability Symposium*. IEEE. 2008, pp. 370–374.

[Dau+17] Daniel Dauwe, Sudeep Pasricha, Anthony A. Maciejewski, and Howard Jay Siegel. "An analysis of resilience techniques for exascale computing platforms". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2017, pp. 914–923.

[Dem+15] James W. Demmel, Laura Grigori, Ming Gu, and Hua Xiang. "Communication Avoiding Rank Revealing QR Factorization with Column Pivoting". In: *SIAM Journal on Matrix Analysis and Applications* 36.1 (2015), pp. 55–89.

[DH11] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software* 38.1 (2011), 1:1–1:25.

[DLP03] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. "The LINPACK benchmark: past, present and future". In: *Concurrency and Computation: practice and experience* 15.9 (2003), pp. 803–820.

[DMR21] Anwesha Das, Frank Mueller, and Barry Rountree. "Systemic assessment of node failures in HPC production platforms". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2021, pp. 267–276.

[DMS97] Jack J. Dongarra, Hans W. Meuer, and Erich Strohmaier. "TOP500 supercomputer sites". In: *Supercomputer* 13 (1997), pp. 89–111.

[DN16] Kiril Dichev and Dimitrios S. Nikolopoulos. "TwinPCG: Dual thread redundancy with forward recovery for preconditioned conjugate gradient methods". In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2016, pp. 506–514.

[Du+12] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. "Algorithm-based fault tolerance for dense matrix factorizations". In: *Acm sigplan notices* 47.8 (2012), pp. 225–234.

[Du+20] Yishu Du, Loris Marchal, Guillaume Pallez, and Yves Robert. *Optimal Checkpointing Strategies for Iterative Applications*. Research Report RR-9371. Inria - Research Centre Grenoble – Rhône-Alpes, 2020.

[Egw+13] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems". In: *The Journal of Supercomputing* 65.3 (2013), pp. 1302–1326.

[EHM14] James John Elliott, Mark Frederick Hoemmen, and Frank Mueller. *Skeptical Programming and Selective Reliability.* Tech. rep. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2014.

[Ema+17] Murali Emani, Ignacio Laguna, Kathryn Mohror, Nawrin Sultana, and Anthony Skjellum. *Checkpointable MPI: A Transparent Fault-Tolerance Approach for MPI.* Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[ES13] Nosayba El-Sayed and Bianca Schroeder. "Reading between the lines of failure logs: Understanding how HPC systems fail". In: *2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE. 2013, pp. 1–12.

[Fal06] Robert D. Falgout. *An introduction to algebraic multigrid.* Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2006.

[Fan+16] Aiman Fang, Ignacio Laguna, Kento Sato, Tanzima Islam, and Kathryn Mohror. *Fault tolerance assistant (fta): An exception handling programming model for mpi applications.* Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[Fra17] Michael P. Frank. "Throwing computing into reverse". In: *IEEE Spectrum* 54.9 (2017), pp. 32–37.

[Gam+14] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. "Exploring automatic, online failure recovery for scientific applications at extreme scales". In: *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 895–906.

[Gan+13] Wilfried N. Gansterer, Gerhard Niederbrucker, Hana Straková, and Stefan Schulze Grotthoff. "Scalable and fault tolerant orthogonalization based on randomized distributed data aggregation". In: *Journal of Computational Science* 4.6 (2013), pp. 480–488.

[Gou09] Brian Gough. *GNU Scientific Library Reference Manual.* 3rd ed. Network Theory Ltd., 2009.

[GR17] Al Geist and Daniel A. Reed. "A survey of high-performance computing scaling challenges". In: *The International Journal of High Performance Computing Applications* 31.1 (2017), pp. 104–113.

[HA84] Kuang-Hua Huang and Jacob A. Abraham. "Algorithm-Based Fault Tolerance for Matrix Operations". In: *IEEE transactions on computers* 33.6 (1984), pp. 518–528.

[Hel+20] Stijn Heldens, Pieter Hijma, Ben Van Werkhoven, Jason Maassen, Adam SZ. Belloum, and Rob V. Van Nieuwpoort. "The landscape of exascale research: A data-driven literature analysis". In: *ACM Computing Surveys (CSUR)* 53.2 (2020), pp. 1–43.

[Hor+20] Atsushi Hori, Kazumi Yoshinaga, Thomas Herault, Aurélien Bouteiller, George Bosilca, and Yutaka Ishikawa. "Overhead of using spare nodes". In: *The International Journal of High Performance Computing Applications* 34.2 (2020), pp. 208–226.

[HR15] Thomas Herault and Yves Robert. *Fault-Tolerance Techniques for High-Performance Computing*. 1st ed. Springer Publishing Company, Incorporated, 2015.

[HSS12] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2012, pp. 111–122.

[JBA15] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. "Flow updating: Fault-tolerant aggregation for dynamic networks". In: *Journal of Parallel and Distributed Computing* 78 (2015), pp. 53–64.

[Kat+18] Amogh Katti, Giuseppe Di Fatta, Thomas Naughton, and Christian Engelmann. "Epidemic failure detection and consensus for extreme parallelism". In: *The International Journal of High Performance Computing Applications* 32.5 (2018), pp. 729–743.

[Kat91] Edgar Katzer. "Multigrid methods for hyperbolic equations". In: *Multigrid methods III*. Springer, 1991, pp. 253–263.

[Key88] Robert W. Keyes. "Miniaturization of electronics and its limits". In: *IBM Journal of Research and Development* 32.1 (1988), pp. 84–88.

[Kis02] Laszlo B. Kish. "End of Moore's law: thermal (noise) death of integration in micro and nano electronics". In: *Physics Letters A* 305.3-4 (2002), pp. 144–149.

[KK20] Israel Koren and C. Mani Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2020.

[Lag+16] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Kathryn Mohror, and Howard Pritchard. "Evaluating and extending user-level fault tolerance in MPI applications". In: *The International Journal of High Performance Computing Applications* 30.3 (2016), pp. 305–319.

[Lan+08] Julien Langou, Zizhong Chen, George Bosilca, and Jack Dongarra. "Recovery patterns for iterative methods in a parallel unstable environment". In: *SIAM Journal on Scientific Computing* 30.1 (2008), pp. 102–116.

[Li+13] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S. Vetter. "Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach". In: *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2013, pp. 1–12.

[Lia+18] Xiang-ke Liao, Kai Lu, Can-qun Yang, Jin-wen Li, Yuan Yuan, Ming-che Lai, Li-bo Huang, Ping-jing Lu, Jian-bin Fang, and Jing Ren. "Moving from exascale to zettascale computing: challenges and techniques". In: *Frontiers of Information Technology & Electronic Engineering* 19.10 (2018), pp. 1236–1244.

[Los+20] Nuria Losada, Patricia González, María J. Martín, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. "Fault tolerance of MPI applications in exascale systems: The ULFM solution". In: *Future Generation Computer Systems* 106 (2020), pp. 467–481.

[LV62] Robert E. Lyons and Wouter Vanderkulk. "The use of triple-modular redundancy to improve computer reliability". In: *IBM journal of research and development* 6.2 (1962), pp. 200–209.

[Mac11] Chris A. Mack. "Fifty years of Moore's law". In: *IEEE Transactions on semiconductor manufacturing* 24.2 (2011), pp. 202–207.

[Mes15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. https://www.mpi-forum.org/docs/. 2015.

[Mes17] Message Passing Interface Forum. *User Level Failure Mitigation*. http://fault-tolerance.org/. 2017.

[MP58] J.A. Morton and W.J. Pietenpol. "The technological impact of transistors". In: *Proceedings of the IRE* 46.6 (1958), pp. 955–959.

[Nic+19] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 911–920.

[Pac+19] Carlos Pachajoa, Markus Levonyak, Wilfried N. Gansterer, and Jesper Larsson Träff. "How to make the preconditioned conjugate gradient method resilient against multiple node failures". In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.

[Pac+20] Carlos Pachajoa, Christina Pacher, Markus Levonyak, and Wilfried N. Gansterer. "Algorithm-based checkpoint-recovery for the conjugate gradient method". In: *49th International Conference on Parallel Processing (ICPP)*. 2020, pp. 1–11.

[PEG20] Carlos Pachajoa, Robert Ernstbrunner, and Wilfried N. Gansterer. "A Generic Strategy for Node-Failure Resilience for Certain Iterative Linear Algebra Methods". In: *2020 IEEE/ACM 10th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE. 2020, pp. 41–50.

[PG17] Carlos Pachajoa and Wilfried N. Gansterer. "On the resilience of conjugate gradient and multigrid methods to node failures". In: *European Conference on Parallel Processing*. Springer. 2017, pp. 569–580.

[PLG18] Carlos Pachajoa, Markus Levonyak, and Wilfried N. Gansterer. "Extending and Evaluating Fault-Tolerant Preconditioned Conjugate Gradient Methods". In: *2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE. 2018, pp. 49–58.

[PPG19] Carlos Pachajoa, Christina Pacher, and Wilfried N. Gansterer. "Node-failure-resistant preconditioned conjugate gradient method without replacement nodes". In: *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE. 2019, pp. 31–40.

[RPG14] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. "Characterizing the impact of intermittent hardware faults on programs". In: *IEEE Transactions on Reliability* 64.1 (2014), pp. 297–310.

[Saa03] Yusuf Saad. *Iterative Methods for Sparse Linear Systems*. 2nd ed. SIAM, 2003.

[Sat+12] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. "Design and Modeling of a Non-Blocking Checkpointing System". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012.

[She94] Jonathan Richard Shewchuk. *An introduction to the conjugate gradient method without the agonizing pain*. 1994. URL: https://www.cs.cmu.edu/~jrs/jrspapers.html.

[Shi+02] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *Proceedings International Conference on Dependable Systems and Networks*. IEEE. 2002, pp. 389–398.

[Sim12] Horst D. Simon. "Barriers to exascale computing". In: *International Conference on High Performance Computing for Computational Science*. Springer. 2012, pp. 1–3.

[SKB12] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. "Algorithmic approaches to low overhead fault detection for sparse linear algebra". In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE. 2012, pp. 1–12.

[SKB13] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance". In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2013, pp. 1–12.

[Sni+14] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A. Chien, Paul Coteus, Nathan A. DeBardeleben, Pedro C. Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. "Addressing failures in exascale computing". In: *The International Journal of High Performance Computing Applications* 28.2 (2014), pp. 129–173.

[Sri+04] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. "The impact of technology scaling on lifetime reliability". In: *International Conference on Dependable Systems and Networks, 2004*. IEEE. 2004, pp. 177–186.

[SSR11] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. "Characterizing the impact of soft errors on iterative methods in scientific computing". In: *Proceedings of the international conference on Supercomputing*. 2011, pp. 152–161.

[SV13] Piyush Sao and Richard Vuduc. "Self-stabilizing iterative solvers". In: *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 2013, pp. 1–8.

[Tah+21] Amir Taherin, Tirthak Patel, Giorgis Georgakoudis, Ignacio Laguna, and Devesh Tiwari. "Examining Failures and Repairs on Supercomputers with Multi-GPU Compute Nodes". In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2021, pp. 305–313.

[Tao+16] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z. Zhang, Darren Kerbyson, and Zizhong Chen. "New-sum: A novel online ABFT scheme for general iterative methods". In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 2016, pp. 43–55.

[TH14] Keita Teranishi and Michael A. Heroux. "Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM". In: *Proceedings of the 21st European MPI Users' Group Meeting*. 2014, pp. 51–56.

[Tom+17] Martin Tomlinson, Cen Jung Tjhai, Marcel A. Ambroze, Mohammed Ahmed, and Mubarak Jibril. *Error-Correction Coding and Decoding: Bounds, Codes, Decoders, Analysis and Applications*. 1st ed. Springer Nature, 2017.

[TOS00] Ulrich Trottenberg, Cornelius W. Oosterlee, and Anton Schuller. *Multigrid*. 1st ed. Elsevier, 2000.

[VY00] Henk A. Van Der Vorst and Qiang Ye. "Residual replacement strategies for Krylov subspace iterative methods for the convergence of true residuals". In: *SIAM Journal on Scientific Computing* 22.3 (2000), pp. 835–852.

[WC14] Panruo Wu and Zizhong Chen. "FT-ScaLAPACK: Correcting soft errors online for ScaLAPACK Cholesky, QR, and LU factorization routines". In: *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 2014, pp. 49–60.

[Wu+16] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. "Towards practical algorithm based fault tolerance in dense linear algebra". In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 2016, pp. 31–42.

[YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on job scheduling strategies for parallel processing*. Springer. 2003, pp. 44–60.

Appendix A

Selected proofs

To support the corollary to follow, we introduce Sylvester's criterion to determine if a matrix is symmetric, positive definite.

Theorem 1 (Sylvester's criterion)**.** *A hermitian matrix $\boldsymbol{A}$ is positive definite if and only if the determinant of all of the upper-left submatrices (the principal minors), and of $\boldsymbol{A}$ itself, are positive.*

We also observe that a row-column permutation maintains symmetric, positiveness.

Lemma 1. *If $\boldsymbol{A}$ is a symmetric, positive definite matrix, then $\boldsymbol{TAT}^\intercal$ is also a symmetric, positive matrix, where $\boldsymbol{T}$ is a permutation matrix.*

Proof. We define the vector $\boldsymbol{y} = \boldsymbol{Tx}$, and notice that $\boldsymbol{y}$ is non-zero if and only if $\boldsymbol{x}$ is non-zero. We define $\boldsymbol{A}' \coloneqq \boldsymbol{TAT}^\intercal$, representing the row-column permutation of matrix $\boldsymbol{A}$. Then, since $\boldsymbol{A}$ is positive definite and $\boldsymbol{T}^{-1} = \boldsymbol{T}^\intercal$ because permutation matrices are orthonormal, we have $\boldsymbol{y}^\intercal \boldsymbol{A}' \boldsymbol{y} = \boldsymbol{x}^\intercal \boldsymbol{T}^\intercal \boldsymbol{TAT}^\intercal \boldsymbol{Tx} = \boldsymbol{x}^\intercal \boldsymbol{Ax} > 0$ if $\boldsymbol{x} \neq \boldsymbol{0}$. Therefore, the matrix $\boldsymbol{A}'$ is positive definite. Furthermore, $\boldsymbol{A}'^\intercal = (\boldsymbol{TAT}^\intercal)^\intercal = \boldsymbol{TA}^\intercal \boldsymbol{T}^\intercal = \boldsymbol{TAT}^\intercal = \boldsymbol{A}'$, Therefore $\boldsymbol{A}'$ is also symmetric. □

Using Sylvester's criterion and Lemma 1, we now introduce the following corollary. It guarantees that a diagonal submatrix of a positive-definite matrix is itself positive definite and thus defines a linear system with a unique solution. It is used in Sections 5.1.2 and 5.2.1.

Corollary 1. *If $\boldsymbol{A}$ is a symmetric, positive definite (SPD) matrix then, for any choice of index set I_j, the submatrix $\boldsymbol{A}_{I_j,I_j}$ is SPD.*

Proof. There exists a permutation matrix $\boldsymbol{T}$ that reorders the entries of a vector such that the entries corresponding to I_j are moved to the first $|I_j|$ positions.

We define a matrix $\boldsymbol{A}' = \boldsymbol{TAT}^\intercal$. Because of our selection of $\boldsymbol{T}$, the submatrix $\boldsymbol{A}_{I_j,I_j}$ is in the upper-left corner of $\boldsymbol{A}'$. According to Lemma 1, $\boldsymbol{A}'$ is SPD.

From Sylvester's criterion, since $\boldsymbol{A}'$ is SPD its upper-left $k \times k$ matrix, $\boldsymbol{A}'^{k}$, has a positive determinant, as well as all the upper-left square matrices of size smaller than $k \times k$. Thus, $\boldsymbol{A}'^{k}$ itself fulfills Sylvester's criterion and is therefore SPD. Consequently, $\boldsymbol{A}_{I_j,I_j}$ is SPD. □

The following corollary insures that a non-singular subsystem can always be formed for purposes of reconstruction. We use this result in Section 7.3.1.

Corollary 2. *Given an invertible matrix $\boldsymbol{A}$ and an index set I_F, there exists an index set I_r with $|I_r| = |I_F|$ such that the matrix $\boldsymbol{A}_{I_r,I_F}$ is invertible.*

Proof. Because $\boldsymbol{A}$ is invertible, all of its columns are linearly independent. Therefore, the dimension of the column space of $\boldsymbol{A}_{I,I_F}$ is $|I_F|$. By the fundamental theorem of linear algebra, the dimension of the row space of $\boldsymbol{A}_{I,I_F}$ must also be $|I_F|$, thus, there are $|I_F|$ linearly independent rows in $\boldsymbol{A}_{I,I_F}$. If the index set I_r represents this selection of rows, then the square matrix $\boldsymbol{A}_{I_r,I_F}$ has $|I_F|$ linearly independent rows, and is therefore invertible. □

We also show that if we replace the values of the solution of the discretized one-dimensional Poisson equation for a fixed-size interval, simulating linear interpolation after a node failure, the increase in the residual norm does not depend on which node failed. We use this proof to explain some of our experimental results in Section 4.4.4.

Theorem 2. *Suppose that the vector x is the solution of the finite-difference discretization of the one-dimensional Poisson equation*

$$x : [0,1] \to \mathbb{R}, \quad \Delta x = f, \quad x(0) = x(1) = 0$$

for a constant forcing function f, whose solution can be found by solving a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A}$ can be described by the three-point stencil $\frac{1}{h^2}\begin{bmatrix}1 & -2 & 1\end{bmatrix}$, where h is the distance between two neighboring grid points, and $\boldsymbol{b}$ is a vector whose entries are the value of the function f.

If a contiguous section of the solution vector is replaced by a sequence of values linearly interpolating the values immediately to the left and to the right of this section, then the residual norm after this change does not depend on the location of the section.

Proof. Let $\hat{\boldsymbol{x}}$ represent the current approximation to the solution. The vector $\boldsymbol{e}$ is the error of the current approximation. The i-th component of the vectors are designated $\boldsymbol{x}_i$, $\boldsymbol{e}_i$, $\boldsymbol{b}_i$ and $\hat{\boldsymbol{x}}_i$.

We model the failure of a node responsible for the index range $\{\alpha + 1 \,..\, \beta - 1\}$. Starting from the solution of the linear system, $\boldsymbol{x}$, the loss of information and subsequent interpolation in the index set can be represented by the addition of an error vector $\boldsymbol{e}$. The

approximation to the solution after the interpolation is then $\hat{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{e}$, where $\boldsymbol{e}$ is defined as follows:

$$\boldsymbol{e}_i = \begin{cases} \left(\boldsymbol{x}_\alpha + (i-\alpha)\frac{\boldsymbol{x}_\beta - \boldsymbol{x}_\alpha}{\beta-\alpha}\right) - \boldsymbol{x}_i & \text{if } i \in \{\alpha+1 \,..\, \beta-1\} \\ 0 & \text{otherwise} \end{cases} \tag{A.1}$$

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}} = \boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x} + \boldsymbol{e}) = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}\boldsymbol{e} = -\boldsymbol{A}\boldsymbol{e} \tag{A.2}$$

For indexes outside of the range $\{\alpha \,..\, \beta\}$, the entries of the residual are zero.

For indexes in the range $\{\alpha+2 \,..\, \beta-2\}$, the set of points covered by the matrix stencil will be, in general, non-zero. Then, inspecting Eq. (A.1), we split the error into three components: a constant part ($\boldsymbol{x}_\alpha$), a linear part ($(i-\alpha)(\boldsymbol{x}_\beta - \boldsymbol{x}_\alpha)/(\beta-\alpha)$), and part of the solution ($\boldsymbol{x}_i$). Multiplying $\boldsymbol{A}$ times vectors of linearly increasing or decreasing entries produces a zero vector, therefore constant and the linear parts are canceled in Eq. (A.2), and what remains is the effect of the matrix on the entries of the solution: In this interval, the residual is equal to the corresponding entry of the right-hand-side vector:

$$\boldsymbol{r}_i = \boldsymbol{b}_i \text{ if } i \in \{\alpha+2..\beta-2\}. \tag{A.3}$$

If the index equals α, the corresponding residual entry will be

$$\boldsymbol{r}_\alpha = \frac{1}{h^2}\left(-\boldsymbol{e}_{\alpha-1} + 2\boldsymbol{e}_\alpha - \boldsymbol{e}_{\alpha+1}\right).$$

Both $\boldsymbol{e}_{\alpha-1}$ and $\boldsymbol{e}_\alpha$ are equal to zero since $\hat{\boldsymbol{x}}$ matches $\boldsymbol{x}$ for these indices. We have

$$\boldsymbol{r}_\alpha = -\frac{1}{h^2}\,\boldsymbol{e}_{\alpha+1} = \frac{1}{h^2}\left((\boldsymbol{x}_{\alpha+1} - \boldsymbol{x}_\alpha) - \frac{\boldsymbol{x}_\beta - \boldsymbol{x}_\alpha}{\beta-\alpha}\right). \tag{A.4}$$

That is, the residual at the point is proportional to the difference of the secants for the grid point of index a. Furthermore, the term $\boldsymbol{x}_b - \boldsymbol{x}_a$ can be expanded into the following telescoping sum:

$$\boldsymbol{x}_\beta - \boldsymbol{x}_\alpha = (\boldsymbol{x}_\beta - \boldsymbol{x}_{\beta-1}) + (\boldsymbol{x}_{\beta-1} - \boldsymbol{x}_{\beta-2}) + \cdots + (\boldsymbol{x}_{\alpha+1} - \boldsymbol{x}_\alpha). \tag{A.5}$$

The number of summands in parenthesis is $\beta - \alpha$.

Since $\boldsymbol{x}$ is the solution of the discretized Poisson system, a set of three consecutive indexes in the array will fulfill

$$\boldsymbol{x}_{i+1} - \boldsymbol{x}_i = \boldsymbol{x}_i - \boldsymbol{x}_{i-1} + k_0 h^2 \tag{A.6}$$

under the condition that all entries of the right-hand-side vector are the constant k_0.

By plugging Eq. (A.6) in (A.5), we obtain the following:

$$\begin{aligned} \boldsymbol{x}_\beta - \boldsymbol{x}_\alpha = \; & (\boldsymbol{x}_{\alpha+1} - \boldsymbol{x}_\alpha) + (\boldsymbol{x}_{\alpha+1} - \boldsymbol{x}_\alpha + k_0 h^2) \\ & + \cdots + (\boldsymbol{x}_{\alpha+1} - \boldsymbol{x}_\alpha + (\beta - \alpha - 1) k_0 h^2) \\ = \; & (\beta - \alpha)(\boldsymbol{x}_{\alpha+1} - \boldsymbol{x}_\alpha) + \tfrac{k_0 h^2}{2}(\beta - \alpha)(\beta - \alpha - 1) \end{aligned} \tag{A.7}$$

Plugging Eq. (A.7) back in (A.4), we obtain:

$$\boldsymbol{r}_\alpha = k_0 \frac{1 - (\beta - \alpha)}{2}. \tag{A.8}$$

For $i = \alpha + 1$ we obtain

$$\begin{aligned} \boldsymbol{r}_{\alpha+1} = \; & \tfrac{1}{h^2}(-e_\alpha + 2\boldsymbol{e}_{\alpha+1} - \boldsymbol{e}_{\alpha+2}) \\ = \; & \tfrac{1}{h^2}\left(2\left(\boldsymbol{x}_\alpha + \tfrac{\boldsymbol{x}_\beta - \boldsymbol{x}_\alpha}{\beta - \alpha} - \boldsymbol{x}_{\alpha+1}\right) - \left(\boldsymbol{x}_\alpha + 2\tfrac{\boldsymbol{x}_\beta - \boldsymbol{x}_\alpha}{\beta - \alpha} - \boldsymbol{x}_{\alpha+2}\right)\right) \\ = \; & \tfrac{1}{h^2}\left(\boldsymbol{x}_\alpha - 2\boldsymbol{x}_{\alpha+1} + \boldsymbol{x}_{\alpha+2}\right) \\ = \; & \boldsymbol{b}_{\alpha+1} \\ = \; & k_0 \end{aligned} \tag{A.9}$$

Similarly, for the indexes $\beta - 1$ and β we obtain

$$\boldsymbol{r}_{\beta-1} = \boldsymbol{b}_{\beta-1} = k_0 \tag{A.10}$$

$$\boldsymbol{r}_\beta = k_0 \frac{1 - (\beta - \alpha)}{2} \tag{A.11}$$

According to Eq. (A.3) (A.8), (A.9), (A.10) and (A.11), the norm of the residual vector does not depend on the location of the interpolated region (therefore, it does not depend on the location of the index set of the failing node), but only on the size of the region where information was lost. □