universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Tolerating Node Failures in Communication-avoiding Conjugate Gradient Methods“

verfasst von / submitted by

Viktoria Mayer, BSc

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

Master of Science (MSc)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wilfried Gansterer, MSc

Acknowledgements

I would like to thank my supervisor Prof. Wilfried Gansterer for his support and guidance throughout creating this thesis.
Furthermore, I would like to thank my family for their support during my studies.

Abstract

Today's growing number of nodes in large-scale parallel computers raise two challenges. Firstly, global communication becomes a major bottleneck. Secondly, the likelihood of node failures increases with the number of nodes. The Preconditioned Conjugate Gradient (PCG) method is an iterative solver for large sparse linear systems facing these challenges.

A commonly used approach to make an algorithm fault-tolerant is Checkpoint-Restart, which periodically saves the state of a solver. To avoid the often considerable overhead of Checkpoint-Restart, algorithm-specific properties can be exploited. The Exact State Reconstruction (ESR) method for the PCG algorithm recovers the lost parts of the state in case of node failures after retrieving redundantly stored data from neighbour nodes. This data is communicated during the execution of resilient PCG with very little overhead by exploiting the inherent data redundancy of the solver.

On large-scale parallel computers, communication becomes a major bottleneck in the PCG method. The parallel matrix-vector products typically only involve communication between local neighbour nodes. However, the scalar products require global reduction operations involving all nodes, resulting in synchronization steps in each iteration. Communication-avoiding s-step methods reduce the number of global synchronizations by a factor of $\mathcal{O}(s)$ by computing the iterations of PCG in blocks of s.

We extend two communication-avoiding s-step methods to make them resilient against node failures using similar ideas as the ESR strategy for the standard PCG method. Theoretical and experimental evaluations show that our new approach is able to achieve resilience with low overhead compared to the non-resilient methods both in the failure-free case and when node failures occur while preserving communication-avoiding properties and therefore scalability of the algorithms.

Kurzfassung

Die wachsende Anzahl von Knoten bei parallelen Großrechnern wirft zwei Herausforderungen auf. Zum einen wird die globale Kommunikation zu einem schwerwiegenden Leistungsengpass. Zum anderen steigt die Wahrscheinlichkeit für Knotenausfälle mit der Knotenanzahl. Das Verfahren der Konjugierten Gradienten mit Vorkonditionierung (PCG-Verfahren) ist ein iteratives Verfahren zur Lösung von großen dünn besetzten linearen Gleichungssystemen, das mit diesen beiden Herausforderungen konfrontiert ist.

Ein häufig verwendeter Ansatz, um einen Algorithmus gegen Knotenausfälle zu schützen, ist Checkpoint-Restart, bei dem der derzeitige Zustand des Verfahrens periodisch gesichert wird. Um den oft erheblichen Zusatzaufwand von Checkpoint-Restart zu vermeiden, können algorithmus-spezifische Eigenschaften verwendet werden. Die Methode "Exact State Reconstruction" (ESR) für das PCG-Verfahren stellt die verlorenen Teile des Zustands zum Zeitpunkt der Knotenausfälle wieder her, nachdem redundant gespeicherte Daten von einem Nachbarknoten erhalten wurden. Diese redundant gespeicherten Daten können während der Ausführung des resilienten PCG-Verfahrens mit sehr wenig Zusatzaufwand kommuniziert werden, indem die dem Algorithmus inhärente Datenredundanz ausgenutzt wird.

Auf parallelen Großrechnern wird die globale Kommunikation zu einem schwerwiegenden Leistungsengpass im PCG-Verfahren. Die parallelen Matrix-Vektor-Produkte benötigen oft nur Kommunikation mit lokalen Nachbarknoten. Die Skalarprodukte hingegen benötigen globale Reduktionsoperationen, in die alle Knoten involviert sind und die daher zu Synchronisationsschritten in jeder Iteration führen. Kommunikationsvermeidende s-step-Verfahren reduzieren die Anzahl der globalen Synchronisationsschritte um den Faktor $\mathcal{O}(s)$, indem die PCG-Iterationen in Blöcken on s berechnet werden.

Wir erweitern zwei kommunikationsvermeidende s-step-Verfahren, sodass diese resilient gegen Knotenausfälle sind. Dabei verwenden wir ähnliche Ansätze wie bei ESR für das Standard-PCG-Verfahren. Theoretische und experimentelle Evaluierungen zeigen, dass unsere neuen Algorithmen imstande sind, Resilienz mit wenig Zusatzaufwand im Vergleich zu den nicht resilienten Methoden zu erreichen, sowohl im fehlerfreien Fall als auch im Fall von Knotenausfällen. Dabei bleiben die kommunikationsvermeidenden Eigenschaften und daher die Skalierbarkeit dieser Verfahren erhalten.

Contents

List of Tables

List of Figures

List of Algorithms

1. Introduction

Resilience against node failures becomes increasingly important on today's large-scale parallel computers. A general approach is the widely used Checkpoint-Restart strategy, where the current state of the algorithm is saved periodically, which is called a checkpoint. When a fault occurs, i.e. a node fails and its data is lost, the last saved state can be retrieved and the algorithm can be continued from the last checkpoint. Other approaches use algorithm-specific strategies to either approximate or exactly reconstruct the lost data. These approaches need very little overhead when used for the Conjugate Gradient (CG) method, a Krylov subspace method for symmetric positive-definite sparse matrices.

On large-scale parallel computers, global communication becomes a major bottleneck. In methods that require the computation of scalar products of dense vectors that are distributed among all nodes, global reduction operations are necessary.

In iterative solvers for sparse matrices, the system matrix and the vectors are distributed among the nodes. For the matrix-vector products in these methods, the distributed parts of the vectors must be communicated between nodes. Depending on the sparsity structure of the matrix, each node might only have to communicate with a few neighbours. However, the scalar products in each iteration require a global reduction operation involving all nodes, resulting in a synchronization step.

Communication-hiding methods overlap this global communication with local communication and computation using non-blocking Allreduce operations. These algorithms still require a synchronization step in every or at least every second iteration. Communication-avoiding s-step methods reduce the number of global synchronizations by $\mathcal{O}(s)$ by rearranging the Conjugate Gradient method such that s iterations can be computed without local and global communication. The distinction between communication-hiding and communication-avoiding methods can be misleading. Communication-hiding methods reduce the number of global reduction operations per iteration by a constant factor and can thus also be seen as communication-avoiding.

When developing algorithms for large-scale parallel computers, both resilience against node failures and the avoidance or hiding of communication bottlenecks have to be considered. While there already exist algorithm-based resilient versions of the standard Conjugate Gradient algorithm and of communication-hiding methods, to the best of our knowledge, literature still lacks resilient communication-avoiding Conjugate Gradient methods.

1.1. Problem statement

The main research question of the master's thesis is how communication-avoiding s-step Conjugate Gradient methods can be made resilient against node failures. Our approach is based on the Exact State Reconstruction (ESR) strategy for the Preconditioned Conjugate

Gradient (PCG) method. Two communication-avoiding methods will be extended to tolerate node failures and their runtime performance will be evaluated theoretically and experimentally.

1.2. Related work

Related work on communication-hiding methods

The Pipelined Preconditioned Conjugate Gradient method (PPCG) presented in Ghysels and Vanroose (2014) hides global communication by overlapping it with local communication and computation. The PCG algorithm is rearranged such that a non-blocking Allreduce operation is started for the scalar products and the result of this operation is not needed until after the matrix-vector product and the preconditioner application. An improved algorithm with deep pipelines proposed by Cools et al. (2019) overlaps communication over several iterations. Tiwari and Vadhiyar (2020) introduce an algorithm based on PPCG and increases runtime performance by using only one global communication operation every two iterations.

The standard PCG algorithm has two global synchronization steps in each iteration. In the algorithms in Ghysels and Vanroose (2014) and Cools et al. (2019), only one global reduction operation must be finalized in each iteration, the algorithm in Tiwari and Vadhiyar (2020) even needs only one synchronization step every two iterations. Thus, these methods are communication-avoiding. However, even when using deep pipelines as in Cools et al. (2019), the number of global reduction operations is only reduced by a constant factor. Moreover, pipelined methods leave the cost of communication for the matrix-vector products and preconditioner applications unchanged.

Related work on communication-avoiding methods

Early methods that aim to avoid communication include D'Azevedo and Romine (1992), where the PCG algorithm is rearranged such that the two scalar products in each iteration can be combined into one global reduction operation, and a PCG method that uses three-term recurrences (Rutishauser, 1959). However, the accuracy of three-term recurrences in Krylov subspace methods is lower than of two-term recurrences in finite precision (Gutknecht and Strakos, 2000). As with pipelined methods, these algorithms reduce the amount of global reduction operations by a constant factor, and the communication cost for the matrix-vector products and preconditioner applications remains unchanged.

Other communication-avoiding methods aim to reduce communication latency by a factor of $\mathcal{O}(s)$ both in terms of global reduction operations and of communication for the matrix-vector products and preconditioner applications. One of the first s-step CG methods developed by Chronopoulos and Gear (1989b) blocks s iterations of standard CG by using a matrix-matrix product instead of s matrix-vector products, where the second matrix has s columns. This algorithm suffers from numerical instabilities, thus Chronopoulos and Gear (1989b) only use $s \leq 5$. A preconditioned version of this approach is presented by Chronopoulos and Gear (1989a).

Hoemmen (2010) gives a detailed description of different s-step basis types. While the methods presented in Chronopoulos and Gear (1989b) and Chronopoulos and Gear (1989a) can only use the monomial basis, newer methods are able to use other basis types to increase numerical stability (Toledo, 1995; Hoemmen, 2010).

The Communication-avoiding Conjugate Gradient (CA-CG) algorithm presented in Toledo (1995) divides the loop of standard CG into an outer and an inner loop. The s iterations of the inner loop can be computed without communication. In Carson et al. (2014), a deflation strategy for CA-CG is proposed, which aims to accelerate the convergence rate and can thus be seen as a type of preconditioning.

In Hoemmen (2010), a communication-avoiding method using similar ideas as Toledo (1995) is proposed, which is based on the three-term recurrence version of the CG method. We denote this algorithm as CA-CG3. In contrast to Toledo (1995), Hoemmen (2010) provides an explicit formulation of a left-preconditioned version of their algorithm, which we denote as CA-PCG3 and which has been applied in large-scale simulations in Mayumi et al. (2016) and Idomura et al. (2018b).

To avoid communication latency for the matrix-vector products, the s-step basis in CA-CG, CA-CG3 and CA-PCG3 is computed with the Matrix Powers Kernel proposed by Demmel et al. (2007), its performance is further optimized in Demmel et al. (2008) and Mohiyuddin et al. (2009). Demmel et al. (2007) extend the Matrix Powers Kernel to a left-preconditioned version, if the monomial basis is used. Hoemmen (2010) identifies sets of non-trivial preconditioners for which communication can be avoided in the Preconditioned Matrix Powers Kernel. Knight et al. (2013) show how data sparsity can be exploited in the Matrix Powers Kernel for matrices with specific structures.

Both communication-hiding and communication-avoiding methods are identical to standard CG respectively PCG in infinite precision arithmetic. However, they have different numerical error propagation, which slows down or prevents convergence for many input matrices. In Carson (2015), an overview of different communication-avoiding Krylov subspace methods is given as well as their numerical stability analyses.

Ghysels and Vanroose (2014) suggest to compute the residual as $r^{(i)} = Ax^{(i)} - b$ in every 50^{th} iteration of PPCG to increase accuracy instead of computing it recursively. Cools et al. (2016) and Cools et al. (2018) propose a more advanced approach of residual replacement where in each iteration the deviation of the recursively computed residual from the true residual is estimated. An analog version for CA-CG and other s-step Krylov subspace methods using residual replacement is presented in Carson and Demmel (2014).

Carson (2018) and Carson (2020) propose an adaptive method based on CA-CG. Instead of using a fixed step size s throughout the algorithm, this approach uses varying step sizes by evaluating how large the step size can be in each outer iteration without too much loss of accuracy.

Related work on resilient methods

Different types of failures can occur on large-scale parallel computers. Existing literature on tolerance of the PCG method against soft errors, i.e. spontaneous changes of the state of the solver, caused for example by bit flips, include Shantharam et al. (2012), Zhang and He (2013) and Schöll et al. (2015). The general approach to make an algorithm resilient against node failures, e.g. a node crashes or looses its connection to the network, is Checkpoint-Restart. A detailed overview of different versions can be found in Herault and Robert (2015).

In Bosilca et al. (2014) and Bosilca et al. (2015), checkpointing is combined with Algorithm Based Fault Tolerance (ABFT) to make applications tolerant against node failures. Operations in an application can be made resilient with algorithm-based approaches in order to decrease the number of necessary checkpoints.

Langou et al. (2007) present an Interpolation Restart (IR) method for Krylov subspace methods. The lost parts of the solution vector are interpolated and the Krylov subspace method is restarted with this interpolation as the initial guess. This approach is extended in Agullo et al. (2013) and Agullo et al. (2016).

In Chen (2011), an algorithm-based approach is presented to make the PCG method and the Biconjugate Gradient Stabilized (BiCGstab) method resilient against node failures. During the communication for the matrix-vector products, additional vector parts are sent to neighbour nodes with little overhead to store redundant copies of the search direction in each iteration. If a node failure occurs, the redundant copy of the lost vector part can be retrieved and used to reconstruct the rest of the lost data.

This resilient PCG method is refined in Pachajoa et al. (2018) and extended to tolerate multiple simultaneous or overlapping node failures in Pachajoa et al. (2019). Pachajoa et al. (2018) and Pachajoa et al. (2019) denote their algorithm to recover the lost data in case of node failures as Exact State Reconstruction (ESR). Pachajoa et al. (2020) reduces the overhead of this approach in the failure-free case by reducing the frequency of storing redundant copies. In Levonyak et al. (2020), a resilient version of PPCG (Ghysels and Vanroose, 2014) is presented, which is based on the algorithm-based strategy in Chen (2011), Pachajoa et al. (2018) and Pachajoa et al. (2019).

1.2.1. Scientific Gap

Current research topics include improving the scalability of iterative solvers and cost-efficient resilience against node failures in algorithms for execution on large-scale parallel computers. This thesis focuses on scalability and resilience of Conjugate Gradient methods.

To develop a version of the Conjugate Gradient algorithm which is suitable for large-scale parallel computers, both resilience against node failures and communication-avoidance can be combined. While a resilient version of the communication-hiding PPCG has been presented in Levonyak et al. (2020), to the best of our knowledge there does not yet exist literature about algorithm-based resilience in communication-avoiding algorithms.

1.3. Notation and conventions

The choice of the distribution of the sparse system matrix among nodes is not restricted in communication-avoiding Conjugate Gradient methods (Hoemmen, 2010). However, literature on algorithm-based resilience of Conjugate Gradient methods usually assume that the system matrix and the preconditioner are block-row distributed and vectors are distributed accordingly, see e.g. Agullo et al. (2016), Chen (2011), Pachajoa et al. (2018), Pachajoa et al. (2019), Pachajoa et al. (2020) and Levonyak et al. (2020). In this thesis, we will therefore assume this distribution.

To address parts of a vector or a matrix owned by a specific node, we use a similar notation as in Agullo et al. (2016) and Pachajoa et al. (2018). We refer to the set of all indices as $I = \{0, 1, ..., n\}$, with n being the problem size. The index subset representing vector elements of a node p is denoted as I_p. The part of a vector v that is owned by node

p is denoted as v_{I_p}. We use the same notation for matrices whose number of columns is much smaller than n.

Similarly, indices of rows and columns of matrices of size $n \times n$ are denoted with index sets. A_{I_p,I_q} denotes the part of the matrix A defined by the row indices I_p and the column indices I_q. To simplify notation, we denote the preconditioner $P = M^{-1}$.

In case of a single node failure, we designate the failed node as f. In case of multiple node failures, we designate the union index sets of all failed nodes as I_f. Thus, the lost parts of the vector v are denoted as v_{I_f} in both cases.

We consider the parallel execution of the solvers on multiple nodes with distributed memory. To be able to communicate between nodes, we assume that the parallel runtime environment provides functionality comparable to the standard Message Passing Interface (MPI) (Message Passing Interface Forum, 2021a).

If a node fails, its data is lost and reconstructed on a replacement node. This replacement node is either a spare node or one of the surviving nodes and takes the place of the failed node when the solver is continued after recovery.

We assume that the runtime environment provides fault-tolerance features as the MPI extension User Level Failure Mitigation (ULFM) (Message Passing Interface Forum, 2021b; Bland et al., 2013). This extension provides the ability to detect node failures during communication operations and to notify the surviving nodes about node failures and which nodes have failed. Moreover, it includes a mechanism for providing replacement nodes and prevents global and local communication from being indefinitely blocked (Losada et al., 2020).

As Pachajoa et al. (2018) and Pachajoa et al. (2019), we define the state of a solver as the (not necessarily minimal) data that defines the future behaviour of the solver, i.e. the data that is necessary to be able to continue the solver after node failures with the same convergence as in the failure-free case.

The Conjugate Gradient algorithms discussed in this thesis are mathematically equivalent. We refer to a *step* as the equivalent of an iteration of the standard PCG method, i.e. s steps of an communication-avoiding s-step Conjugate Gradient method are mathematically equivalent to s iterations of PCG.

We denote a zero matrix of size $i \times j$ as $0_{i,j}$.

1.4. Chapter overview

Chapter 2 of the thesis contains a description of PCG and of PCG3, the three-term recurrence version that is equivalent to PCG in exact precision arithmetic.

Chapter 3 contains state-of-the-art s-step Conjugate Gradient algorithms that aim to reduce communication latency by a factor of $\mathcal{O}(s)$ to increase performance on large-scale parallel computers. Two communication-avoiding methods are investigated, one based on standard CG (CA-CG), and one that is based on the three-term recurrence version (CA-CG3 respectively its preconditioned version CA-PCG3). CA-CG is extended to a left-preconditioned version, which to our knowledge does not yet exist in literature and which we will refer to as CA-PCG.

Chapter 4 describes the problem of node failures and already existing approaches of making Conjugate Gradient methods resilient against node failures. We describe in detail an algorithm-based approach for resilience in PCG that exploits the inherent data redundancy due to the communication for the matrix-vector product in each iteration. In case of node failures, the lost parts of the state are recovered using the Exact State Reconstruction (ESR) strategy.

In Chapter 5, we develop novel resilient versions of CA-PCG and CA-PCG3 based on similar ideas as used for the Exact State Reconstruction (ESR) strategy for PCG. While the state of resilient CA-PCG can be reconstructed using almost the same ESR as for PCG, the amount of data that has to be reconstructed after node failures during resilient CA-PCG3 increases with s. Chapter 5 also provides a theoretical evaluation of the cost of incorporating resilience into CA-PCG and CA-PCG3. We show that the extensions for resilience do not influence the scalability of the communication-avoiding algorithms.

Chapter 6 provides an experimental evaluation regarding the performance of these novel resilient communication-avoiding methods both in the failure-free case and when failures occur. Our experiments confirmed our theoretical evaluation in the failure-free case and showed that the step size s does not have a significant impact on the recovery cost in case of failures for the band matrices used.

Chapter 7 contains a conclusion of the resilient communication-avoiding Conjugate Gradient methods that have been developed and analyzed.

2. The Preconditioned Conjugate Gradient method

The Conjugate Gradient (CG) method was originally published by Hestenes and Stiefel (1952) and is an algorithm for computing the solution $x \in \mathbb{R}^n$ of the system of linear equations $Ax = b$ with a large sparse and symmetric positive-definite (SPD) matrix $A \in \mathbb{R}^{n \times n}$ and a right-hand side $b \in \mathbb{R}^n$. While the method computes a solution in exact arithmetic in at most n steps, it is typically used as an iterative method for sparse matrices that are too large to be solved efficiently by a direct solver. To accelerate convergence, a preconditioner $P = M^{-1}$ can be used to reduce the condition number of the system. In the Preconditioned Conjugate Gradient (PCG) method, the system $PAx = Pb$ is solved (Barrett et al., 1994).

The Conjugate Gradient method can be classified as a Krylov subspace method. A Krylov subspace is built by repeatedly applying a matrix to a start vector. In Krylov subspace methods, the start vector is the residual $r^{(0)} = b - Ax^{(0)}$, where $x^{(0)}$ is an arbitrary initial guess of the solution. In the unpreconditioned case, the Krylov subspace then has the form

$$\mathcal{K}_i\left(A, r^{(0)}\right) = \operatorname{span}\left\{r^{(0)}, Ar^{(0)}, \ldots, A^{i-1}r^{(0)}\right\}. \tag{2.1}$$

In iteration i, the Krylov subspace method minimizes the residual over the space

$$x^{(0)} + \mathcal{K}_i\left(A, r^{(0)}\right). \tag{2.2}$$

The method terminates if convergence or a user-defined maximum number of iterations is reached (Kelley, 1995). Krylov subspace methods have a variety of applications such as solving systems of linear equations, linear least squares problems or eigenvalue problems (Barrett et al., 1994).

Sections 2.1 and 2.2 are based on Hestenes and Stiefel (1952), Kelley (1995) and Saad (2003), and show the basic principles of the CG algorithm and its preconditioned version PCG. Section 2.3 describes the three-term recurrence variants CG3 and PCG3 and is based on Saad (2003) and Hoemmen (2010).

2.1. The Conjugate Gradient (CG) method

Solving the system of linear equations $Ax = b$ is equivalent to minimizing the quadratic form

$$f(x) = \frac{1}{2}x^T Ax - bx + c \tag{2.3}$$

by setting the derivative $f'(x) = Ax - b$ to zero, with c being a scalar. Starting with an initial guess $x^{(0)}$, in each iteration i the approximate solution $x^{(i)}$ is updated with a search direction $p^{(i)}$ that is A-orthogonal to the search directions of all previous iterations, i.e. $p^{(i)}Ap^{(j)} = 0$ for $i \neq j$.

$$x^{(i+1)} = x^{(i)} + \alpha^{(i)} p^{(i)} \tag{2.4}$$

By multiplying Equation (2.4) by $-A$ and adding b, we get a recursive update equation for the residual $r^{(i+1)} = b - Ax^{(i+1)}$.

$$r^{(i+1)} = r^{(i)} - \alpha^{(i)} A p^{(i)} \tag{2.5}$$

Since each search direction $p^{(i)}$ is only used once, we have to find the minimum point along $p^{(i)}$. Thus, we derive $f\left(x^{(i+1)}\right)$ for $\alpha^{(i)}$ and set this derivation to zero.

$$\begin{aligned} \frac{d}{d\alpha^{(i)}} f\left(x^{(i+1)}\right) &= f'\left(x^{(i+1)}\right)^T \frac{d}{d\alpha^{(i)}} x^{(i+1)} \\ &= -r^{(i+1)^T} \frac{d}{d\alpha^{(i)}} \left(x^{(i)} + \alpha^{(i)} p^{(i)}\right) = -r^{(i+1)^T} p^{(i)} = 0 \end{aligned} \tag{2.6}$$

We therefore know that $r^{(i+1)^T} p^{(i)} = \left(r^{(i)} - \alpha^{(i)} A p^{(i)}\right)^T p^{(i)}$ is zero and can write

$$\alpha^{(i)} = \frac{r^{(i)^T} p^{(i)}}{p^{(i)^T} A p^{(i)}}. \tag{2.7}$$

To define the search directions, we use a successive A-Orthogonalization process. We define a set of n linear independent vectors $v^{(0)}, v^{(1)}, \ldots, v^{(n-1)}$ and choose these vectors to be the residuals, i.e. $v^{(i)} = r^{(i)}$. To get $p^{(i)}$, we subtract all components of $v^{(i)} = r^{(i)}$ that are not A-orthogonal to all previous search directions.

$$p^{(i)} = r^{(i)} + \sum_{k=0}^{i-1} \beta^{(i-1,k)} p^{(k)} \tag{2.8}$$

For the values $\beta^{(i-1,j)}$ we take the inner product of Equation (2.8) and $Ap^{(j)}$. Since the search directions are A-orthogonal, we can write the following for $i > j$.

$$p^{(i)^T} A p^{(j)} = r^{(i)^T} A p^{(j)} + \sum_{k=0}^{i-1} \beta^{(i-1,k)} p^{(k)^T} A p^{(j)} \tag{2.9}$$

$$0 = r^{(i)^T} A p^{(j)} + \beta^{(i-1,j)} p^{(j)^T} A p^{(j)} \tag{2.10}$$

$$\beta^{(i-1,j)} = -\frac{r^{(i)^T} A p^{(j)}}{p^{(j)^T} A p^{(j)}} \tag{2.11}$$

In exact arithmetic, we have the exact solution $x^{(n)} = x$ after at most n iterations, the residual $r^{(n)}$ is zero. Using Equation (2.5), we can write

$$r^{(n)} = r^{(0)} - \sum_{j=0}^{n-1} \alpha^{(j)} A p^{(j)} = 0, \tag{2.12}$$

$$r^{(i)} = r^{(0)} - \sum_{j=0}^{i-1} \alpha^{(j)} A p^{(j)} = \sum_{j=i}^{n-1} \alpha^{(j)} A p^{(j)}. \tag{2.13}$$

By multiplying $r^{(i)}$ in Equation (2.13) with any search direction $p^{(k)}$ with $k < i$, we see that the residual is orthogonal to all previous search directions. If we take the inner product of Equation (2.8) and $r^{(j)}$ with $j \geq i$, we get

$$p^{(i)^T} r^{(j)} = r^{(i)^T} r^{(j)} + \sum_{k=0}^{i-1} \beta^{(i-1,k)} p^{(k)^T} r^{(j)} = r^{(i)^T} r^{(j)}. \tag{2.14}$$

If $j > i$, $p^{(i)^T} r^{(j)} = r^{(i)^T} r^{(j)} = 0$. Thus, the residual is orthogonal to all previous residuals. Using Equation (2.14) we are able to define $\alpha^{(i)}$ as

$$\alpha^{(i)} = \frac{r^{(i)^T} r^{(i)}}{p^{(i)^T} A p^{(i)}}. \tag{2.15}$$

If we computed $p^{(i)}$ with Equation (2.8), we would have to store all previous search directions. To simplify this, we take the inner product of $r^{(i)}$ and Equation (2.5) for $r^{(j+1)}$.

$$r^{(i)^T} r^{(j+1)} = r^{(i)^T} r^{(j)} - \alpha^{(j)} r^{(i)^T} A p^{(j)} \tag{2.16}$$

$$\alpha^{(j)} r^{(i)^T} A p^{(j)} = r^{(i)^T} r^{(j)} - r^{(i)^T} r^{(j+1)} \tag{2.17}$$

Since all residuals are orthogonal to each other, $r^{(i)^T} A p^{(j)}$ is zero if $i \neq j$ and $i \neq j+1$. We do not have to consider the case where $i = j$ because we only need to consider $i > j$ for Equation (2.11). For $i = j + 1$, we can write

$$r^{(i)^T} A p^{(j)} = -\frac{1}{\alpha^{(i-1)}} r^{(i)^T} r^{(i)}. \tag{2.18}$$

All values of $\beta^{(i-1,j)}$ are therefore zero except $\beta^{(i-1,i-1)}$, which we will call $\beta^{(i-1)}$ from now on. Using Equations (2.11), (2.15) and (2.18), we get the following term for $\beta^{(i)}$.

$$\beta^{(i)} = -\frac{-\frac{1}{\alpha^{(i)}} r^{(i+1)^T} r^{(i+1)}}{p^{(i)^T} A p^{(i)}} = \frac{r^{(i+1)^T} r^{(i+1)}}{r^{(i)^T} r^{(i)}} \tag{2.19}$$

From Equation (2.8) we can form a recursive definition for the search direction.

$$p^{(i+1)} = r^{(i+1)} + \beta^{(i)} p^{(i)} \tag{2.20}$$

2.2. The Preconditioned Conjugate Gradient (PCG) method

Let $\kappa(A)$ be the spectral condition number of the system matrix A and $\|v\|_A = \sqrt{v^T A v}$ be the A-norm of a vector v. As stated in Saad (2003), the convergence rate of the CG method is

$$\left\| x^{(i)} - x \right\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^i \left\| x^{(0)} - x \right\|_A , \tag{2.21}$$

where x is the exact solution and $x^{(i)}$ is the approximate solution after i iterations. Since A is SPD,

$$\kappa(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}, \tag{2.22}$$

where $\lambda_{min}(A)$ and $\lambda_{max}(A)$ are the smallest and largest eigenvalues of A.

Due to round-off errors, the CG method is often not able to converge to the exact solution in n iterations, especially if the matrix has a high condition number. Moreover, in cases where the matrix is very large, it is often not feasible to compute even n steps. In practice, typically a preconditioner $P = M^{-1}$ is used to increase the convergence rate. Instead of solving $Ax = b$, the system $PAx = Pb$ is solved. M approximates A such that $PA = M^{-1}A$ has a lower condition number than A.

Like A, the matrix M is SPD and can be decomposed into $M = LL^T$ using Cholesky decomposition. Consequently, $P = M^{-1} = L^{-T}L^{-1}$. To preserve symmetry of the matrix of the linear system, we can split the preconditioner P and transform the problem $Ax = b$ into another problem, namely solving $\hat{A}\hat{x} = \hat{b}$ for $\hat{x} = L^T x$, where $\hat{A} = L^{-1}AL^{-T}$ and $\hat{b} = L^{-1}b$.

The approximate solution in each iteration for this transformed system is $\hat{x}^{(i)} = L^T x^{(i)}$, the search direction is defined as $\hat{p}^{(i)} = L^T p^{(i)}$. The residual is

$$\hat{r}^{(i)} = \hat{b} - \hat{A}\hat{x}^{(i)} = L^{-1}\left(b - Ax^{(i)}\right) = L^{-1}r^{(i)}. \tag{2.23}$$

The preconditioned residual is defined as $u^{(i)} = Pr^{(i)}$. The inner product ${\hat{r}^{(i)}}^T\hat{r}^{(i)}$ is computed as

$${\hat{r}^{(i)}}^T\hat{r}^{(i)} = {r^{(i)}}^T L^{-T}L^{-1}r^{(i)} = {r^{(i)}}^T Pr^{(i)} = {r^{(i)}}^T u^{(i)}. \tag{2.24}$$

The recursive update equation for $\hat{x}^{(i+1)} = L^{-T}x^{(i+1)}$ can be replaced by $x^{(i+1)} = x^{(i)} + \alpha^{(i)}L^{-T}\hat{p}^{(i)}$ since $\hat{x}$ is not needed. Similarly, we can use the recursive update equation $r^{(i+1)} = r^{(i)} - \alpha^{(i)}AL^{-T}\hat{p}^{(i)}$ to compute $r^{(i+1)}$ instead of $\hat{r}^{(i+1)}$. If we replace $\hat{p}^{(i)}$ by $L^T p^{(i)}$ in these two recursions, we get the update recursions defined in Equations (2.4) and (2.5).

The inner product ${\hat{p}^{(i)}}^T\hat{A}\hat{p}^{(i)}$ can be rewritten as

$${\hat{p}^{(i)}}^T\hat{A}\hat{p}^{(i)} = {\hat{p}^{(i)}}^T L^{-1}AL^{-T}\hat{p}^{(i)} = p^{(i)}Ap^{(i)}. \tag{2.25}$$

The recursive update equation of the search direction $\hat{p}^{(i+1)} = \hat{r}^{(i)} + \beta^{(i)}\hat{p}_i$ can be multiplied with L^{-T}.

$$p^{(i+1)} = u^{(i)} + \beta^{(i)}p^{(i)} \tag{2.26}$$

Since PCG is a Krylov subspace method, by induction

$$u^{(i)}, p^{(i)} \in \mathcal{K}_i\left(PA, u^{(0)}\right), \tag{2.27}$$

$$x^{(i)} - x^{(0)} \in \mathcal{K}_{i-1}\left(PA, u^{(0)}\right). \tag{2.28}$$

The Preconditioned Conjugate Gradient (PCG) algorithm is outlined in Algorithm 1.

While there exist a large variety of different types of preconditioners, those that do not require global communication are preferred in communication-hiding and communication-avoiding methods to preserve scalability (Hoemmen, 2010; Cools et al., 2019). Examples are the Jacobi and the Block Jacobi preconditioner. When using the Block Jacobi preconditioner, M consists of diagonal blocks of A. A Block Jacobi preconditioner with block size 1 is the Jacobi preconditioner, M consists of the diagonal elements of A.

Algorithm 1 PCG (Preconditioned Conjugate Gradient algorithm)

```
Input: SPD system matrix $A \in \mathbb{R}^{n\times n}$, preconditioner $P \in \mathbb{R}^{n\times n}$,
       right-hand side $b \in \mathbb{R}^n$, initial guess $x^{(0)} \in \mathbb{R}^n$
Output: approximate solution $x^{(i)} \in \mathbb{R}^n$ of $Ax = b$
1: $r^{(0)} = b - Ax^{(0)}, u^{(0)} = Pr^{(0)}, p^{(0)} = u^{(0)}$
2: for $i = 0, 1, ...$ until convergence do
3:     $\alpha^{(i)} = {r^{(i)}}^T u^{(i)} / {p^{(i)}}^T A p^{(i)}$
4:     $x^{(i+1)} = x^{(i)} + \alpha^{(i)} p^{(i)}$
5:     $r^{(i+1)} = r^{(i)} - \alpha^{(i)} A p^{(i)}$
6:     $u^{(i+1)} = P r^{(i+1)}$
7:     $\beta^{(i)} = {r^{(i+1)}}^T u^{(i+1)} / {r^{(i)}}^T u^{(i)}$
8:     $p^{(i+1)} = u^{(i+1)} + \beta^{(i)} p^{(i)}$
9: end for
```

In both cases, the preconditioner application is very efficient and does not require much communication between nodes. The blocks of the Block Jacobi preconditioner can be chosen such that no block overlaps different nodes, such that no communication is necessary. The Jacobi preconditioner does not require any communication between nodes.

2.3. The three-term recurrence variant PCG3

As shown by Rutishauser (1959), the Conjugate Gradient method can be represented with three-term instead of two-term recurrences. The three-term recurrence method CG3 and its preconditioned version PCG3 are mathematically equivalent to CG respectively PCG. In exact arithmetic, the residual vector $r^{(i)}$ in iteration i of CG3 or PCG3 is equal to the corresponding residual vector in CG respectively PCG. Instead of computing a search direction, CG3 and PCG3 solely compute the residual and the approximate solution in each iteration using three-term recurrences. This section is based on Saad (2003) and Hoemmen (2010).

The residual vectors of CG are non-zero scalar multiples of the symmetric Lanczos basis vectors, which are computed using a three-term recurrence. Based on this three-term recurrence, we can write the following recursive definition for the residual vectors in CG3. By definition, $x^{(-1)} = 0_{n,1}$, thus $r^{(-1)} = b$.

$$r^{(i+1)} = \rho^{(i)} \left(r^{(i)} - \gamma^{(i)} A r^{(i)} \right) + \left(1 - \rho^{(i)} \right) r^{(i-1)} \tag{2.29}$$

By subtracting b on both sides of Equation (2.29) and then multiplying with $-A^{-1}$, we get a recursive formula for the approximate solution vectors.

$$x^{(i+1)} = \rho^{(i)} \left(x^{(i)} + \gamma^{(i)} r^{(i)} \right) + \left(1 - \rho^{(i)} \right) x^{(i-1)} \tag{2.30}$$

To get a definition for $\gamma^{(i)}$, we take the inner product of Equation (2.29) and $r^{(i)}$. Since all residuals are orthogonal to each other (see Section 2.1),

$$r^{(i)^T} r^{(i+1)} = \rho^{(i)} \left(r^{(i)^T} r^{(i)} - \gamma^{(i)} r^{(i)^T} A r^{(i)} \right) + \left(1 - \rho^{(i)} \right) r^{(i)^T} r^{(i-1)}, \tag{2.31}$$

$$\gamma^{(i)} = \frac{r^{(i)^T} r^{(i)}}{r^{(i)^T} A r^{(i)}}. \tag{2.32}$$

For the definition of $\rho^{(0)}$, we use Equation (2.29) for $r^{(1)}$ and the relation $r^{(0)} = b - Ax^{(0)}$.

$$\begin{aligned} r^{(1)} &= \rho^{(0)} \left(r^{(0)} - \gamma^{(0)} A r^{(0)} \right) + \left(1 - \rho^{(0)} \right) b \\ &= \rho^{(0)} \left(-Ax^{(0)} - \gamma^{(0)} A r^{(0)} \right) + r^{(0)} + Ax^{(0)} \\ &= r^{(0)} + \left(1 - \rho^{(0)} \right) Ax^{(0)} - \rho^{(0)} \gamma^{(0)} A r^{(0)} \end{aligned} \tag{2.33}$$

We take the inner product of $r^{(1)}$ and $r^{(0)}$ and insert Equation (2.32) for $\gamma^{(0)}$.

$$r^{(0)^T} r^{(1)} = r^{(0)^T} r^{(0)} + \left(1 - \rho^{(0)} \right) r^{(0)^T} A x^{(0)} - \rho^{(0)} \gamma^{(0)} r^{(0)^T} A r^{(0)} \tag{2.34}$$

$$0 = \left(1 - \rho^{(0)} \right) r^{(0)^T} r^{(0)} + \left(1 - \rho^{(0)} \right) r^{(0)^T} A x^{(0)} \tag{2.35}$$

Since the initial guess $x^{(0)}$ can be chosen arbitrarily, $\rho^{(0)}$ has to be 1 for Equation (2.35) to hold for any chosen $x^{(0)}$.

For the definition of $\rho^{(i)}$ for $i > 0$, we take the inner product of $r^{(i)}$ and Equation (2.29) for $r^{(i)}$.

$$\begin{aligned} r^{(i)^T} r^{(i)} &= \rho^{(i-1)} \left(r^{(i)^T} r^{(i-1)} - \gamma^{(i-1)} r^{(i)^T} A r^{(i-1)} \right) + \left(1 - \rho^{(i-1)} \right) r^{(i)^T} r^{(i-2)} \\ &= -\rho^{(i-1)} \gamma^{(i-1)} r^{(i)^T} A r^{(i-1)} \end{aligned} \tag{2.36}$$

Next, we take the inner product of $r^{(i-1)}$ and Equation (2.29).

$$r^{(i-1)^T} r^{(i+1)} = \rho^{(i)} \left(r^{(i-1)^T} r^{(i)} - \gamma^{(i)} r^{(i-1)^T} A r^{(i)} \right) + \left(1 - \rho^{(i)} \right) r^{(i-1)^T} r^{(i-1)} \tag{2.37}$$

$$0 = -\rho^{(i)} \gamma^{(i)} r^{(i-1)^T} A r^{(i)} + \left(1 - \rho^{(i)} \right) r^{(i-1)^T} r^{(i-1)} \tag{2.38}$$

Since we only consider the case where real arithmetic is used, $r^{(i)^T} A r^{(i-1)} = r^{(i-1)^T} A r^{(i)}$. We can thus use Equations (2.36) and (2.38) to formulate a definition of $\rho^{(i)}$ for $i > 0$.

$$\rho^{(i)} = \left(1 - \frac{r^{(i)^T} r^{(i)}}{r^{(i-1)^T} r^{(i-1)}} \cdot \frac{\gamma^{(i)}}{\gamma^{(i-1)}} \cdot \frac{1}{\rho^{(i-1)}} \right)^{-1}. \tag{2.39}$$

The derivation of the preconditioned version PCG3 is analogous to that for PCG. Again, we define $\hat{r}^{(i)} = L^{-1} r^{(i)}$ and $u^{(i)} = P r^{(i)}$. The recurrence equation for $\hat{r}^{(i+1)}$ can be multiplied with L. Since $\hat{A} = L^{-1} A L^{-T}$,

$$r^{(i+1)} = \rho^{(i)} \left(r^{(i)} - \gamma^{(i)} A u^{(i)} \right) + \left(1 - \rho^{(i)} \right) r^{(i-1)}. \tag{2.40}$$

Algorithm 2 PCG3 (3-term recurrence variant of PCG)

```
Input: SPD system matrix $A \in \mathbb{R}^{n \times n}$, preconditioner $P \in \mathbb{R}^{n \times n}$,
       right-hand side $b \in \mathbb{R}^n$, initial guess $x^{(0)} \in \mathbb{R}^n$
Output: approximate solution $x^{(i)} \in \mathbb{R}^n$ of $Ax = b$
 1: $r^{(-1)} = 0_{n,1}, u^{(-1)} = 0_{n,1}, x^{(-1)} = 0_{n,1}$
 2: $r^{(0)} = b - Ax^{(0)}, u^{(0)} = Pr^{(0)}$
 3: for $i = 0, 1, ...$ until convergence do
 4:     $w^{(i)} = Au^{(i)}$
 5:     $v^{(i)} = Pw^{(i)}$
 6:     $\mu^{(i)} = {r^{(i)}}^T u^{(i)}$
 7:     $\nu^{(i)} = {w^{(i)}}^T u^{(i)}$
 8:     $\gamma^{(i)} = \mu^{(i)} / \nu^{(i)}$
 9:     if $i = 0$ then
10:         $\rho^{(i)} = 1$
11:     else
12:         $\rho^{(i)} = \left(1 - \frac{\gamma^{(i)}}{\gamma^{(i-1)}} \cdot \frac{\mu^{(i)}}{\mu^{(i-1)}} \cdot \frac{1}{\rho^{(i-1)}}\right)^{-1}$
13:     end if
14:     $x^{(i+1)} = \rho^{(i)} \left(x^{(i)} + \gamma^{(i)} u^{(i)}\right) + \left(1 - \rho^{(i)}\right) x^{(i-1)}$
15:     $r^{(i+1)} = \rho^{(i)} \left(r^{(i)} - \gamma^{(i)} w^{(i)}\right) + \left(1 - \rho^{(i)}\right) r^{(i-1)}$
16:     $u^{(i+1)} = \rho^{(i)} \left(u^{(i)} - \gamma^{(i)} v^{(i)}\right) + \left(1 - \rho^{(i)}\right) u^{(i-1)}$
17: end for
```

We can adapt the inner products in the same way as for PCG. Since we do not need $\hat{x}$, the recursive update equation for $\hat{x}^{(i+1)} = L^T x^{(i+1)}$ is replaced by

$$x^{(i+1)} = \rho^{(i)} \left(x^{(i)} + \gamma^{(i)} u^{(i)}\right) + \left(1 - \rho^{(i)}\right) x^{(i-1)}. \tag{2.41}$$

The preconditioned residual $u^{(i)} = Pr^{(i)}$ is computed recursively by multiplying Equation (2.40) by P.

$$u^{(i+1)} = \rho^{(i)} \left(u^{(i)} - \gamma^{(i)} PAu^{(i)}\right) + \left(1 - \rho^{(i)}\right) u^{(i-1)} \tag{2.42}$$

The Preconditioned Conjugate Gradient algorithm based on three-term recurrences (PCG3) is outlined in Algorithm 2.

PCG and PCG3 both have to compute two scalar products, one matrix-vector product, one application of the preconditioner and local vector updates in each iteration. In PCG, the two scalar products ${p^{(i)}}^T Ap^{(i)}$ and ${r^{(i+1)}}^T u^{(i+1)}$, which are computed in lines 3 and 7 of Algorithm 1, require two separate global reduction operations since $r^{(i+1)}$ (line 5 of Algorithm 1) cannot be computed without ${p^{(i)}}^T Ap^{(i)}$. In contrast, PCG3 can combine the communication of the two scalar products in lines 6 and 7 of Algorithm 2 in one global collective. Thus, PCG3 is a communication-avoiding algorithm that is slightly more suitable than PCG for large-scale parallel computers with distributed memory.

3. Communication-avoiding Conjugate Gradient methods

On large-scale parallel computers, global communication becomes a major bottleneck in Conjugate Gradient methods. The system matrix A and the vectors are distributed among the nodes. For the matrix-vector product in each iteration, the distributed vector parts must be communicated between nodes. Since A is sparse, depending on its sparsity structure, a node usually only needs to exchange data with a few neighbour nodes, which is referred to as communication with local neighbours or local communication below. However, the scalar products of distributed dense vectors in each iteration require a global reduction operation involving all nodes, resulting in a global synchronization step (Ghysels and Vanroose, 2014).

Pipelined versions of the Conjugate Gradient method such as presented in Ghysels and Vanroose (2014) hide global communication by overlapping it with local communication and computation. Additional auxiliary vectors are introduced, which are updated locally and therefore do not cause additional communication.

The Pipelined PCG (PPCG) method presented in Ghysels and Vanroose (2014) is a rearranged version of PCG. The two scalar products are computed at the beginning of each iteration, their communication can be combined in one global collective. The results of the scalar products are only needed after the matrix-vector product and the application of the preconditioner. Thus, the global communication can be overlapped by those two operations that are assumed to only require local communication.

The cost for overlapping communication is additional local computation for the vector updates of the additional auxiliary vectors. If enough nodes are used, the effect of hiding the communication excels the cost of the additional local computation. Other pipelined algorithms, such as those proposed in Cools et al. (2019) and Tiwari and Vadhiyar (2020), are based on similar principles. The cost of communication for the matrix-vector products and preconditioner applications remains unchanged in pipelined methods (Ghysels and Vanroose, 2014).

Communication-avoiding s-step methods aim to reduce communication latency by a factor of $\mathcal{O}(s)$. Communication only takes place every s steps equivalent to s iterations of the standard PCG method in exact arithmetic. Subsequently, s steps can be performed locally on each node without communication. In contrast to pipelined methods, both communication for the global collective operations and local communication for the matrix-vector products and preconditioner applications can be reduced (Chronopoulos and Gear, 1989b; Chronopoulos and Gear, 1989a; Toledo, 1995; Hoemmen, 2010).

While early methods combine the matrix-vector products of s PCG iterations into one matrix-matrix product (Chronopoulos and Gear, 1989b; Chronopoulos and Gear, 1989a), newer algorithms divide the loop of PCG into an outer and an inner loop, where the latter has s iterations (Toledo, 1995; Hoemmen, 2010). Communication is only necessary at the beginning of each outer iteration, the s iterations of the inner loop can be computed locally on each node without communication.

Section 3.1 describes different basis types that can be used in communication-avoiding s-step algorithms. The communication-avoiding method based on CG (CA-CG) proposed in Toledo (1995) is described in Section 3.2. In Sections 3.3 and 3.4, the communication-avoiding methods based on CG3 (CA-CG3) respectively PCG3 (CA-PCG3) presented in Hoemmen (2010) are explained. To the best of our knowledge, there does not yet exist an explicit formulation of a left-preconditioned version of CA-CG analogously to CA-PCG3. In Section 3.5, we therefore derive the preconditioned version CA-PCG. In Section 3.6, we describe different Matrix Powers Kernel algorithms for computing the basis matrices in communication-avoiding s-step algorithms. We theoretically compare the two algorithms CA-PCG and CA-PCG3 in Section 3.7.

3.1. Choice of the basis

The basis matrices in the communication-avoiding methods presented in Toledo (1995) and Hoemmen (2010) require the computation of $O(s)$ matrix-vector products at the beginning of each outer iteration, which are computed with the Matrix Powers Kernel presented in Demmel et al. (2007). Different Matrix Powers Kernel algorithms are described in Section 3.6.

The choice of the basis is the main factor that influences stability of communication-avoiding Krylov subspace methods (Hoemmen, 2010). The Matrix Powers Kernel computes the columns of the matrix

$$T = \left[\rho_0(A)v, \rho_1(A)v, ..., \rho_s(A)v\right], \tag{3.1}$$

where $v \in \mathbb{R}^n$, $A \in \mathbb{R}^{n\times n}$ and $\rho_j(z)$ is a polynomial of degree j (Carson et al., 2014; Carson, 2018) that satisfies the three-term recurrence

$$\begin{aligned} &\rho_0(z) = 1, \quad \rho_1(z) = (z-\theta_0)\rho_0(z)/\gamma_0, \\ &\rho_j(z) = ((z-\theta_{j-1})\rho_{j-1}(z) + \sigma_{j-2}\rho_{j-2}(z))/\gamma_{j-1}, \quad j \geq 2. \end{aligned} \tag{3.2}$$

When using the monomial basis, the columns of T consist of the first iterations of the Power Iteration, where a start vector is multiplied with the same matrix multiple times.

$$T = [v, Av, ..., A^s v] \tag{3.3}$$

The iterate of the Power Iteration converges to the eigenvector corresponding to the largest eigenvalue. This can lead to severe accuracy loss in Krylov subspace methods if $s > 5$. Since the residuals get smaller in magnitude throughout the iterations of a Conjugate Gradient method, they get vulnerable to round-off errors. In finite precision, these vectors are not linearly independent anymore, which causes the algorithm to converge slower or not at all (Toledo, 1995).

The Newton basis uses shifts obtained from estimating eigenvalues by computing s or $2s$ iterations of standard CG before starting the actual solver (Saad, 2003; Hoemmen, 2010). Another possibility is the Chebyshev basis, which uses scaled and shifted Chebyshev polynomials. A detailed description of these three basis types can be found in Hoemmen (2010).

In the preconditioned case, two basis matrices with the initial vectors w and $v = Pw$ are computed (Hoemmen, 2010).

$$T = [\rho_0(AP)w, \rho_1(AP)w, ..., \rho_s(AP)w] \tag{3.4}$$
$$PT = [\rho_0(PA)v, \rho_1(PA)v, ..., \rho_s(PA)v] \tag{3.5}$$

3.2. Communication-avoiding CG (CA-CG)

The communication-avoiding method based on standard CG introduced in Toledo (1995) divides the loop of CG into an outer and an inner loop and performs the inner loop iterations without communication. While Toledo (1995) name their algorithm "KrylovBasisConjugateGradient", we denote this method as CA-CG as most subsequent publications, e.g. Carson and Demmel (2014), Carson et al. (2014) and Carson (2018). This section is based on Toledo (1995) and Carson (2018).

The outer loop is indexed with $k = 0, 1, \ldots$ and terminates if convergence or a user-defined maximum of iterations is reached. The inner loop has s iterations $j = 0, \ldots, s-1$. We do a linear transformation on the vectors in the CG algorithm by using a basis that is built with the residual $r^{(sk)}$ and the search direction $p^{(sk)}$ at the beginning of each outer iteration k.

In Equation (2.1), the Krylov subspace of the Conjugate Gradient algorithm is built using the start vector $r^{(0)}$. Here, we cannot build the search space based only on $r^{(sk)}$ because $p^{(sk)} \neq r^{(sk)}$. By induction, for $0 \leq j \leq s$,

$$r^{(sk+j)}, p^{(sk+j)} \in \mathcal{K}_{s+1}\left(A, p^{(sk)}\right) + \mathcal{K}_s\left(A, r^{(sk)}\right), \tag{3.6}$$
$$x^{(sk+j)} - x^{(sk)} \in \mathcal{K}_s\left(A, p^{(sk)}\right) + \mathcal{K}_{s-1}\left(A, r^{(sk)}\right). \tag{3.7}$$

At the beginning of outer iteration k, the s-step basis matrix $Y^{(k)} = \left[P^{(k)}, R^{(k)}\right]$ is computed. The Matrix Powers Kernel is invoked twice to compute $P^{(k)}$ and $R^{(k)}$ such that

$$\operatorname{span}\left(P^{(k)}\right) = \mathcal{K}_{s+1}\left(A, p^{(sk)}\right), \tag{3.8}$$
$$\operatorname{span}\left(R^{(k)}\right) = \mathcal{K}_s\left(A, r^{(sk)}\right). \tag{3.9}$$

To compute the next s steps without communication, we define small vectors $p^{(k,j)'}$, $r^{(k,j)'}$ and $x^{(k,j)'}$ for $0 \leq j \leq s$ in a different basis to express the vectors $p^{(sk+j)}$, $r^{(sk+j)}$ and $x^{(sk+j)} - x^{(sk)}$.

$$p^{(sk+j)} = Y^{(k)} p^{(k,j)'} \tag{3.10}$$
$$r^{(sk+j)} = Y^{(k)} r^{(k,j)'} \tag{3.11}$$
$$x^{(sk+j)} - x^{(sk)} = Y^{(k)} x^{(k,j)'} \tag{3.12}$$

In each inner iteration j, these small vectors are updated analogously to the recursive update equations used in CG. At the end of each outer iteration, the vectors in the original basis are regained using Equations (3.10), (3.11) and (3.12).

Algorithm 3 CA-CG (Communication-avoiding CG)

```
Input: SPD system matrix $A \in \mathbb{R}^{n\times n}$, preconditioner $P \in \mathbb{R}^{n\times n}$,
       right-hand side $b \in \mathbb{R}^n$, initial guess $x^{(0)} \in \mathbb{R}^n$, step size $s$
Output: approximate solution $x^{(sk+j)} \in \mathbb{R}^n$ of $Ax = b$
 1: $r^{(0)} = b - Ax^{(0)}, p^{(0)} = r^{(0)}$
 2: for $k = 0, 1, ...$ until convergence do
 3:     Compute $Y^{(k)}$ such that span $\left(Y^{(k)}\right) = \mathcal{K}_{s+1}\left(A, p^{(sk)}\right) + \mathcal{K}_s\left(A, r^{(sk)}\right)$
 4:     Compute $G^{(k)} = {Y^{(k)}}^T Y^{(k)}$
 5:     Assemble $B^{(k)}$ such that $A\underline{Y}^{(k)} = Y^{(k)}B^{(k)}$, see Equation (3.13)
 6:     $p^{(k,0)'} = [1, 0_{1,2s}]^T, r^{(k,0)'} = [0_{1,s+1}, 1, 0_{1,s-1}]^T, x^{(k,0)'} = [0_{1,2s+1}]^T$
 7:     for $j = 0, 1, ..., s-1$ do
 8:         $\alpha^{(sk+j)} = {r^{(k,j)'}}^T G^{(k)} r^{(k,j)'} / {p^{(k,j)'}}^T G^{(k)} B^{(k)} p^{(k,j)'}$
 9:         $x^{(k,j+1)'} = x^{(k,j)'} + \alpha^{(sk+j)} p^{(k,j)'}$
10:         $r^{(k,j+1)'} = r^{(k,j)'} - \alpha^{(sk+j)} B^{(k)} p^{(k,j)'}$
11:         $\beta^{(sk+j)} = {r^{(k,j+1)'}}^T G^{(k)} r^{(k,j+1)'} / {r^{(k,j)'}}^T G^{(k)} r^{(k,j)'}$
12:         $p^{(k,j+1)'} = r^{(k,j+1)'} + \beta^{(sk+j)} p^{(k,j)'}$
13:     end for
14:     $p^{(sk+s)} = Y^{(k)} p^{(k,s)'}$
15:     $r^{(sk+s)} = Y^{(k)} r^{(k,s)'}$
16:     $x^{(sk+s)} = x^{(sk)} + Y^{(k)} x^{(k,s)'}$
17: end for
```

The matrix-vector products $Ap^{(sk+j)}$ are obtained using a matrix $B^{(k)} \in \mathbb{R}^{(2s+1)\times(2s+1)}$ such that

$$A\underline{Y}^{(k)} = Y^{(k)}B^{(k)}, \tag{3.13}$$

with $\underline{Y}^{(k)} = \left[\underline{P}^{(k)}, \underline{R}^{(k)}\right]$, where $\underline{P}^{(k)}$ and $\underline{R}^{(k)}$ are the same as $P^{(k)}$ and $R^{(k)}$ except their respective last column is all zeroes. When using the monomial basis, applying $B^{(k)}$ shifts all columns of $P^{(k)}$ and $R^{(k)}$ to the left. To compute the matrix-vector products in the changed basis in the inner iterations, we compute $B^{(k)}p^{(k,j)'}$.

The scalar products in the inner iterations are computed using the so-called Gram matrix $G^{(k)} = {Y^{(k)}}^T Y^{(k)}$, which is computed at the beginning of outer iteration k.

$${r^{(sk+j)}}^T r^{(sk+j)} = {r^{(k,j)'}}^T {Y^{(k)}}^T Y^{(k)} r^{(k,j)'} = {r^{(k,j)'}}^T G^{(k)} r^{(k,j)'} \tag{3.14}$$

$${p^{(sk+j)}}^T A p^{(sk+j)} = {p^{(k,j)'}}^T {Y^{(k)}}^T Y^{(k)} B^{(k)} p^{(k,j)'} = {p^{(k,j)'}}^T G^{(k)} B^{(k)} p^{(k,j)'} \tag{3.15}$$

The Communication-avoiding Conjugate Gradient algorithm based on CG (CA-CG) is outlined in Algorithm 3.

Only one global reduction operation per outer iteration occurs in CA-CG, compared to $2s$ or s global reductions in s iterations of CG/PCG respectively CG3/PCG3. Hence, communication latency is reduced by a factor of $O(s)$. Moreover, for matrices with special sparsity structures, using the Matrix Powers Kernel can reduce communication latency for the matrix-vector products in each outer iteration by a factor of $O(s)$, see Section 3.6.

While CG computes s matrix-vector products in s iterations, $2s-1$ products are computed in one outer iteration of CA-CG to form $Y^{(k)}$. CG and PCG compute two scalar products in each iteration, i.e. $\mathcal{O}(s)$ scalar products in s iterations. CA-CG computes $\mathcal{O}(s^2)$ scalar products in each outer iteration to form the Gram matrix of size $(2s+1) \times (2s+1)$. However, on large-scale parallel computers, where communication becomes a major bottleneck, this additional local computation is usually negligible (Carson, 2018).

3.3. Communication-avoiding CG3 (CA-CG3)

The communication-avoiding method based on CG3 presented in Hoemmen (2010) is based on similar ideas as CA-CG. Although Hoemmen (2010) and subsequent publications, e.g. Mayumi et al. (2016), Idomura et al. (2018a) and Idomura et al. (2018b), denote this algorithm as "CA-CG" as well, we refer to it as CA-CG3 in this thesis to distinguish it from the communication-avoiding method based on CG in Section 3.2.

The loop of CG3 is partitioned into an outer loop with index $k = 0, 1, \ldots$, which is terminated at convergence or when a maximum number of iterations is reached, and an inner loop with index $j = 0, 1, \ldots, s-1$. At the beginning of outer iteration k, the basis matrix $W^{(k)}$ is computed using the Matrix Powers Kernel such that

$$\operatorname{span}\left(W^{(k)}\right) = \mathcal{K}_{s+1}\left(A, r^{(sk)}\right). \tag{3.16}$$

In contrast to CA-CG, only s matrix-vector products are computed in one outer iteration, the same amount as in s iterations of CG/PCG and CG3/PCG3.

The residual matrix $R^{(k)}$ stores the residual vectors computed in iteration k.

$$R^{(k)} = \left[r^{(sk)}, \ldots, r^{(sk+s-1)}\right] \tag{3.17}$$

In each inner iteration j, the vector $w^{(sk+j)} = Ar^{(sk+j)}$ is formed without explicitly computing $Ar^{(sk+j)}$ by using auxiliary vectors $d^{(sk+j)}$ of size $2s+1$ that fulfill the relation

$$Ar^{(sk+j)} = \left[R^{(k-1)}, W^{(k)}\right] d^{(sk+j)}. \tag{3.18}$$

To determine $d^{(sk+j)}$, we rearrange the recursive update equation of the residual in Equation (2.29) for $r^{(sk+j+1)}$ as

$$Ar^{(sk+j)} = \frac{1-\rho^{(sk+j)}}{\rho^{(sk+j)}\gamma^{(sk+j)}} r^{(sk+j-1)} + \frac{1}{\gamma^{(sk+j)}} r^{(sk+j)} - \frac{1}{\rho^{(sk+j)}\gamma^{(sk+j)}} r^{(sk+j+1)}. \tag{3.19}$$

Equation (3.19) can be summarized for all vectors $Ar^{(sk+j)}$ with $0 \le j < s$ as

$$AR^{(k)} = \frac{1-\rho^{(sk)}}{\rho^{(sk)}\gamma^{(sk)}} r^{(sk-1)} e_1^T + R^{(k)} T^{(k)} - \frac{1}{\rho^{(sk+s-1)}\gamma^{(sk+s-1)}} r^{(sk+s)} e_s^T, \tag{3.20}$$

where e_1 and e_s represent the first and the last column of the $s \times s$ identity matrix. The $s \times s$ matrix $T^{(k)}$ is defined as

$$T^{(k)} = \tilde{T}^{(k)} \cdot \operatorname{diag}\left(\rho^{(sk)}\gamma^{(sk)}, \ldots, \rho^{(sk+s-1)}\gamma^{(sk+s-1)}\right)^{-1}, \tag{3.21}$$

$$\tilde{T}^{(k)} = \begin{pmatrix} \rho^{(sk)} & 1-\rho^{(sk+1)} & 0 & \cdots & 0 \\ -1 & \rho^{(sk+1)} & \ddots & \ddots & \vdots \\ \vdots & & \ddots & & 1-\rho^{(sk+s-1)} \\ & & & -1 & \rho^{(sk+s-1)} \end{pmatrix}. \tag{3.22}$$

All elements of $d^{(-1)}$ are zero since we never need $Ar^{(-1)}$. For $k > 0$, we use Equation (3.19) to define

$$d^{(sk-1)} = \left[0_{1,s-2}, \ \frac{1-\rho^{(sk-1)}}{\rho^{(sk-1)}\gamma^{(sk-1)}}, \ \frac{1}{\rho^{(sk-1)}}, \ -\frac{1}{\rho^{(sk-1)}\gamma^{(sk-1)}}, \ 0_{1,s}\right]^T. \tag{3.23}$$

Similar to Equation (3.13) for CA-CG, we define a matrix $B^{(k)} \in \mathbb{R}^{(s+1)\times(s+1)}$ such that

$$A\underline{W}^{(k)} = W^{(k)}B^{(k)}, \tag{3.24}$$

where $\underline{W}^{(k)}$ is the same as $W^{(k)}$ except the last column is all zeroes. We can define

$$d^{(sk)} = \left[0_{1,s}, B^{(k)}(:,0)^T\right]^T \tag{3.25}$$

with $B^{(k)}(:,0)$ being the first column of $B^{(k)}$.

For $1 \leq j < s$, we multiply the recursive update equation of the residual in Equation (2.29) for $r^{(sk+j)}$ by A and use Equation (3.18).

$$\begin{aligned} Ar^{(sk+j)} &= \rho^{(sk+j-1)}Ar^{(sk+j-1)} - \rho^{(sk+j-1)}\gamma^{(sk+j-1)}A^2r^{(sk+j-1)} \\ &\quad + \left(1-\rho^{(sk+j-1)}\right)Ar^{(sk+j-2)} \end{aligned} \tag{3.26}$$

$$\begin{aligned} \left[R^{(k-1)}, W^{(k)}\right]d^{(sk+j)} &= \rho^{(sk+j-1)}\left[R^{(k-1)}, W^{(k)}\right]d^{(sk+j-1)} \\ &\quad - \rho^{(sk+j-1)}\gamma^{(sk+j-1)}A\left[R^{(k-1)}, W^{(k)}\right]d^{(sk+j-1)} \\ &\quad + \left(1-\rho^{(sk+j-1)}\right)\left[R^{(k-1)}, W^{(k)}\right]d^{(sk+j-2)} \end{aligned} \tag{3.27}$$

To compute $A\left[R^{(k-1)}, W^{(k)}\right]d^{(sk+j-1)}$, we use Equation (3.20) for $AR^{(k-1)}$ and Equation (3.24) for $A\underline{W}^{(k)}$. We can use the latter since, as stated by Hoemmen (2010), the last element of $d^{(sk+j-1)}$ is always zero for $1 \leq j < s$.

To compute the scalar products $\mu^{(sk+j)}$ and $\gamma^{(sk+j)}$, we define auxiliary vectors $g^{(sk+j)}$ of size $2s+1$ that fulfill the relation

$$r^{(sk+j)} = \left[R^{(k-1)}, W^{(k)}\right]g^{(sk+j)}. \tag{3.28}$$

The residuals $r^{(sk-1)}$ and $r^{(sk)}$ are stored in the last column of $R^{(k-1)}$ and the first column of $W^{(k)}$. Therefore, $g^{(sk-1)}$ and $g^{(sk)}$ are standard basis vectors. For $1 \leq j < s$, we use Equations (2.29), (3.18) and (3.28).

$$\begin{aligned} r^{(sk+j)} &= \rho^{(sk+j-1)}r^{(sk+j-1)} - \rho^{(sk+j-1)}\gamma^{(sk+j-1)}Ar^{(sk+j-1)} \\ &\quad + \left(1-\rho^{(sk+j-1)}\right)r^{(sk+j-2)} \end{aligned} \tag{3.29}$$

$$\begin{aligned} \left[R^{(k-1)}, W^{(k)}\right]g^{(sk+j)} &= \rho^{(sk+j-1)}\left[R^{(k-1)}, W^{(k)}\right]g^{(sk+j-1)} \\ &\quad - \rho^{(sk+j-1)}\gamma^{(sk+j-1)}\left[R^{(k-1)}, W^{(k)}\right]d^{(sk+j-1)} \\ &\quad + \left(1-\rho^{(sk+j-1)}\right)\left[R^{(k-1)}, W^{(k)}\right]g^{(sk+j-2)} \end{aligned} \tag{3.30}$$

The Gram matrix $G^{(k)} = \left[R^{(k-1)}, W^{(k)}\right]^T\left[R^{(k-1)}, W^{(k)}\right]$ has $(2s+1)\cdot(2s+1)$ elements. However, only $(2s+1)\cdot(s+1)$ have to be computed since $G^{(k)}$ is symmetric and ${R^{(k-1)}}^T R^{(k-1)}$ is a diagonal matrix due to the orthogonality of the residual vectors. The diagonal entries of ${R^{(k-1)}}^T R^{(k-1)}$ are computed in the previous outer iteration as the scalars $\mu^{(sk-s)}, \dots, \mu^{(sk-1)}$. Using the Gram matrix, the scalars

$$\mu^{(sk+j)} = {r^{(sk+j)}}^T r^{(sk+j)} = {g^{(sk+j)}}^T G^{(k)} g^{(sk+j)}, \tag{3.31}$$

$$\nu^{(sk+j)} = {w^{(sk+j)}}^T r^{(sk+j)} = {g^{(sk+j)}}^T G^{(k)} d^{(sk+j)} \tag{3.32}$$

for $0 \leq j < s$ are computed in the inner iterations without communication.

3.4. Communication-avoiding PCG3 (CA-PCG3)

The preconditioned version of CA-CG3 presented in Hoemmen (2010) is denoted as CA-PCG3 in this thesis. In each outer iteration k of CA-PCG3, two matrices $W^{(k)}$ and $V^{(k)} = PW^{(k)}$ are computed, where

$$\operatorname{span}\left(W^{(k)}\right) = \mathcal{K}_{s+1}\left(AP, r^{(sk)}\right), \tag{3.33}$$

$$\operatorname{span}\left(V^{(k)}\right) = \mathcal{K}_{s+1}\left(PA, u^{(sk)}\right). \tag{3.34}$$

The Matrix Powers Kernel can be invoked twice, once for $W^{(k)}$ and once for $V^{(k)}$. Alternatively, since $V^{(k)} = PW^{(k)}$, the computation of $V^{(k)}$ can be implicitly included in the calculation of $W^{(k)}$.

In addition to the residual matrix $R^{(k)}$, its preconditioned version is also required in CA-PCG3.

$$U^{(k)} = PR^{(k)} = \left[u^{(sk)}, \dots, u^{(sk+s-1)}\right] \tag{3.35}$$

To compute the matrix-vector products $Au^{(sk+j)}$ for $0 \leq j < s$ without communication, we define auxiliary vectors $d^{(sk+j)}$ such that

$$Au^{(sk+j)} = \left[R^{(k-1)}, W^{(k)}\right] d^{(sk+j)}, \tag{3.36}$$

$$PAu^{(sk+j)} = \left[U^{(k-1)}, V^{(k)}\right] d^{(sk+j)}. \tag{3.37}$$

Analogously to Equation (3.20) for CA-CG3, we can rearrange the update recursion of the unpreconditioned residual of PCG3 in Equation (2.40) and summarize for all vectors $Au^{(sk+j)}$ with $0 \leq j < s$ as

$$AU^{(k)} = \frac{1-\rho^{(sk)}}{\rho^{(sk)}\gamma^{(sk)}} r^{(sk-1)} e_1^T + R^{(k)} T^{(k)} - \frac{1}{\rho^{(sk+s-1)}\gamma^{(sk+s-1)}} r^{(sk+s)} e_s^T. \tag{3.38}$$

For $1 \leq j < s$, Equation (3.27) changes to

$$\begin{aligned}\left[R^{(k-1)}, W^{(k)}\right] d^{(sk+j)} = & \rho^{(sk+j-1)} \left[R^{(k-1)}, W^{(k)}\right] d^{(sk+j-1)} \\ & - \rho^{(sk+j-1)}\gamma^{(sk+j-1)} A \left[U^{(k-1)}, V^{(k)}\right] d^{(sk+j-1)} \\ & + \left(1-\rho^{(sk+j-1)}\right) \left[R^{(k-1)}, W^{(k)}\right] d^{(sk+j-2)}.\end{aligned} \tag{3.39}$$

By rearranging the update recursion of the preconditioned residual of PCG3 in Equation (2.42), similar equations as (3.38) and (3.39) can be formulated for $PAU^{(k)}$ and $\left[U^{(k-1)}, V^{(k)}\right] d^{(sk+j)}$.

Analogously to Equation (3.24) for CA-CG3,

$$A\underline{V}^{(k)} = W^{(k)} B^{(k)}, \tag{3.40}$$

$$PA\underline{V}^{(k)} = V^{(k)} B^{(k)}, \tag{3.41}$$

where $B^{(k)} \in \mathbb{R}^{(s+1)\times(s+1)}$ is defined as in CA-CG3 and $\underline{V}^{(k)}$ is the same as $V^{(k)}$ except the last column is all zeroes.

The computation of the vectors $d^{(sk+j)}$ in CA-CG3 and CA-PCG3 are therefore identical. Similarly, we can compute $g^{(sk+j)}$ in CA-PCG3 the same way as in CA-CG3.

$$r^{(sk+j)} = \left[R^{(k-1)}, W^{(k)}\right] g^{(sk+j)} \tag{3.42}$$

$$u^{(sk+j)} = \left[U^{(k-1)}, V^{(k)}\right] g^{(sk+j)} \tag{3.43}$$

To compute the scalars, we compute the Gram matrix as

$$G^{(k)} = \left[R^{(k-1)}, W^{(k)}\right]^T P \left[R^{(k-1)}, W^{(k)}\right] = \left[U^{(k-1)}, V^{(k)}\right]^T \left[R^{(k-1)}, W^{(k)}\right]. \tag{3.44}$$

Analogously to the diagonal matrix ${R^{(k-1)}}^T R^{(k-1)}$ in CA-CG3, the diagonal entries of the upper left part ${U^{(k-1)}}^T R^{(k-1)}$ of $G^{(k)}$ are computed in the previous outer iteration as the scalars $\mu^{(sk-s)}, \ldots, \mu^{(sk-1)}$.

The Communication-avoiding Preconditioned Conjugate Gradient algorithm based on PCG3 (CA-PCG3) is outlined in Algorithm 4.

3.5. Communication-avoiding PCG (CA-PCG)

Preconditioned versions of communication-avoiding Krylov subspace methods and in particular CG and BiCG methods are mentioned e.g. in Toledo (1995), Carson et al. (2013) and Carson (2015). The preconditioned version of the Matrix Powers Kernel and suitable preconditioners are investigated in Demmel et al. (2007) and Hoemmen (2010). However, to the best of our knowledge, there does not yet exist an explicit formulation of a left-preconditioned version of CA-CG analogously to CA-PCG3.

In this section, we derive a left-preconditioned communication-avoiding method (CA-PCG) based on standard PCG that uses similar concepts as CA-PCG3 regarding preconditioning. We note that the deflation technique for CA-CG presented in Carson et al. (2014) aims to accelerate convergence and can thus also be seen as preconditioning.

In Equations (3.6) and (3.7), search spaces for the next s steps of CA-CG are defined based on the residual and the search direction at the beginning of an outer iteration (Toledo, 1995; Carson, 2018). We adapt these search spaces for the preconditioned version CA-PCG and prove their correctness. For $k \geq 0$, $s \geq 1$ and $0 \leq j \leq s$,

$$u^{(sk+j)}, p^{(sk+j)} \in \mathcal{K}_{s+1}\left(PA, p^{(sk)}\right) + \mathcal{K}_s\left(PA, u^{(sk)}\right), \tag{3.45}$$

$$x^{(sk+j)} - x^{(sk)} \in \mathcal{K}_s\left(PA, p^{(sk)}\right) + \mathcal{K}_{s-1}\left(PA, u^{(sk)}\right). \tag{3.46}$$

Algorithm 4 CA-PCG3 (Communication-avoiding PCG3)

```
Input: SPD system matrix $A \in \mathbb{R}^{n\times n}$, preconditioner $P \in \mathbb{R}^{n\times n}$,
       right-hand side $b \in \mathbb{R}^n$, initial guess $x^{(0)} \in \mathbb{R}^n$, step size $s$
Output: approximate solution $x^{(sk+j)} \in \mathbb{R}^n$ of $Ax = b$
 1: $r^{(-1)} = 0_{n,1}, u^{(-1)} = 0_{n,1}, x^{(-1)} = 0_{n,1}$
 2: $r^{(0)} = b - Ax^{(0)}, u^{(0)} = Pr^{(0)}$
 3: for $k = 0, 1, ...$ until convergence do
 4:     Compute $W^{(k)}$ such that span $\left(W^{(k)}\right) = \mathcal{K}_{s+1}\left(AP, r^{(sk)}\right)$
 5:     Compute $V^{(k)}$ such that span $\left(V^{(k)}\right) = \mathcal{K}_{s+1}\left(PA, u^{(sk)}\right)$
 6:     Compute $G^{(k)} = \left[U^{(k-1)}, V^{(k)}\right]^T \left[R^{(k-1)}, W^{(k)}\right]$
 7:     for $j = 0, 1, ..., s-1$ do
 8:         Compute $d^{(sk+j)}$ and $g^{(sk+j)}$ such that Eqns. (3.36), (3.37), (3.42), (3.43) hold
 9:         $\mu^{(sk+j)} = g^{(sk+j)^T} G^{(k)} g^{(sk+j)}$          ▷ $\mu^{(sk+j)} = r^{(sk+j)^T} u^{(sk+j)}$
10:         $\nu^{(sk+j)} = g^{(sk+j)^T} G^{(k)} d^{(sk+j)}$          ▷ $\nu^{(sk+j)} = w^{(sk+j)^T} u^{(sk+j)}$
11:         $\gamma^{(sk+j)} = \mu^{(sk+j)} / \nu^{(sk+j)}$
12:         $w^{(sk+j)} = \left[W^{(k)}, R^{(k-1)}\right] d^{(sk+j)}$          ▷ $w^{(sk+j)} = Au^{(sk+j)}$
13:         $v^{(sk+j)} = \left[V^{(k)}, U^{(k-1)}\right] d^{(sk+j)}$          ▷ $v^{(sk+j)} = PAu^{(sk+j)}$
14:         if $sk + j = 0$ then
15:             $\rho^{(sk+j)} = 1$
16:         else
17:             $\rho^{(sk+j)} = \left(1 - \frac{\gamma^{(sk+j)}}{\gamma^{(sk+j-1)}} \cdot \frac{\mu^{(sk+j)}}{\mu^{(sk+j-1)}} \cdot \frac{1}{\rho^{(sk+j-1)}}\right)^{-1}$
18:         end if
19:         $x^{(sk+j+1)} = \rho^{(sk+j)}\left(x^{(sk+j)} + \gamma^{(sk+j)} u^{(sk+j)}\right) + \left(1 - \rho^{(sk+j)}\right) x^{(sk+j-1)}$
20:         $r^{(sk+j+1)} = \rho^{(sk+j)}\left(r^{(sk+j)} - \gamma^{(sk+j)} w^{(sk+j)}\right) + \left(1 - \rho^{(sk+j)}\right) r^{(sk+j-1)}$
21:         $u^{(sk+j+1)} = \rho^{(sk+j)}\left(u^{(sk+j)} - \gamma^{(sk+j)} v^{(sk+j)}\right) + \left(1 - \rho^{(sk+j)}\right) u^{(sk+j-1)}$
22:     end for
23: end for
```

From the properties of PCG, which is equivalent to CA-PCG in exact arithmetic, we know that

$$x^{(sk+j)} = x^{(sk)} + \sum_{l=sk}^{sk+j-1} \alpha^{(l)} p^{(l)}. \tag{3.47}$$

Using Equation (3.45) thus leads to Equation (3.46).

We prove Equation (3.45) by induction. More specifically, it can be proven that

$$p^{(sk+j)} \in \mathcal{K}_{j+1}\left(PA, p^{(sk)}\right) + \mathcal{K}_j\left(PA, u^{(sk)}\right), \tag{3.48}$$

$$u^{(sk+j)} \in \mathcal{K}_j\left(PA, PAp^{(sk)}\right) + \mathcal{K}_j\left(PA, u^{(sk)}\right), \tag{3.49}$$

$$r^{(sk+j)} \in \mathcal{K}_j\left(AP, Ap^{(sk)}\right) + \mathcal{K}_j\left(AP, r^{(sk)}\right), \tag{3.50}$$

which proves Equation (3.45) since the spaces in Equations (3.48) and (3.49) are subspaces of the space in Equation (3.45). The spaces defined in Equations (3.48) and (3.49) are almost identical, except that the first Krylov subspace in Equation (3.49) does not contain $p^{(sk)}$. Equation (3.50) immediately follows from Equation (3.49) since $u^{(sk+j)} = Pr^{(sk+j)}$.

The case $j = 0$ is trivial. Assuming that Equations (3.48) and (3.49) hold for an arbitrary value j, we prove that they hold for $j + 1$.

Multiplying the update recursion of the unpreconditioned residual of PCG in Equation (2.5) with P on both sides provides a recursive update equation for the preconditioned residual. For $u^{(sk+j+1)}$ we have

$$u^{(sk+j+1)} = u^{(sk+j)} - \alpha^{(sk+j)} PAp^{(sk+j)}. \tag{3.51}$$

Thus, $u^{(sk+j+1)}$ is a linear combination of $u^{(sk+j)}$ and $PAp^{(sk+j)}$. Using Equation (3.48), we get

$$PAp^{(sk+j)} \in \mathcal{K}_{j+1}\left(PA, PAp^{(sk)}\right) + \mathcal{K}_j\left(PA, PAu^{(sk)}\right). \tag{3.52}$$

Combining Equations (3.49) and (3.52), we get

$$u^{(sk+j+1)} \in \mathcal{K}_{j+1}\left(PA, PAp^{(sk)}\right) + \mathcal{K}_{j+1}\left(PA, u^{(sk)}\right). \tag{3.53}$$

From the update recursion of the search direction of PCG in Equation (2.26), we know that $p^{(sk+j+1)}$ is a linear combination of $u^{(sk+j+1)}$ and $p^{(sk+j)}$. Thus, we combine Equations (3.48) and (3.53).

$$p^{(sk+j+1)} \in \mathcal{K}_{j+2}\left(PA, p^{(sk)}\right) + \mathcal{K}_{j+1}\left(PA, u^{(sk)}\right) \tag{3.54}$$

We can therefore write for $k \geq 0$, $s \geq 1$ and $0 \leq j \leq s$

$$p^{(sk+j)} \in \mathcal{K}_{s+1}\left(PA, p^{(sk)}\right) + \mathcal{K}_s\left(PA, u^{(sk)}\right), \tag{3.55}$$

$$u^{(sk+j)} \in \mathcal{K}_s\left(PA, PAp^{(sk)}\right) + \mathcal{K}_s\left(PA, u^{(sk)}\right), \tag{3.56}$$

$$r^{(sk+j)} \in \mathcal{K}_s\left(AP, Ap^{(sk)}\right) + \mathcal{K}_s\left(AP, r^{(sk)}\right), \tag{3.57}$$

$$x^{(sk+j)} - x^{(sk)} \in \mathcal{K}_s\left(PA, p^{(sk)}\right) + \mathcal{K}_{s-1}\left(PA, u^{(sk)}\right). \tag{3.58}$$

In each outer iteration k of CA-PCG, four basis matrices are formed that span these spaces. We define the basis matrix $P^{(k)}$ based on the search direction, its unpreconditioned form $Q^{(k)}$, the basis matrix $R^{(k)}$ based on the residual and its preconditioned version $U^{(k)}$.

$$\operatorname{span}\left(Q^{(k)}\right) = \mathcal{K}_{s+1}\left(AP, Mp^{(sk)}\right) \tag{3.59}$$

$$\operatorname{span}\left(P^{(k)}\right) = \mathcal{K}_{s+1}\left(PA, p^{(sk)}\right) \tag{3.60}$$

$$\operatorname{span}\left(R^{(k)}\right) = \mathcal{K}_s\left(AP, r^{(sk)}\right) \tag{3.61}$$

$$\operatorname{span}\left(U^{(k)}\right) = \mathcal{K}_s\left(PA, u^{(sk)}\right) \tag{3.62}$$

Note that $R^{(k)}$ and $U^{(k)}$ differ from the matrices in CA-CG3 and CA-PCG3, where $R^{(k)}$ and $U^{(k)}$ denote the residual matrices, whose columns contain the already orthogonalized residuals of outer iteration k. In CA-PCG, $R^{(k)}$ and $U^{(k)}$ denote basis matrices computed by the Matrix Powers Kernel.

To compute $Q^{(k)}$, $P^{(k)}$, $R^{(k)}$ and $U^{(k)}$, the Matrix Powers Kernel is invoked four times in outer iteration k. Alternatively, since $P^{(k)} = PQ^{(k)}$ and $U^{(k)} = PR^{(k)}$, the computation of $P^{(k)}$ and $U^{(k)}$ can be implicitly included in the calculation of $Q^{(k)}$ and $R^{(k)}$.

We define the s-step basis matrix $Y^{(k)} = \left[Q^{(k)}, R^{(k)}\right]$ and its preconditioned version $Z^{(k)} = PY^{(k)} = \left[P^{(k)}, U^{(k)}\right]$. Analogously to CA-CG, we define small vectors $p^{(k,j)'}$, $r^{(k,j)'}$ and $x^{(k,j)'}$ for $0 \leq j \leq s$ in a different basis to express the vectors $p^{(sk+j)}$, $r^{(sk+j)}$, $u^{(sk+j)}$ and $x^{(sk+j)} - x^{(sk)}$.

$$p^{(sk+j)} = Z^{(k)} p^{(k,j)'} \tag{3.63}$$

$$r^{(sk+j)} = Y^{(k)} r^{(k,j)'} \tag{3.64}$$

$$u^{(sk+j)} = Z^{(k)} r^{(k,j)'} \tag{3.65}$$

$$x^{(sk+j)} - x^{(sk)} = Z^{(k)} x^{(k,j)'} \tag{3.66}$$

The matrix-vector products $Ap^{(sk+j)}$ are obtained in a similar way as in CA-CG. We define a matrix $B^{(k)} \in \mathbb{R}^{(s+1)\times(s+1)}$ such that

$$A\underline{Z}^{(k)} = Y^{(k)} B^{(k)}, \tag{3.67}$$

$$PA\underline{Z}^{(k)} = Z^{(k)} B^{(k)}, \tag{3.68}$$

with $\underline{Z}^{(k)} = \left[\underline{P}^{(k)}, \underline{U}^{(k)}\right]$, where $\underline{P}^{(k)}$ and $\underline{U}^{(k)}$ are the same as $P^{(k)}$ and $U^{(k)}$ except their respective last column is all zeroes. We can therefore compute $B^{(k)} p^{(k,j)'}$ for the matrix-vector products in the changed basis in the inner iterations.

The scalar products are computed with the symmetric Gram matrix

$$G^{(k)} = Y^{(k)} P Y^{(k)} = Y^{(k)^T} Z^{(k)} = Z^{(k)^T} Y^{(k)}. \tag{3.69}$$

The scalar product $p^{(sk+j)^T} A p^{(sk+j)}$ is computed using Equation (3.67).

$$r^{(sk+j)^T} u^{(sk+j)} = r^{(k,j)'^T} Y^{(k)^T} Z^{(k)} r^{(k,j)'} = r^{(k,j)'^T} G^{(k)} r^{(k,j)'} \tag{3.70}$$

$$\begin{aligned} p^{(sk+j)^T} A p^{(sk+j)} &= p^{(k,j)'^T} Z^{(k)^T} A Z^{(k)} p^{(k,j)'} \\ &= p^{(k,j)'^T} Z^{(k)^T} Y^{(k)} B^{(k)} p^{(k,j)'} = p^{(k,j)'^T} G^{(k)} B^{(k)} p^{(k,j)'} \end{aligned} \tag{3.71}$$

3.5.1. CA-PCG with the Monomial Basis

Forming the basis matrix $Q^{(k)}$ requires the computation of $Mp^{(sk)}$, see Equation (3.59). However, in practice we cannot rely on the availability of M as often only the preconditioner $P = M^{-1}$ is given. In this section, we discuss why we do not need to compute $Mp^{(sk)}$ explicitly when using the monomial basis. In this case, the first column of $Q^{(k)}$ can be set to zero.

The unpreconditioned s-step basis matrix $Y^{(k)}$ is used in two steps of CA-PCG in each outer iteration k, the computation of the Gram matrix $G^{(k)}$ and the recovery of the iterates at the end of each outer iteration using Equations (3.63) to (3.66). More specifically, only the unpreconditioned residual $r^{(sk+s)}$ is recovered using $Y^{(k)}$ in Equation (3.64) while $u^{(sk+s)}$, $p^{(sk+s)}$ and $x^{(sk+s)}$ are recovered using $Z^{(k)}$.

From Equation (3.57) we know that $r^{(sk+j)}$ does not depend on $Mp^{(sk)}$. Since the monomial basis is formed exactly by the vectors defined by the Krylov subspaces, the first element of $r^{(k,j)'}$ is always zero. The residual vector in the changed basis is updated with

$$r^{(k,j+1)'} = r^{(k,j)'} - \alpha^{(sk+j)} B^{(k)} p^{(k,j)'}. \tag{3.72}$$

The first element of $r^{(k,0)'}$ is zero by definition. When using the monomial basis,

$$B^{(k)} = \left[e_2, e_3, ..., e_{s+1}, 0_{2s+1,1}, e_{s+3}, ..., e_{2s+1}, 0_{2s+1,1}\right], \tag{3.73}$$

where e_j represents the j^{th} column of the $(2s+1) \times (2s+1)$ identity matrix. The first row of $B^{(k)}$ consists only of zeroes. Therefore, the first element of $B^{(k)} p^{(k,j)'}$ is always zero.

The matrix $G^{(k)}$ is symmetric. Thus, if we compute $G^{(k)} = {Y^{(k)}}^T Z^{(k)}$, we do not need to use the first column of $Y^{(k)}$ to compute the first row of $G^{(k)}$ since we can compute the first column without $Mp^{(sk)}$ except for the upper left element of $G^{(k)}$.

The upper left element of $G^{(k)}$ is ${p^{(sk)}}^T Mp^{(sk)}$. Since we showed that the first element of $r^{(k,j+1)'}$ is zero, we do not need ${p^{(sk)}}^T Mp^{(sk)}$ to compute ${r^{(k,j)'}}^T G^{(k)} r^{(k,j)'}$. We cannot make such an assumption for the scalar product ${p^{(k,j)'}}^T G^{(k)} B^{(k)} p^{(k,j)'}$ since the first element of $p^{(k,0)'}$ is not zero.

However, the matrix $B^{(k)}$ defined in Equation (3.73) does not contain e_1, the first column of the $(2s+1) \times (2s+1)$ identity matrix. Thus, $G^{(k)} B^{(k)}$ does not contain the values of the first column of $G^{(k)}$. Moreover, we showed above that the first element of $B^{(k)} p^{(k,j)'}$ is always zero. Thus, the upper left element of $G^{(k)}$ is not used when computing ${p^{(k,j)'}}^T G^{(k)} B^{(k)} p^{(k,j)'}$.

The upper left element of $G^{(k)}$ can be set to zero to avoid using M. Alternatively, it can be set to the scalar product ${r^{(sk)}}^T r^{(sk)}$ to use as a convergence criterion without additional communication latency overhead.

3.5.2. CA-PCG with other basis types

When using a basis other than the monomial basis, we cannot assume that the explicit computation of $Mp^{(sk)}$ is not necessary. If the basis uses shifts, as with the Newton basis, $P^{(k)}$ and $Q^{(k)}$ are computed as

$$P^{(k)} = \left[p^{(sk)}, \left(PA - \theta_0 I\right) p^{(sk)}, \left(PA - \theta_1 I\right)\left(PA - \theta_0 I\right) p^{(sk)}, \dots\right], \tag{3.74}$$

$$Q^{(k)} = \left[Mp^{(sk)}, \left(AP - \theta_0 I\right) Mp^{(sk)}, \left(AP - \theta_1 I\right)\left(AP - \theta_0 I\right) Mp^{(sk)}, \dots\right], \tag{3.75}$$

with shifts $\theta_0, \theta_1, \dots \theta_{s-1}$. We do not need the first column of $Q^{(k)}$, but $Mp^{(sk)}$ is required for the computation of the other columns of this basis matrix.

We therefore introduce the unpreconditioned search direction $q^{(sk)} = Mp^{(sk)}$. In the first outer iteration, since $p^{(0)} = u^{(0)}$, $q^{(0)} = r^{(0)}$. The small vector $r^{(k,j)'}$ in the changed basis is used both for the recovery of the unpreconditioned and the preconditioned residual $r^{(sk+j)}$ and $u^{(sk+j)}$ in Equations (3.64) and (3.65). Similarly, we can recover $q^{(sk+j)}$ and $p^{(sk+j)}$ with

$$q^{(sk+j)} = Y^{(k)} p^{(k,j)'}, \tag{3.76}$$

$$p^{(sk+j)} = Z^{(k)} p^{(k,j)'}. \tag{3.77}$$

Algorithm 5 CA-PCG (Communication-avoiding PCG) for arbitrary basis types

```
Input: SPD system matrix $A \in \mathbb{R}^{n \times n}$, preconditioner $P \in \mathbb{R}^{n \times n}$,
       right-hand side $b \in \mathbb{R}^n$, initial guess $x^{(0)} \in \mathbb{R}^n$, step size $s$
Output: approximate solution $x^{(sk+j)} \in \mathbb{R}^n$ of $Ax = b$
 1: $r^{(0)} = b - Ax^{(0)}, u^{(0)} = Pr^{(0)}, q^{(0)} = r^{(0)}, p^{(0)} = u^{(0)}$
 2: for $k = 0, 1, ...$ until convergence do
 3:     Compute $Y^{(k)}$ such that span $\left(Y^{(k)}\right) = \mathcal{K}_{s+1}\left(AP, q^{(sk)}\right) + \mathcal{K}_s\left(AP, r^{(sk)}\right)$
 4:     Compute $Z^{(k)}$ such that span $\left(Z^{(k)}\right) = \mathcal{K}_{s+1}\left(PA, p^{(sk)}\right) + \mathcal{K}_s\left(PA, u^{(sk)}\right)$
 5:     Compute $G^{(k)} = {Z^{(k)}}^T Y^{(k)}$
 6:     Assemble $B^{(k)}$ such that $A\underline{Z}^{(k)} = Y^{(k)} B^{(k)}$, see Equation (3.67)
 7:     $p^{(k,0)'} = [1, 0_{1,2s}]^T, r^{(k,0)'} = [0_{1,s+1}, 1, 0_{1,s-1}]^T, x^{(k,0)'} = [0_{1,2s+1}]^T$
 8:     for $j = 0, 1, ..., s-1$ do
 9:         $\alpha^{(sk+j)} = {r^{(k,j)'}}^T G^{(k)} r^{(k,j)'} / {p^{(k,s)'}}^T G^{(k)} B^{(k)} p^{(k,j)'}$
10:         $x^{(k,j+1)'} = x^{(k,j)'} + \alpha^{(sk+j)} p^{(k,j)'}$
11:         $r^{(k,j+1)'} = r^{(k,j)'} - \alpha^{(sk+j)} B^{(k)} p^{(k,j)'}$
12:         $\beta^{(sk+j)} = {r^{(k,j+1)'}}^T G^{(k)} r^{(k,j+1)'} / {r^{(k,j)'}}^T G^{(k)} r^{(k,j)'}$
13:         $p^{(k,j+1)'} = r^{(k,j+1)'} + \beta^{(sk+j)} p^{(k,j)'}$
14:     end for
15:     $q^{(sk+s)} = Y^{(k)} p^{(k,s)'}$
16:     $p^{(sk+s)} = Z^{(k)} p^{(k,s)'}$
17:     $r^{(sk+s)} = Y^{(k)} r^{(k,s)'}$
18:     $u^{(sk+s)} = Z^{(k)} r^{(k,s)'}$
19:     $x^{(sk+s)} = x^{(sk)} + Z^{(k)} x^{(k,s)'}$
20: end for
```

The Communication-avoiding Preconditioned Conjugate Gradient algorithm based on PCG (CA-PCG) for arbitrary basis types is outlined in Algorithm 5.

When using the monomial basis, we do not have to explicitly store the unpreconditioned search direction $q^{(sk)}$. Thus, we do not have to compute the recovery of $q^{(sk+s)}$ with Equation (3.76) at the end of each outer iteration. However, this step is performed on each node locally. Therefore, using a different basis does not result in communication overhead, only in additional local computation.

3.6. Matrix Powers Kernel

Communication-avoiding and communication-hiding methods aim to avoid or hide latency of global communication for the scalar products. Pipelined methods such as PPCG (Ghysels and Vanroose, 2014) leave the communication for the matrix-vector products and for the preconditioner applications unchanged since they assume that only communication with local neighbours is necessary for these operations. In contrast, the Matrix Powers Kernel first presented in Demmel et al. (2007) provides a possibility for communication-avoiding s-step methods to reduce communication latency for the matrix-vector products and preconditioner applications by a factor of $\mathcal{O}(s)$.

Demmel et al. (2007) present different Matrix Powers Kernel algorithms for the monomial basis, which Hoemmen (2010) generalizes for arbitrary basis types as defined in Equations (3.1) and (3.2). In the unpreconditioned case, the Matrix Powers Kernel computes the basis matrix

$$T = [\rho_0(A)v, \rho_1(A)v, \ldots, \rho_s(A)v]\,, \tag{3.78}$$

where $v \in \mathbb{R}^n$, $A \in \mathbb{R}^{n\times n}$. The polynomial $\rho_j(z)$ of degree j satisfies the three-term recurrence defined in Equation (3.2). Since $\rho_0(A) = 1$,

$$T = [v, \rho_1(A)v, \ldots, \rho_s(A)v]\,. \tag{3.79}$$

In a parallel environment with distributed memory, the sparse matrix A, the vector v and the matrix T are distributed among the nodes. For example, A and T can be block-row distributed, and v is distributed accordingly. Other distributions are also possible when using the Matrix Powers Kernel (Hoemmen, 2010).

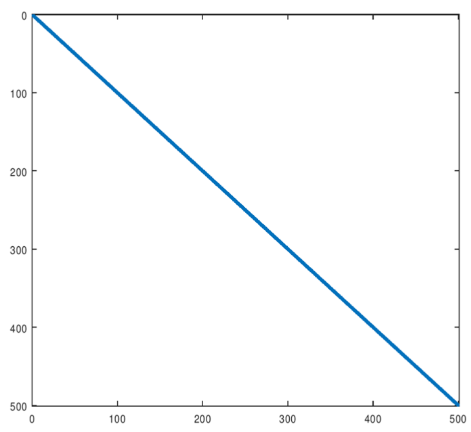
In a non-optimized algorithm, s consecutive matrix-vector products with the corresponding s communication phases for gathering the required parts of the vectors $v, \rho_1(A)v, \ldots, \rho_{s-1}(A)v$ on each node are performed. This method is referred to as the Matrix Powers Kernel Parallel Algorithm 0 (MPK-PA0) (Demmel et al., 2007).

The Matrix Powers Kernel Parallel Algorithm 1 (MPK-PA1) requires only one communication phase for s matrix-vector products. Every node computes parts of T that can be computed locally while communicating all parts of v necessary to compute the rest of T without further communication. After communication of v is finished, every node computes the remaining parts of T. This leads to increased computation since parts of T have to be computed redundantly. If A has a special sparsity structure, e.g. it is a band matrix, the communication phase at the beginning of MPK-PA1 has asymptotically the same latency cost as a single matrix-vector product. Therefore, communication latency is reduced by a factor of $\mathcal{O}(s)$ compared to MPK-PA0 (Demmel et al., 2007).

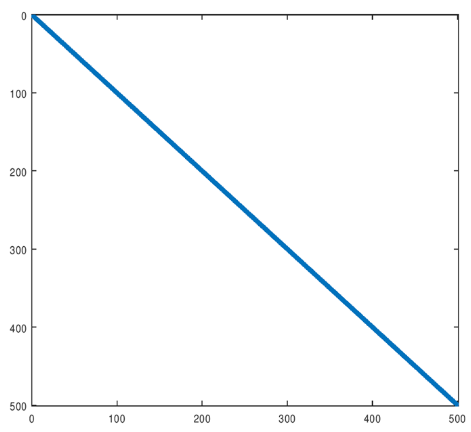
MPK-PA1 can increase local computation considerably. Depending on the value of s and the sparsity structure of A, the nodes potentially have to compute large redundant parts of T. The Matrix Powers Kernel Parallel Algorithm 2 (MPK-PA2) aims to minimize the amount of redundant local computation while simultaneously requiring only one communication phase. Before the communication phase, each node computes locally computable parts of T that are computed redundantly by its neighbours when using MPK-PA1. These parts are then communicated along with v to reduce redundant computation compared to MPK-PA1. Simultaneously, the rest of the locally computable parts of T are computed on each node. When the communication phase is finished, the remaining parts of T are computed (Demmel et al., 2007).

In the sequential case, the matrix might not fit into fast memory and must be read from slow memory s times to compute T in a straight-forward implementation. Demmel et al. (2007) provide sequential algorithms that reduce accesses to slow memory for the case where the matrix or even the columns of T do not fit into fast memory by dividing A and T into row-blocks and emulating MPK-PA1. These algorithms can be combined with the parallel versions of the Matrix Powers Kernel above (Demmel et al., 2007).

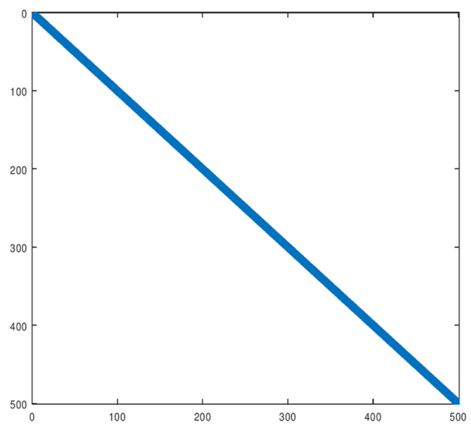
Figures 1 to 3 in Demmel et al. (2007) give an illustrative impression of MPK-PA1 and MPK-PA2 for tridiagonal matrices. Although the algorithms can be used for general sparsity patterns, performance of MPK-PA1 and MPK-PA2 is influenced significantly by the structure of A (Knight et al., 2013).



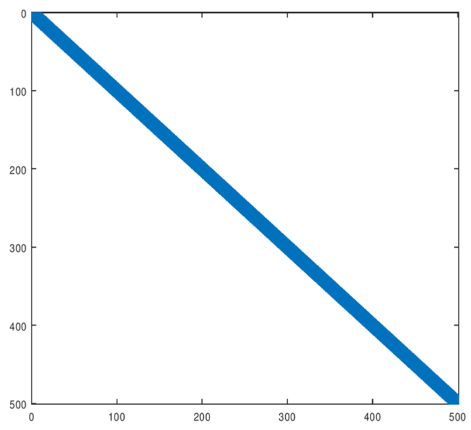
(a) $s = 1$, sparsity pattern of A



(b) $s = 2$, sparsity pattern of A^2



(c) $s = 5$, sparsity pattern of A^5



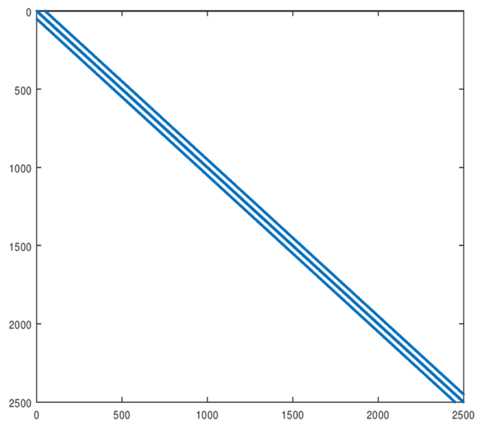
(d) $s = 10$, sparsity pattern of A^{10}

Figure 3.1.: Sparsity patterns of powers A^s of a 500×500 tridiagonal matrix A. On the x-axis, the column index is displayed, on the y-axis the row index.

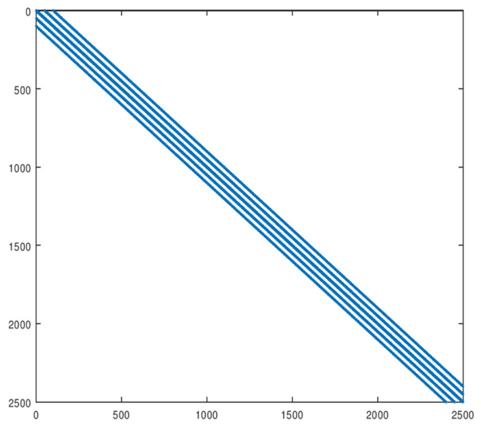
For the s consecutive matrix-vector products in MPK-PA0, communication of the distributed vector parts between nodes is determined by the sparsity pattern of A. When using MPK-PA1, communication of the distributed parts of v is determined by the sparsity pattern of $(A + A^2 + \cdots + A^s)$, ignoring potential cancellations. MPK-PA1 and MPK-PA2 communicate different values but the same amount of data (Demmel et al., 2008).

Consequently, higher values of s potentially require a greater amount of communicated data and of redundant local computations. Due to performance concerns, MPK-PA1 and MPK-PA2 are therefore only suitable for matrices that are well-partitioned (after potential reordering), i.e. for matrices where the amount of communication latency does not increase significantly with s (Demmel et al., 2007; Knight et al., 2013).

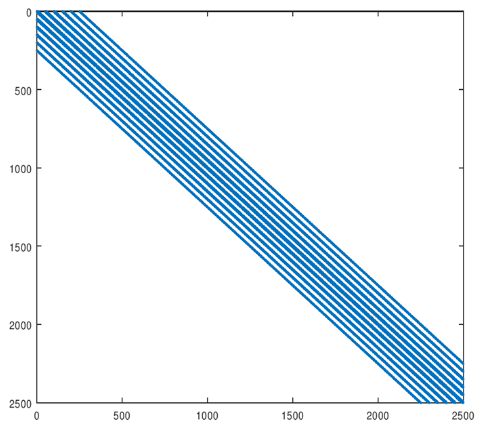
Figures 3.1a to 3.1d show the sparsity patterns of a randomly generated tridiagonal matrix A of size 500×500 and its powers A^s for different values of s. For larger values of s, the bandwidth of A^s gets larger, but is still relatively small for $s = 10$.



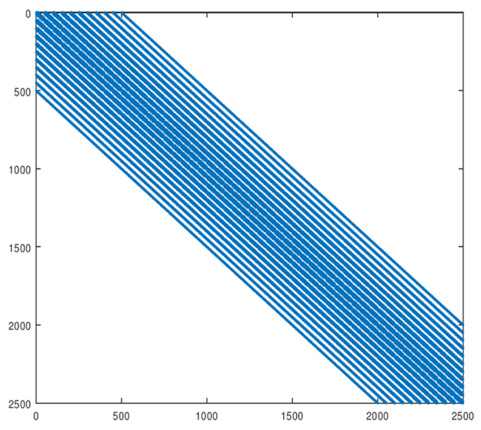
(a) $s = 1$, sparsity pattern of A



(b) $s = 2$, sparsity pattern of A^2



(c) $s = 5$, sparsity pattern of A^5



(d) $s = 10$, sparsity pattern of A^{10}

Figure 3.2.: Sparsity patterns of powers A^s of a 2500×2500 poisson matrix A. On the x-axis, the column index is displayed, on the y-axis the row index.

Figures 3.2a to 3.2d show the sparsity patterns of a 2500×2500 poisson matrix A and its powers A^s for different values of s. This matrix has a larger bandwidth as a tridiagonal matrix, but for the matrix-vector products in MPK-PA0, only communication with local neighbours is required, see Figure 3.2a. However, the bandwidth of A^s with $s = 10$ in Figure 3.2d is approximately 20% of the problem size. Thus, MPK-PA1 does not asymptotically have the same communication latency cost as a single matrix-vector product for this matrix when using $s = 10$, and large parts of T have to be computed redundantly.

Communication between nodes and therefore performance of the Preconditioned Matrix Powers Kernel is influenced by the sparsity pattern of both A and PA (Demmel et al., 2007). While Demmel et al. (2007) extend the Matrix Powers Kernel to a left-preconditioned version for the monomial basis, Hoemmen (2010) shows how to incorporate preconditioning when other basis types are used.

Demmel et al. (2007) show how the Preconditioned Matrix Powers Kernel can be used with simple preconditioners, e.g. the Jacobi preconditioner or the Block Jacobi preconditioner with a small block size. Hoemmen (2010) identifies sets of non-trivial preconditioners, i.e. preconditioners whose application needs communication between nodes, for which communication can be avoided. We will not explore the topic of the Preconditioned Matrix Powers Kernel in detail since we used the Jacobi preconditioner in our experiments, whose application does not require communication between nodes.

3.7. Theoretical comparison of CA-PCG and CA-PCG3

In this section, we theoretically compare CA-PCG and CA-PCG3 in terms of computational and communication costs. Both algorithms form an unpreconditioned and a preconditioned basis matrix at the beginning of each outer iteration. Although all of these basis matrices have $2s+1$ columns, the cost of their computation is not identical for the two algorithms.

To compute the unpreconditioned basis matrix in outer iteration k, CA-PCG invokes the Matrix Powers Kernel twice in line 3 of Algorithm 5 to compute $Y^{(k)} = \left[Q^{(k)}, R^{(k)}\right]$. In contrast, CA-PCG3 invokes the Matrix Powers Kernel only once in outer iteration k in line 4 of Algorithm 4 to compute $W^{(k)}$. The remaining columns of the unpreconditioned basis matrix in CA-PCG3 consist of the last s residual vectors that have already been computed during the previous outer iteration. The preconditioned basis matrices are either computed analogously to the unpreconditioned basis matrices, or implicitly during the computation of the unpreconditioned basis matrices.

CA-PCG requires twice as many Matrix Powers Kernel invocations as CA-PCG3 and almost twice as much data must be communicated to compute the basis matrices. However, the vector parts communicated during the Matrix Powers Kernel can be sent together for both kernel invocations in an outer iteration of CA-PCG. For both algorithms, the computation of the Gram matrix in line 5 of Algorithm 5 and line 6 of Algorithm 4 requires one global reduction operation. Thus, CA-PCG and CA-PCG3 have the same communication latency cost per outer iteration.

CA-PCG requires local computation of almost twice as many matrix-vector multiplications and preconditioner applications as CA-PCG3. Depending on the sparsity structure of the system matrix and the cost for applying the preconditioner, this can significantly impact performance if communication is not the most important factor in performance.

While CA-PCG3 computes the same vectors as PCG3 in the inner iterations, CA-PCG computes the inner iterations in a changed basis with small vectors, and only computes every s^{th} search direction and residual explicitly. At the beginning of an outer CA-PCG iteration, the current residual and search direction must be available. In contrast, the current residual as well as the previous s residuals must be available at the beginning of an outer CA-PCG3 iteration, leading to a more expensive recovery in case of node failures, which we describe in Chapter 5.

4. Resilient Conjugate Gradient methods

In a parallel environment, different types of failures can occur. Soft errors are spontaneous changes of the state of a solver, caused for example by bit flips. An algorithm can be made tolerant against such failures by computing checksums, see for example Shantharam et al. (2012), Zhang and He (2013) and Schöll et al. (2015).

Another type of failure that can occur in a parallel environment is the failure of a node, e.g. it crashes or looses its connection to the network. This kind of failure causes an interruption of the solver and the data owned by the failed node is lost (Agullo et al., 2013; Pachajoa et al., 2019). This thesis addresses resilience against node failures.

A general approach to make an algorithm tolerant against node failures is Checkpoint-Restart. The current state of the algorithm, i.e. the data needed to continue the algorithm from this checkpoint, is saved regularly. This data is saved in reliable external storage or replicated on other nodes such that it can be retrieved if a node fails (Plank et al., 1997; Herault and Robert, 2015).

Checkpoint-Restart can be applied to a wide range of algorithms. However, saving the complete state necessary to restart from a checkpoint can lead to large overhead, depending on how frequently the checkpointing is done, and has to be done even if no node failure occurs. In case of a node failure, the algorithm has to rollback to the last checkpoint and recalculate all iterations computed since the checkpoint, which increases the overhead (Herault and Robert, 2015; Chen, 2011; Cappello et al., 2014).

Section 4.1 describes an algorithm-based approach presented in Chen (2011), which ensures resilience against a single node failure without checkpointing. In Section 4.2, an extension of this approach provided in Pachajoa et al. (2018) to tolerate multiple simultaneous or overlapping node failures is described.

We assume that the runtime environment provides fault-tolerance features as the MPI extension User Level Failure Mitigation (ULFM) (Message Passing Interface Forum, 2021b; Bland et al., 2013). This extension is able to detect and notify about node failures and includes a mechanism for providing replacement nodes, which reconstruct the lost data of the failed nodes and are either a spare node or one of the surviving nodes. For more details, see Section 1.3.

4.1. Algorithm-based fault-tolerant PCG

We assume that A is block-row distributed and vectors are distributed accordingly. Thus, the part of the residual $r^{(i)} = b - Ax^{(i)}$ owned by node f can be written as

$$r_{I_f}^{(i)} = b_{I_f} - \sum_p A_{I_f,I_p} x_{I_p}^{(i)} \tag{4.1}$$

with the sum iterating over all nodes p. Extracting the part of the matrix-vector product involving $x_{I_f}^{(i)}$, i.e. the part of the solution vector owned by node f, leads to

$$A_{I_f,I_f} x_{I_f}^{(i)} = b - r_{I_f}^{(i)} - \sum_{p \neq f} A_{I_f,I_p} x_{I_p}^{(i)}. \tag{4.2}$$

If a node failure of f occurs, the vector parts $r_{I_f}^{(i)}$ and $x_{I_f}^{(i)}$ are lost. At convergence, $x^{(i)} = x$, where x is the exact solution. Thus, all elements of the residual vector are zero (Langou et al., 2007).

Langou et al. (2007) present an Interpolation Restart (IR) method for Krylov subspace methods. It is assumed that the residual is already small and the part $r_{I_f}^{(i)}$ of the residual owned by the failed node is set to zero in Equation (4.2). The lost part of the solution vector is interpolated by solving the system of linear equations

$$A_{I_f,I_f} x_{I_f}^{(approx)} = b - \sum_{p \neq f} A_{I_f,I_p} x_{I_p}^{(i)} \tag{4.3}$$

for the approximation $x_{I_f}^{(approx)}$ of $x_{I_f}^{(i)}$. The Krylov subspace method is restarted with this interpolation $x_{I_f}^{(approx)}$ and with $x_{I \setminus I_f}^{(i)}$, the parts of $x^{(i)}$ owned by the surviving nodes, as the initial guess.

Computing $x_{I_f}^{(approx)}$ only requires the cost of a matrix-vector product as done in each iteration of a Krylov subspace method and solving a small system of equations with A_{I_f,I_f} as the system matrix, which can be done on the replacement node locally. To be able to solve the linear system in Equation (4.3), the matrix A_{I_f,I_f} must be non-singular, which is guaranteed if A is SPD (Langou et al., 2007). Conjugate Gradient methods can only be used with SPD matrices. For Krylov subspace methods that are suitable for general matrices, the IR method is extended in Agullo et al. (2013) and Agullo et al. (2016).

4.1.1. Exact State Reconstruction (ESR)

The Exact State Reconstruction (ESR) method presented in Chen (2011) reconstructs the state of an iteration of PCG exactly and can be seen as a refinement of IR (Pachajoa et al., 2018). The IR method computes an interpolation using Equation (4.3) since it is assumed that the residual is small and its lost part is therefore set to zero. With ESR, the state of the current iteration of PCG is reconstructed exactly, including the lost part of the residual. The lost part of the solution vector can therefore be reconstructed exactly by solving the linear system in Equation (4.2). PCG does not have to be restarted and can be continued after reconstruction (Chen, 2011).

Chen (2011) uses the properties of PCG to save search directions redundantly with low communication overhead. With these search directions, the rest of the algorithm's state can be reconstructed. In the remainder of this thesis, we will refer to this algorithm-based approach, as well as to its extension for tolerating multiple node failures (see Levonyak et al. (2020) and Section 4.2), as resilient PCG.

Using ESR, the state of iteration i is reconstructed with the previous and current search directions $p^{(i-1)}$ and $p^{(i)}$ and the scalar $\beta^{(i-1)}$. The scalar $\beta^{(i-1)}$ has the same value on every node and can be obtained from any surviving node. Since $p^{(i+1)}$ is computed with

Algorithm 6 Exact State Reconstruction (ESR) for PCG

```
Input: part of surviving nodes' state in iteration $i$ of resilient PCG:
       $r^{(i)}_{I\setminus I_f}$, $x^{(i)}_{I\setminus I_f}$, $\beta^{(i-1)}$
       redundant vector copies for failed node $f$:
       $p^{(i)}_{I_f}$, $p^{(i-1)}_{I_f}$
Output: reconstructed state of failed/replacement node $f$:
       $A_{I_f,I}$, $P_{I_f,I}$, $b_{I_f}$, $r^{(i)}_{I_f}$, $u^{(i)}_{I_f}$, $p^{(i)}_{I_f}$, $x^{(i)}_{I_f}$
1: Retrieve static data $A_{I_f,I}$, $P_{I_f,I}$ and $b_{I_f}$
2: Gather $r^{(i)}_{I\setminus I_f}$ and $x^{(i)}_{I\setminus I_f}$
3: Retrieve $\beta^{(i-1)}$, $p^{(i)}_{I_f}$ and $p^{(i-1)}_{I_f}$
4: Compute $u^{(i)}_{I_f} = p^{(i)}_{I_f} - \beta^{(i-1)} p^{(i-1)}_{I_f}$
5: Compute $v = u^{(i)}_{I_f} - P_{I_f,I\setminus I_f} r^{(i)}_{I\setminus I_f}$
6: Solve $P_{I_f,I_f} r^{(i)}_{I_f} = v$ for $r^{(i)}_{I_f}$
7: Compute $w = b_{I_f} - r^{(i)}_{I_f} - A_{I_f,I\setminus I_f} x^{(i)}_{I\setminus I_f}$
8: Solve $A_{I_f,I_f} x^{(i)}_{I_f} = w$ for $x^{(i)}_{I_f}$
```

$p^{(i+1)} = u^{(i+1)} + \beta^{(i)} p^{(i)}$ in iteration i of PCG, the lost part of the preconditioned residual $u^{(i)}$ can be reconstructed using the relation $p^{(i)} = u^{(i)} + \beta^{(i-1)} p^{(i-1)}$.

Subsequently, since $u^{(i)} = Pr^{(i)}$, the lost part of the unpreconditioned residual $r^{(i)}_{I_f}$ is reconstructed by solving the small linear system

$$P_{I_f,I_f} r^{(i)}_{I_f} = u^{(i)}_{I_f} - \sum_{p \neq f} P_{I_f,I_p} r^{(i)}_{I_p} \tag{4.4}$$

locally on the replacement node. The lost part of the approximate solution $x^{(i)}$ is reconstructed by solving the small linear system in Equation (4.2). To compute the right-hand sides of Equations (4.2) and (4.4), all necessary parts of $r^{(i)}_{I\setminus I_f}$ and $x^{(i)}_{I\setminus I_f}$ are gathered from the surviving nodes on the replacement node.

The system matrix A along with the right hand side b and the preconditioner P are considered as static data. If the matrix is constructed before starting PCG, it can be reconstructed in case of a node failure. If a checkpoint is done before starting PCG, the lost parts of the static data can be retrieved from reliable external storage (Chen, 2011; Pachajoa et al., 2018).

Since A and P are SPD when using Conjugate Gradient methods, the small system matrices A_{I_f,I_f} and P_{I_f,I_f} in Equations (4.2) and (4.4) are SPD as well. Thus, after computing the right-hand sides, these small linear systems can be solved with PCG on the replacement node (Levonyak et al., 2020).

The Exact State Reconstruction (ESR) for PCG is outlined in Algorithm 6.

4.1.2. Data redundancy in resilient PCG

Chen (2011) provides a mechanism that ensures enough data redundancy to tolerate a single node failure. Since A is assumed to be block-row distributed and the vectors are distributed accordingly, every node p holds a local part of the search direction $p^{(i)}$, which has to be stored redundantly on a different node q in case node p fails. Additionally, p's part of the previous search direction $p^{(i-1)}$ has to be stored redundantly on q.

The search direction $p^{(i)}$ is communicated for the matrix-vector product $Ap^{(i)}$ in PCG. If A was dense, the whole vector $p^{(i)}$ would be gathered on each node before computing the matrix-vector product. In non-resilient PCG, the gathered parts of $p^{(i)}$ are discarded after calculating the product. In resilient PCG, some of these parts could be kept and therefore no additional communication for distributing the redundantly stored parts of $p^{(i)}$ would be necessary. Since A is sparse, only a few vector parts have to be received by each node to compute its local part of $Ap^{(i)}$, depending on the sparsity structure.

If elements of $p^{(i)}$ are not communicated for the matrix-vector product, they have to be sent additionally. To decrease latency overhead, these elements can be sent to nodes which are already receiving vector parts from this node. Chen (2011) suggests that a node p sends such vector elements to the neighbour node $(p+1) \bmod N$, where N is the total number of nodes.

4.1.3. Comparison of resilient PCG with other strategies

PCG using Checkpoint-Restart and resilient PCG using ESR (Chen, 2011) both have no computation overhead in the failure-free case. In contrast to PCG using Checkpoint-Restart, resilient PCG (Chen, 2011) has low communication and storage overhead if no node failure occurs and low overhead for reconstruction in case of a node failure. While the IR method has no overhead in the failure-free execution, convergence of the Krylov subspace method may be slowed down after a node failure compared to the failure-free execution since the lost part of the solution vector is only approximated (Chen, 2011; Pachajoa et al., 2018).

Chen (2011) compares resilient PCG with Parallel-I/O-Based and Diskless Checkpointing experimentally and shows that the algorithm-based fault-tolerance has significant better runtime performance than Checkpoint-Restart for both cases where no node failure or a periodic simulated node failure occurs.

Pachajoa et al. (2018) compare resilient PCG with the IR method for PCG experimentally. In case of a node failure, the overhead of IR is much larger than the cost of ESR for the test matrices used in Pachajoa et al. (2018). Since IR only computes an interpolation of the lost part of the solution vector, the algorithm potentially requires a much larger number of iterations to converge after a failure (Pachajoa et al., 2018).

4.2. Extension to multiple node failures

Pachajoa et al. (2019) provide an extension of the resilient PCG method described in Section 4.1 to tolerate multiple simultaneous or overlapping node failures.

If node $p+1$ stores parts of the redundant copy of node p's local vector, and both p and $p+1$ fail, the redundant copy of p's local vector is lost. To extend the algorithm to be resilient against multiple node failures, there have to be at least $\phi+1$ copies of all vector elements to be able to reconstruct data of ϕ failed nodes.

As described in Section 4.1.2, the neighbour nodes that receive data from a specific node p for the matrix-vector product are determined by the sparsity pattern of A. Some vector elements are probably already sent to several neighbour nodes. If a vector element is already sent to k neighbour nodes, it only has to be sent to $\max(0, \phi - k)$ additional nodes to make the algorithm tolerant against ϕ simultaneous or overlapping node failures.

At most ϕ copies of each vector element have to be communicated additionally. For the copies $k = 1, 2, ..., \phi$, Pachajoa et al. (2019) suggest to send them to the nodes $(p + \lceil k/2 \rceil) \bmod N$ if k is odd, and to the nodes $(p - k/2) \bmod N$ if k is even, where N is the total number of nodes. This is a heuristic for minimizing the communication overhead for matrices whose entries are mostly clustered around the diagonal. For these matrices, data has to be sent to nodes holding close parts of the matrix for the matrix-vector product with high probability. Thus, the data can be sent in one message, minimizing latency overhead.

In terms of latency overhead and thus runtime performance, this strategy works especially well if the matrix is not too sparse around the diagonal within a bandwidth of $\lceil \phi n/(2N) \rceil$, where n is the problem size. For matrices with other sparsity patterns, a different strategy has to be chosen (Pachajoa et al., 2019).

The value of ϕ should be chosen as small as possible since the overhead of communication in the failure-free case may increase with ϕ. This increase does not necessarily have to be linear. Pachajoa et al. (2019) show experiments where the overhead increases superlinearly with ϕ. This behaviour strongly depends on the sparsity pattern of the matrix.

In case of multiple node failures, each replacement node executes the steps outlined in Algorithm 6 for its part of the data. Most steps are executed independently of the other replacement nodes. Lines 2, 3, 5 and 7 only involve communication with surviving nodes while line 4 only involves local computation. However, the small systems of equations in lines 6 and 8 are distributed among the replacement nodes and are therefore solved in parallel.

These small systems are solved using PCG. Therefore, the reconstructed solver state is only an approximation. Since these systems of equations are small, the residual tolerance for terminating PCG can be set to a low value without increasing the overhead considerably. Pachajoa et al. (2019) show that executions with node failures are only slightly less accurate than the failure-free version when using a relative residual tolerance of 10^{-14} for the small systems, Levonyak et al. (2020) use a factor of 10^{-11} as a convergence criterion.

5. Resilient Communication-avoiding Conjugate Gradient methods

To achieve efficient and reliable executions on large-scale parallel computers, algorithms both have to be resilient against node failures and reduce the impact of communication bottlenecks. Levonyak et al. (2020) present a Conjugate Gradient algorithm that is able to address these two challenges by extending the communication-hiding PPCG algorithm using similar approaches as for resilient PCG shown in Chen (2011), Pachajoa et al. (2019) and Chapter 4.

To the best of our knowledge, we are the first to combine the communication-avoidance of s-step methods with algorithm-based resilience against node failures. In this chapter, we present novel resilient communication-avoiding s-step Conjugate Gradient methods. We extend CA-PCG and CA-PCG3 to make them resilient against node failures using similar ideas as proposed in Chen (2011) and Pachajoa et al. (2019) and described in Chapter 4 for PCG.

No communication occurs in the inner loop of CA-PCG and CA-PCG3. Therefore, no data can be stored redundantly during the inner loop iterations and the state of an inner iteration cannot be reconstructed. We derive strategies that recover the state of the solvers at the beginning of the outer iteration in which the node failures occurred. The neighbours that store redundant vector copies for a node are selected with the same strategy as described in Section 4.2 for PCG.

In Section 5.1, we present resilient CA-PCG, whose reconstruction strategy is similar to ESR for PCG presented in Chen (2011) and described in Section 4.1.1. Moreover, we show how our approach can be adapted for two algorithms that are based on the unpreconditioned algorithm CA-CG. In Section 5.2, we derive resilient PCG3, which we use to derive a resilient version of CA-PCG3 in Section 5.3. For the latter, we discuss multiple possibilities to recover the lost parts of the residual matrix $R^{(k-1)}$. In Section 5.4, we theoretically evaluate the overhead of these resilient communication-avoiding algorithms compared to their non-resilient versions.

5.1. Resilient CA-PCG

Since no communication occurs in the inner iterations of CA-PCG, redundant vector copies can only be communicated at the beginning of each outer iteration during the communication phase(s) of the Matrix Powers Kernel. In case of node failures, the current outer iteration has to be restarted. If the node failures happen before the redundant vector copies are communicated, the previous outer iteration is restarted instead.

The Matrix Powers Kernel is invoked to compute

$$Q^{(k)} = \left[\rho_0(AP)q^{(sk)}, \rho_1(AP)q^{(sk)}, \ldots, \rho_s(AP)q^{(sk)}\right], \tag{5.1}$$

$$R^{(k)} = \left[\rho_0(AP)r^{(sk)}, \rho_1(AP)r^{(sk)}, \ldots, \rho_{s-1}(AP)r^{(sk)}\right], \tag{5.2}$$

where the polynomial $\rho_j(z)$ of degree j satisfies the three-term recurrence defined in Equation (3.2) (see Section 3.1). If the computation of $P^{(k)} = PQ^{(k)}$ and $U^{(k)} = PR^{(k)}$ is not implicitly included in the computation of $Q^{(k)}$ and $R^{(k)}$, the Matrix Powers Kernel is also invoked to compute

$$P^{(k)} = \left[\rho_0(PA)p^{(sk)}, \rho_1(PA)p^{(sk)}, \ldots, \rho_s(PA)p^{(sk)}\right], \tag{5.3}$$

$$U^{(k)} = \left[\rho_0(PA)u^{(sk)}, \rho_1(PA)u^{(sk)}, \ldots, \rho_{s-1}(PA)u^{(sk)}\right]. \tag{5.4}$$

Both in communication-hiding and communication-avoiding methods, preconditioners that do not require global communication are preferred to preserve scalability (Hoemmen, 2010; Cools et al., 2019). For some preconditioner types, e.g. the Jacobi preconditioner, no communication is required for the application. Thus, we cannot rely on communicating enough parts of $q^{(sk)}$ and $r^{(sk)}$ during the application of the preconditioner to store redundant copies with little overhead.

Moreover, $p^{(sk)}$ and $u^{(sk)}$ are already available at the beginning of outer iteration k. They are computed without communication using the small vectors $p^{(k-1,s)'}$ and $r^{(k-1,s)'}$ at the end of outer iteration $k-1$ (see Algorithm 5 in Section 3.5).

During the computation of $Q^{(k)}$ and $R^{(k)}$ in Equations (5.1) and (5.2), redundant copies of $p^{(sk)} = Pq^{(sk)}$ and, if $s > 1$, $u^{(sk)} = Pr^{(sk)}$ can be stored with little overhead by exploiting the inherent redundancy of the matrix-vector products $Ap^{(sk)}$ and, if $s > 1$, $Au^{(sk)}$. Analogously, redundant copies of $p^{(sk)}$ and $u^{(sk)}$ can be stored when invoking the Matrix Powers Kernel in Equations (5.3) and (5.4). We therefore do not have to distinguish between the cases where $P^{(k)}$ and $U^{(k)}$ are computed implicitly during the computation of $Q^{(k)}$ and $R^{(k)}$ or with two additional Matrix Powers Kernel invocations.

If $s = 1$, the Matrix Powers Kernel is invoked for every search direction, since only one step is done in each outer iteration. We are able to store redundant copies of the current and the previous search direction as in resilient PCG. In case of node failures, the exact state can fully be reconstructed using ESR for PCG as described in Section 4.1 and shown in Algorithm 6 if the monomial basis is used. For other basis types, the strategy is extended by the reconstruction of the lost parts of the unpreconditioned search direction $q^{(sk)}$ using the relation $p^{(sk)} = Pq^{(sk)}$.

If $s > 1$, not every search direction is computed explicitly. Only the search direction $p^{(sk)}$ at the beginning of each outer iteration k is available and is communicated during the Matrix Powers Kernel. Thus, only every s^{th} search direction is stored redundantly and the preconditioned residual cannot be reconstructed as in ESR for PCG.

If $s = 1$, the Matrix Powers Kernel is not invoked to compute $R^{(k)}$ and $U^{(k)}$ since they only consist of one column. If $s > 1$, the communication phase(s) of the Matrix Powers Kernel when computing $R^{(k)}$ and $U^{(k)}$ in Equations (5.2) and (5.4) gives the opportunity to store $u^{(sk)}$ redundantly in each outer iteration.

In resilient PCG, the two previous search directions are stored redundantly to be able to reconstruct the lost parts of $u^{(i)}$ with the relation $p^{(i)} = u^{(i)} + \beta^{(i-1)}p^{(i-1)}$. In resilient

Algorithm 7 Exact State Reconstruction (ESR) for CA-PCG

```
Input: part of surviving nodes' state in outer iteration $k$ of resilient CA-PCG:
       $r^{(sk)}_{I\setminus I_f}$, $q^{(sk)}_{I\setminus I_f}$, $x^{(sk)}_{I\setminus I_f}$, $\beta^{(sk-1)}$ (if $s = 1$), $G^{(k)}$ (if already computed)
       redundant vector copies for failed nodes:
       $p^{(sk)}_{I_f}$, $p^{(sk-1)}_{I_f}$ (if $s = 1$) or $u^{(sk)}_{I_f}$ (if $s > 1$)
Output: reconstructed state of failed/replacement nodes:
       $A_{I_f,I}$, $P_{I_f,I}$, $b_{I_f}$, $r^{(sk)}_{I_f}$, $u^{(sk)}_{I_f}$, $q^{(sk)}_{I_f}$, $p^{(sk)}_{I_f}$, $x^{(sk)}_{I_f}$, $G_k$ (if already computed)
 1: Retrieve static data $A_{I_f,I}$, $P_{I_f,I}$ and $b_{I_f}$
 2: Gather $r^{(sk)}_{I\setminus I_f}$, $q^{(sk)}_{I\setminus I_f}$ and $x^{(sk)}_{I\setminus I_f}$
 3: if $s = 1$ then
 4:     Retrieve $\beta^{(sk-1)}$, $G^{(k)}$          ▷ $G^{(k)}$ is retrieved if it has already been computed
 5:     Retrieve $p^{(sk)}_{I_f}$ and $p^{(sk-s)}_{I_f} = p^{(sk-1)}_{I_f}$
 6:     Compute $u^{(sk)}_{I_f} = p^{(sk)}_{I_f} - \beta^{(sk-1)} p^{(sk-1)}_{I_f}$
 7: else
 8:     Retrieve $G^{(k)}$                          ▷ $G^{(k)}$ is retrieved if it has already been computed
 9:     Retrieve $p^{(sk)}_{I_f}$ and $u^{(sk)}_{I_f}$
10: end if
11: Compute $v = u^{(sk)}_{I_f} - P_{I_f,I\setminus I_f} r^{(sk)}_{I\setminus I_f}$
12: Solve $P_{I_f,I_f} r^{(sk)}_{I_f} = v$ for $r^{(sk)}_{I_f}$
13: Compute $w = b_{I_f} - r^{(sk)}_{I_f} - A_{I_f,I\setminus I_f} x^{(sk)}_{I\setminus I_f}$
14: Solve $A_{I_f,I_f} x^{(sk)}_{I_f} = w$ for $x^{(sk)}_{I_f}$
15: Compute $t = p^{(sk)}_{I_f} - P_{I_f,I\setminus I_f} q^{(sk)}_{I\setminus I_f}$
16: Solve $P_{I_f,I_f} q^{(sk)}_{I_f} = t$ for $q^{(sk)}_{I_f}$
```

CA-PCG, if $s > 1$, storing the previous search direction $p^{(sk-1)}$ redundantly is not needed to reconstruct $u^{(sk)}_{I_f}$ since $u^{(sk)}$ is stored redundantly itself.

The elements of the Gram matrix G_k have the same values on all nodes. Therefore, it can be retrieved from any surviving node if it has already been computed before the node failures happened. This avoids the recomputation of $G^{(k)}$ and thus the global reduction operation in the restarted outer iteration.

The Exact State Reconstruction (ESR) for CA-PCG for arbitrary basis types is outlined in Algorithm 7.

5.1.1. Resilience in algorithms based on CA-CG

Instead of using a fixed step size s, the adaptive CA-CG algorithms presented in Carson (2018) and Carson (2020) determine a varying step size $s^{(k)}$ for every outer iteration k to reduce accuracy loss. The largest possible step size is a user-given value s_{max}. Adaptive CA-CG can be made fault-tolerant using similar strategies as for resilient CA-PCG. In each outer iteration k of resilient adaptive CA-CG, redundant copies are stored for the search direction and, if $s^{(k)} > 1$, the residual.

If node failures occur in outer iteration k, and $s^{(k)} > 1$, we stored both the search direction and the residual redundantly at the beginning of this outer iteration. If $s^{(k)} = 1$ and $s^{(k-1)} = 1$, both the current and the previous search direction have been stored redundantly. In both cases, the state can be reconstructed using Algorithm 7 with a small extension (see below).

If $s^{(k)} = 1$ and $s^{(k-1)} > 1$, neither redundant copies of the current residual nor of the previous search direction are available. Thus, we do not have enough data to reconstruct the state of outer iteration k. Instead, the state of outer iteration $k - 1$ must be reconstructed, for which both the lost parts of the search direction and the residual can be retrieved. In the worst case, $s^{(k-1)} = s_{max}$ and $s^{(k)} = 1$, so after recovery, $s_{max} + 1$ steps must be recomputed and the communication phases of outer iterations k and $k - 1$ must be executed again.

Determining $s^{(k)}$ in outer iteration k of adaptive CA-CG is done by each node locally and requires scalars that have the same values on all nodes. Thus, the state is reconstructed using Algorithm 7 with the extension of retrieving these scalars from any surviving node.

The CA-CG algorithm using the residual replacement strategy provided in Carson and Demmel (2014) locally computes scalars to determine whether the deviation of the recursively computed residual from the true residual becomes too large. The state is reconstructed by extending Algorithm 7 by retrieving these scalars from any surviving node. Since this algorithm uses a fixed step size, redundancy to be able to tolerate node failures can be ensured using the same strategy as for resilient CA-PCG.

5.2. Resilient PCG3

To be able to derive a resilient version of CA-PCG3, we first derive resilient PCG3. The state of iteration i in PCG3 can be described with the vectors $r^{(i)}$, $u^{(i)}$, $x^{(i)}$, $r^{(i-1)}$, $u^{(i-1)}$ and $x^{(i-1)}$ and the scalars $\gamma^{(i-1)}$, $\mu^{(i-1)}$ and $\rho^{(i-1)}$.

The scalars have the same values on all nodes and can therefore be retrieved from any surviving node. Redundant copies of $u^{(i)}$ are stored during the communication phase for the matrix-vector product $Au^{(i)}$. The lost parts of $r^{(i)}$ are reconstructed using the relation $u^{(i)} = Pr^{(i)}$. Using $r^{(i)}$, the lost parts of $x^{(i)}$ are recovered as in ESR for PCG, see lines 7 and 8 of Algorithm 6.

Since PCG3 is based on three-term recurrences, the redundant copies of $u^{(i-1)}$, which were communicated in the previous iteration, are kept in iteration i. With these redundant vector copies, the lost parts of $r^{(i-1)}$ and $x^{(i-1)}$ are reconstructed in case of node failures.

The Exact State Reconstruction (ESR) for PCG3 is outlined in Algorithm 8.

5.3. Resilient CA-PCG3

We derive a resilient version of CA-PCG3 based on resilient PCG3. Analogously to resilient CA-PCG, the outer iteration in which the failures occurred has to be restarted. If the node failures happened before the communication of the redundant vector copies during the Matrix Powers Kernel, the previous outer iteration must be restarted.

Resilient PCG and CA-PCG reconstruct the same data to recover the state, except for the unpreconditioned search direction. In contrast, the state of an iteration in resilient PCG3 is not enough to describe an outer iteration of resilient CA-PCG3, whose state also contains $R^{(k-1)}$, $U^{(k-1)}$ and $T^{(k-1)}$.

Algorithm 8 Exact State Reconstruction (ESR) for PCG3

```
Input: part of surviving nodes' state in iteration $i$ of resilient PCG3:
    $x_{I\setminus I_f}^{(i)}$, $x_{I\setminus I_f}^{(i-1)}$, $r_{I\setminus I_f}^{(i)}$, $r_{I\setminus I_f}^{(i-1)}$, $\gamma^{(i-1)}$, $\mu^{(i-1)}$, $\rho^{(i-1)}$
    redundant vector copies for failed nodes:
    $u_{I_f}^{(i)}$, $u_{I_f}^{(i-1)}$
Output: reconstructed state of failed/replacement nodes:
    $A_{I_f,I}$, $P_{I_f,I}$, $b_{I_f}$, $\gamma^{(i-1)}$, $\mu^{(i-1)}$, $\rho^{(i-1)}$, $r_{I_f}^{(i)}$, $r_{I_f}^{(i-1)}$, $u_{I_f}^{(i)}$, $u_{I_f}^{(i-1)}$ $x_{I_f}^{(i)}$, $x_{I_f}^{(i-1)}$
 1: Retrieve static data $A_{I_f,I}$, $P_{I_f,I}$ and $b_{I_f}$
 2: Gather $x_{I\setminus I_f}^{(i)}$, $x_{I\setminus I_f}^{(i-1)}$, $r_{I\setminus I_f}^{(i)}$ and $r_{I\setminus I_f}^{(i-1)}$
 3: Retrieve $\gamma^{(i-1)}$, $\mu^{(i-1)}$, $\rho^{(i-1)}$, $u_{I_f}^{(i)}$ and $u_{I_f}^{(i-1)}$
 4: Compute $v = u_{I_f}^{(i)} - P_{I_f,I\setminus I_f} r_{I\setminus I_f}^{(i)}$
 5: Solve $P_{I_f,I_f} r_{I_f}^{(i)} = v$ for $r_{I_f}^{(i)}$
 6: Compute $w = b_{I_f} - r_{I_f}^{(i)} - A_{I_f,I\setminus I_f} x_{I\setminus I_f}^{(i)}$
 7: Solve $A_{I_f,I_f} x_{I_f}^{(i)} = w$ for $x_{I_f}^{(i)}$
 8: Compute $s = u_{I_f}^{(i-1)} - P_{I_f,I\setminus I_f} r_{I\setminus I_f}^{(i-1)}$
 9: Solve $P_{I_f,I_f} r_{I_f}^{(i-1)} = s$ for $r_{I_f}^{(i-1)}$
10: Compute $t = b_{I_f} - r_{I_f}^{(i-1)} - A_{I_f,I\setminus I_f} x_{I\setminus I_f}^{(i-1)}$
11: Solve $A_{I_f,I_f} x_{I_f}^{(i-1)} = t$ for $x_{I_f}^{(i-1)}$
```

An exception is the case $s = 1$, for which ESR for PCG3 can be used for reconstruction. In resilient CA-PCG3 with $s = 1$, every residual can be stored redundantly. The lost parts of $R^{(k-1)} = \left[r^{(sk-1)}\right]$ and $U^{(k-1)} = \left[u^{(sk-1)}\right]$ are recovered in ESR for PCG3. $T^{(k-1)}$ does not need to be recovered since it is used to compute $d^{(sk+j)}$ and $g^{(sk+j)}$ for $1 \leq j < s = 1$. Thus, $T^{(k)}$ is never used if $s = 1$.

5.3.1. Recovery of scalars

The matrix $T^{(k-1)}$ as defined in Equations (3.21) and (3.22) is computed at the end of the previous outer iteration $k-1$ with the scalars $\gamma^{(sk-s)}, \ldots, \gamma^{(sk-1)}$ and $\rho^{(sk-s)}, \ldots, \rho^{(sk-1)}$. In case of node failures, these scalars are retrieved from any surviving node and $T^{(k-1)}$ can be recomputed locally on the replacement nodes. Alternatively, since $T^{(k-1)}$ is of size $s \times s$ and its elements have the same values on all nodes, it can be retrieved from any surviving node. Depending on the method chosen to recover the lost vector parts of the residuals, the replacement nodes might have to retrieve $\gamma^{(sk-s)}, \ldots, \gamma^{(sk-1)}$ and $\rho^{(sk-s)}, \ldots, \rho^{(sk-1)}$ anyway.

The upper left part $U^{(k-1)^T} R^{(k-1)}$ of the Gram matrix $G^{(k)}$ is a diagonal matrix (see Section 3.4). The diagonal entries are already computed in the previous outer iteration as the scalars $\mu^{(sk-s)}, \ldots, \mu^{(sk-1)}$ and can be retrieved from any surviving node. As with resilient CA-PCG, the Gram matrix can be retrieved from any surviving node if it has already been computed before the nodes failed to avoid the overhead of the global reduction operation in the restarted outer iteration. In this case, the scalars $\mu^{(sk-s)}, \ldots, \mu^{(sk-2)}$ do not have to be retrieved. For $s = 1$, ESR of PCG3 can be extended by the retrieval of the Gram matrix.

Let l be the inner loop iteration in which the node failures happened, i.e. $0 \leq l < s$. The lost parts of the residuals and the approximate solution computed during the iterations of the inner loop cannot be reconstructed. They must be recomputed after restarting the outer iteration.

However, we could retrieve locally computed scalars and small vectors that have the same values on all nodes from any surviving node. These include the vectors $d^{(sk+j)}$ and $g^{(sk+j)}$ for $j = -1, 0, 1, \ldots, l$, which are each of size $2s + 1$, and the scalars $\gamma^{(sk+j)}$, $\rho^{(sk+j)}$ and $\mu^{(sk+j)}$ for $j = 0, 1, \ldots, l$. However, recomputing these values in the inner loop of the restarted outer iteration is computationally inexpensive and does not involve communication.

5.3.2. Recovery of lost vector parts

To restart the outer CA-PCG3 iteration k, the lost parts of $u^{(sk)}$, $r^{(sk)}$, $x^{(sk)}$, $x^{(sk-1)}$, $R^{(k-1)}$ and $U^{(k-1)}$ need to be recovered. The lost parts of $u^{(sk-1)}$ and $r^{(sk-1)}$ are recovered implicitly as the last columns of $R^{(k-1)}$ and $U^{(k-1)}$. The lost parts of the solution vectors can be recovered as in ESR for PCG3.

In each outer iteration k of CA-PCG3, the Matrix Powers Kernel is invoked to compute

$$W^{(k)} = \left[\rho_0(AP)r^{(sk)}, \rho_1(AP)r^{(sk)}, \ldots, \rho_s(AP)r^{(sk)}\right], \tag{5.5}$$

$$V^{(k)} = \left[\rho_0(PA)u^{(sk)}, \rho_1(PA)u^{(sk)}, \ldots, \rho_s(PA)u^{(sk)}\right]. \tag{5.6}$$

Similar to resilient CA-PCG, since we can store $u^{(sk)}$ redundantly in both Equations (5.5) and (5.6), we do not have to distinguish between the case where we compute $V^{(k)} = PW^{(k)}$ with an additional Matrix Powers Kernel invocation or implicitly during the computation of $W^{(k)}$.

If $s > 1$, $u^{(sk-1)}$ is never communicated in non-resilient CA-PCG3. Moreover, to compute the matrix-vector products $Au^{(sk+j)} = \left[R^{(k-1)}, W^{(k)}\right] d^{(sk+j)}$ for $0 \leq j < s$ without communication in the inner iterations, we need the matrix $R^{(k-1)}$, whose columns are the s residuals of the previous outer iteration. Below, we present three different methods for recovering these residuals.

Method 1: Retrieving lost parts of $R^{(k-1)}$ from surviving neighbour nodes

In the first method, redundant copies of $R^{(k-1)}$ are stored at the beginning of each outer iteration k. The elements of $R^{(k-1)}$ owned by a node can be sent along with the elements of $u^{(sk)}$ of this node in the communication phase(s) of the Matrix Powers Kernel to avoid additional latency.

With $R^{(k-1)}$, the lost parts of the matrix $U^{(k-1)} = PR^{(k-1)}$ can be recomputed. Since $r^{(sk-1)}$ and $u^{(sk-1)}$ are the last columns of $R^{(k-1)}$ and $U^{(k-1)}$, these vectors are implicitly recovered.

Additional latency is not necessary to communicate $R^{(k-1)}$ during the Matrix Powers Kernel. Moreover, each node does not have to send its elements of $R^{(k-1)}$ to every neighbour it communicates with, but only to the ϕ nodes that will store its data redundantly. The other neighbours can still receive only its elements of $u^{(sk)}$.

Algorithm 9 Exact State Reconstruction (ESR) for CA-PCG3 using Method 1

```
Input: part of surviving nodes' state in outer iteration $k$ of resilient CA-PCG3:
        $x^{(sk)}_{I\setminus I_f}$, $x^{(sk-1)}_{I\setminus I_f}$, $r^{(sk)}_{I\setminus I_f}$, $R^{(k-1)}_{I\setminus I_f}$, $\gamma^{(sk-1)}$, $\mu^{(sk-1)}$, $\rho^{(sk-1)}$,
        $T^{(k-1)}$ (if it will be retrieved) or $\gamma^{(sk-s)}, \dots, \gamma^{(sk-2)}$, $\rho^{(sk-s)}, \dots, \rho^{(sk-2)}$,
        $\mu^{(sk-s)}, \dots, \mu^{(sk-2)}$ or $G^{(k)}$ (if already computed)
        redundant vector copies for failed nodes:
        $u^{(sk)}_{I_f}$, $u^{(sk-s)}_{I_f}$, last $s-1$ columns of $R^{(k-1)}_{I_f}$
Output: reconstructed state of failed/replacement nodes:
        $A_{I_f,I}$, $P_{I_f,I}$, $b_{I_f}$, $\gamma^{(sk-1)}, \rho^{(sk-1)}, \mu^{(sk-s)}, \dots, \mu^{(sk-1)}$ or $G_k$ (if already computed),
        $T^{(k-1)}$, $r^{(sk)}_{I_f}$, $u^{(sk)}_{I_f}$, $x^{(sk)}_{I_f}$, $x^{(sk-1)}_{I_f}$, $R^{(k-1)}_{I_f}$, $U^{(k-1)}_{I_f}$
 1: Retrieve static data $A_{I_f,I}$, $P_{I_f,I}$ and $b_{I_f}$
 2: Gather $x^{(sk)}_{I\setminus I_f}$, $x^{(sk-1)}_{I\setminus I_f}$, $r^{(sk)}_{I\setminus I_f}$ and $R^{(k-1)}_{I\setminus I_f}$
 3: Retrieve $\gamma^{(sk-1)}$, $\rho^{(sk-1)}$, $\mu^{(sk-1)}$,
        $\gamma^{(sk-s)}, \dots, \gamma^{(sk-2)}$, $\rho^{(sk-s)}, \dots, \rho^{(sk-2)}$, ▷ only necessary if $T^{(k-1)}$ is recomputed
        $G^{(k)}$ or $\mu^{(sk-s)}, \dots, \mu^{(sk-2)}$ ▷ Depending on whether $G^{(k)}$ has been computed yet
 4: Retrieve or recompute $T^{(k-1)}$
 5: Retrieve $u^{(sk)}_{I_f}$, $u^{(sk-s)}_{I_f}$ and the last $s-1$ columns of $R^{(k-1)}_{I_f}$
 6: Compute $v = u^{(sk)}_{I_f} - P_{I_f,I\setminus I_f} r^{(sk)}_{I\setminus I_f}$
 7: Solve $P_{I_f,I_f} r^{(sk)}_{I_f} = v$ for $r^{(sk)}_{I_f}$
 8: Compute $s = u^{(sk-s)}_{I_f} - P_{I_f,I\setminus I_f} r^{(sk-s)}_{I\setminus I_f}$
 9: Solve $P_{I_f,I_f} r^{(sk-s)}_{I_f} = s$ for $r^{(sk-s)}_{I_f}$
10: Compute the last $s-1$ columns of $U^{(k-1)}_{I_f} = P_{I_f,I} R^{(k-1)}_I$
11: Recover $x^{(sk)}_{I_f}$ and $x^{(sk-1)}_{I_f}$ using relation $b - Ax^{(i)} = r^{(i)}$
                                            ▷ (cf. lines 6, 7, 10 and 11 of Algorithm 8)
```

The residual $u^{(sk-s)}$ is stored redundantly in outer iteration $k-1$. If these redundant copies are kept in outer iteration k, they can be retrieved in case of node failures in outer iteration k. The first column of $R^{(k-1)}$ can be recovered using the relation $u^{(sk-s)} = Pr^{(sk-s)}$. Thus, only the last $s-1$ columns of $R^{(k-1)}$ need to be sent along with $u^{(sk)}$ in each outer iteration k.

Method 1 of the Exact State Reconstruction (ESR) for CA-PCG3 is outlined in Algorithm 9.

Since node failures are rare, we aim to reduce the overhead for resilience in the failure-free case. In resilient CA-PCG3 using Method 1, the amount of data communicated increases with s since $R^{(k-1)}$ has s columns. We therefore present alternatives to this approach in which the amount of communicated data in each outer iteration is independent of s.

Method 2: Replacing the outer iteration to be restarted with s PCG3 iterations

In this second method, instead of $R^{(k-1)}$, only its last column $r^{(sk-1)}$ is communicated together with $u^{(sk)}$ during the Matrix Powers Kernel. Alternatively, the last column of $U^{(sk)}$, i.e. $u^{(sk-1)}$, can be communicated with $u^{(sk)}$.

Algorithm 10 Exact State Reconstruction (ESR) for CA-PCG3 using Method 2

```
Input: part of surviving nodes' state in outer iteration $k$ of resilient CA-PCG3:
       $x^{(sk)}_{I\setminus I_f}$, $x^{(sk-1)}_{I\setminus I_f}$, $r^{(sk)}_{I\setminus I_f}$, $r^{(sk-1)}_{I\setminus I_f}$, $\gamma^{(sk-1)}$, $\mu^{(sk-1)}$, $\rho^{(sk-1)}$, $G^{(k)}$ (if already computed)
       redundant vector copies for failed nodes:
       $u^{(sk)}_{I_f}$, $u^{(sk-1)}_{I_f}$
Output: reconstructed state of failed/replacement nodes such that the algorithm
        can be continued with $s$ PCG3 iterations:
        $A_{I_f,I}$, $P_{I_f,I}$, $b_{I_f}$, $\gamma^{(sk-1)}$, $\rho^{(sk-1)}$, $\mu^{(sk-1)}$, $G_k$ (if already computed),
        $r^{(sk)}_{I_f}$, $r^{(sk-1)}_{I_f}$, $u^{(sk)}_{I_f}$, $u^{(sk-1)}_{I_f}$, $x^{(sk)}_{I_f}$, $x^{(sk-1)}_{I_f}$
1: Retrieve static data $A_{I_f,I}$, $P_{I_f,I}$ and $b_{I_f}$
2: Gather $x^{(sk)}_{I\setminus I_f}$, $x^{(sk-1)}_{I\setminus I_f}$, $r^{(sk)}_{I\setminus I_f}$ and $r^{(sk-1)}_{I\setminus I_f}$
3: Retrieve $\gamma^{(sk-1)}, \rho^{(sk-1)}, \mu^{(sk-1)}, G^{(k)}$          ▷ $G^{(k)}$ is retrieved if already computed
4: Retrieve $u^{(sk)}_{I_f}$ and $r^{(sk-1)}_{I_f}$
5: Compute $v = u^{(sk)}_{I_f} - P_{I_f,I\setminus I_f} r^{(sk)}_{I\setminus I_f}$
6: Solve $P_{I_f,I_f} r^{(sk)}_{I_f} = v$ for $r^{(sk)}_{I_f}$
7: Compute $u^{(sk-1)}_{I_f} = P_{I_f,I} r^{(sk-1)}_I$
8: Recover $x^{(sk)}_{I_f}$ and $x^{(sk-1)}_{I_f}$ using relation $b - Ax^{(i)} = r^{(i)}$
                                              ▷ (cf. lines 6, 7, 10 and 11 of Algorithm 8)
```

Depending on whether $r^{(sk-1)}$ or $u^{(sk-1)}$ are communicated, either the lost parts of $u^{(sk-1)}$ are recovered using the relation $u^{(sk-1)} = Pr^{(sk-1)}$, or a small system of equations for the lost parts of $r^{(sk-1)}$ is solved. One of these two options may offer a performance advantage, depending on how the preconditioner is stored and applied. We chose to store $r^{(sk-1)}$ redundantly in resilient CA-PCG3 using Methods 2 and 3 in our experiments. Therefore, we will assume this choice in the remainder of the thesis.

Since the lost parts of $R^{(k-1)}$ cannot be retrieved from a surviving neighbour, s iterations of PCG3 are executed instead of restarting the outer iteration. After s steps of PCG3 and therefore obtaining s residuals, the algorithm can be continued with CA-PCG3 iterations. Executing the next s steps with PCG3 increases the communication overhead after node failures. In addition to communication for s single matrix-vector products and preconditioner applications, s global reduction operations are executed.

If the Gram matrix has already been computed before the node failures happened, its elements can be retrieved from any surviving node. In this case, even without recovering the lost parts of $R^{(k-1)}$ and $U^{(k-1)}$, the scalar products $\mu^{(sk+j)} = {r^{(sk+j)}}^T u^{(sk+j)}$ and $\nu^{(sk+j)} = {w^{(sk+j)}}^T u^{(sk+j)}$ in the inner iterations $j = 0, \dots s-1$ can be computed as $\mu^{(sk+j)} = {g^{(sk+j)}}^T G^{(k)} g^{(sk+j)}$ and $\nu^{(sk+j)} = {g^{(sk+j)}}^T G^{(k)} d^{(sk+j)}$ as in lines 9 and 10 of Algorithm 4 without global communication. The computation of $d^{(sk+j)}$ and $g^{(sk+j)}$ is inexpensive and can be done locally on each node.

Method 2 of the Exact State Reconstruction (ESR) for CA-PCG3 is outlined in Algorithm 10.

Method 3: Recovering lost parts of $R^{(k-1)}$ using the three-term recurrence relation

In the worst case, Method 2 needs s additional global reduction operations involving all nodes to compute the scalar products in the restarted outer iteration after node failures. To avoid this global communication overhead, the lost parts of $R^{(k-1)}$ can be reconstructed by exploiting the three-term recurrence relation between consecutive residuals.

In this third alternative, $u^{(sk)}$ and in every second outer iteration additionally $r^{(sk-1)}$ are stored redundantly. As in Method 2, we could also choose to communicate $u^{(sk-1)}$ instead of $r^{(sk-1)}$.

The lost parts of $r^{(sk)}$ and $r^{(sk-s)}$, the first column of $R^{(k-1)}$, are reconstructed using the relations $u^{(sk)} = Pr^{(sk)}$ respectively $u^{(sk-s)} = Pr^{(sk-s)}$. The lost parts of the remaining columns of $R^{(k-1)}$ are recovered using the recursive formula for the residual in Equation (2.40), which we repeat here for $r^{(sk-s+j+1)}$.

$$\begin{aligned} r^{(sk-s+j+1)} =& \rho^{(sk-s+j)} \left(r^{(sk-s+j)} - \gamma^{(sk-s+j)} A u^{(sk-s+j)} \right) \\ &+ \left(1 - \rho^{(sk-s+j)} \right) r^{(sk-s+j-1)} \end{aligned} \tag{5.7}$$

By rearranging Equation (5.7), we get

$$\begin{aligned} r^{(sk-s+j-1)} =& \frac{1}{1-\rho^{(sk-s+j)}} r^{(sk-s+j+1)} \\ &- \frac{\rho^{(sk-s+j)}}{1-\rho^{(sk-s+j)}} \left(r^{(sk-s+j)} - \gamma^{(sk-s+j)} A u^{(sk-s+j)} \right). \end{aligned} \tag{5.8}$$

If we stored $r^{(sk-s-1)}$ redundantly in outer iteration $k-1$, the lost parts of $R^{(k-1)}$ are recomputed using Equation (5.7) for $j = 0, \ldots, s-2$. If we stored $r^{(sk-1)}$ redundantly in outer iteration k, the lost parts of $R^{(k-1)}$ are reconstructed using Equation (5.8) for $j = 2, \ldots, s-1$. In this case, one less matrix-vector multiplication is required since the last column of $R^{(k-1)}$ is stored redundantly.

We do not need to compute the global result vectors of the matrix-vector products and preconditioner applications in Equations (5.7) and (5.8), only the parts owned by the replacement nodes. The surviving nodes still hold their parts of $r^{(sk-s+j)}$ and $u^{(sk-s+j)}$ since they are stored in their parts of $R^{(k-1)}$ and $U^{(k-1)}$. The neighbours of the replacement nodes communicate their parts of $r^{(sk-s+j)}$ for the local computation of the lost parts of $u^{(sk-s+j)} = Pr^{(sk-s+j)}$. After receiving the necessary parts of $u^{(sk-s+j)}$ from their neighbours, the replacement nodes compute their parts of $Au^{(sk-s+j)}$ locally.

Using Equations (5.7) and (5.8) requires resilient CA-PCG3 to additionally store $\rho^{(sk-s+j)}$ and $\gamma^{(sk-s+j)}$ for $0 \leq j < s-1$ respectively $2 \leq j < s$. These scalars have been computed in the previous outer iteration and can be retrieved from any surviving node.

The lost parts of $u^{(sk-s)}$, the first column of $U^{(k-1)}$, are retrieved since $u^{(sk-s)}$ has been stored redundantly in outer iteration $k-1$. The lost parts of almost all remaining columns of $U^{(k)}$ are computed implicitly during the recovery of the lost parts of $R^{(k-1)}$. When using Equation (5.8), the lost parts of the second column of $U^{(k-1)}$ are computed using the relation $u^{(sk-s+1)} = Pr^{(sk-s+1)}$. When using Equation (5.7), the lost parts of the last column of $U^{(k-1)}$ are computed using the relation $u^{(sk-1)} = Pr^{(sk-1)}$.

The recovery of the lost parts of $U^{(k-1)}$ requires $s-1$ preconditioner applications. While the recomputation of the lost parts of $R^{(k-1)}$ needs $s-1$ matrix-vector products if

Equation (5.7) is used, only $s-2$ are necessary when using Equation (5.8) since in that case $r^{(sk-1)}$ is stored redundantly.

Method 3 of the Exact State Reconstruction (ESR) for CA-PCG3 is outlined in Algorithm 11.

Whether Method 1, 2 or 3 should be used to make CA-PCG3 resilient against node failures depends on what the main bottleneck of performance is. Unlike Methods 2 and 3, the overhead of Method 1 increases with s in the failure-free case since the last $s-1$ columns of $R^{(k-1)}$ are communicated in each outer iteration k. However, the cost for recovery is higher for Methods 2 and 3 than for Method 1. Method 2 executes s PCG3 iterations instead of the restarted outer CA-PCG3 iteration, and more local computation is necessary in ESR for CA-PCG3 using Method 3 than in ESR for CA-PCG3 using Method 1 or 2. A detailed theoretical comparison of resilient CA-PCG and resilient CA-PCG3 using Method 1, 2 and 3 can be found in following section.

5.4. Theoretical evaluation of resilient CA-PCG and CA-PCG3

In this section, we analyse memory storage and communication overhead for data redundancy in resilient CA-PCG and CA-PCG3 and theoretically compare the cost for recovery in case of node failures.

5.4.1. Overhead for data redundancy

Overhead for algorithm-based resilience due to data redundancy occurs even in the failure-free case. We assume that $s > 1$ for the communication-avoiding methods since resilient CA-PCG and CA-PCG3 with $s = 1$ have the same memory storage and communication overhead as resilient PCG respectively PCG3.

Memory storage overhead

Since $\phi + 1$ copies of a vector are stored to be able to tolerate ϕ simultaneous node failures, the memory storage overhead increases linearly with ϕ. Resilient CA-PCG and CA-PCG3 need to keep a few scalars throughout the outer iteration that are discarded earlier in the non-resilient algorithms, which we neglect in the following. We assume that $r^{(sk-1)}$ is communicated instead of $u^{(sk-1)}$ in resilient CA-PCG3 using Methods 2 and 3.

Resilient CA-PCG stores redundant copies of $p^{(sk)}$ and $u^{(sk)}$ in every outer iteration k. Resilient PCG3 and resilient CA-PCG3 using Method 2 store redundant copies of the current and the last residual vectors in every (outer) iteration, i.e. $u^{(i)}$ and $u^{(i-1)}$ respectively $u^{(sk)}$ and $r^{(sk-1)}$. Thus, the memory storage overhead is the same as for resilient PCG, which stores the last two search directions $p^{(i)}$ and $p^{(i-1)}$ redundantly.

Resilient CA-PCG3 using Method 3 stores redundant copies of $u^{(sk)}$, $u^{(sk-s)}$ and either $r^{(sk-1)}$ or $r^{(sk-s-1)}$. Thus, the memory storage overhead is 1.5 times that of resilient PCG.

Resilient CA-PCG3 using Method 1 stores redundant copies of $u^{(sk)}$, $u^{(sk-s)}$ and of the last $s-1$ columns of $R^{(k-1)}$, i.e. of $s+1$ vectors. Therefore, the memory storage overhead is $(s+1)/2$ times that of resilient PCG.

In non-resilient CA-PCG3, the approximate solutions $x^{(sk-2)}$ and $x^{(sk-1)}$ are not needed and therefore discarded after the first and second inner iterations. In resilient CA-PCG3, these vectors have to be kept throughout outer iteration k to be able to recover their lost

Algorithm 11 Exact State Reconstruction (ESR) for CA-PCG3 using Method 3

```
Input: part of surviving nodes' state in outer iteration $k$ of resilient CA-PCG3:
    $x^{(sk)}_{I\setminus I_f}$, $x^{(sk-1)}_{I\setminus I_f}$, $r^{(sk)}_{I\setminus I_f}$, $R^{(k-1)}_{I\setminus I_f}$, $U^{(k-1)}_{I\setminus I_f}$, $\gamma^{(sk-s+2)}, \dots, \gamma^{(sk-1)}$,
    $\rho^{(sk-s+2)}, \dots, \rho^{(sk-1)}$, $\mu^{(sk-1)}$, $T^{(k-1)}$ (if it will be retrieved),
    $\gamma^{(sk-s)}, \gamma^{(sk-s+1)}$, $\rho^{(sk-s)}, \rho^{(sk-s+1)}$, (if $T^{(k-1)}$ is recomputed or if $r^{(sk-s-1)}$
    was stored redundantly), $\mu^{(sk-s)}, \dots, \mu^{(sk-2)}$ or $G^{(k)}$ (if already computed)
    redundant vector copies for failed nodes:
    $u^{(sk)}_{I_f}$, $u^{(sk-s)}_{I_f}$, $r^{(sk-1)}_{I_f}$ or $r^{(sk-s-1)}_{I_f}$
Output: reconstructed state of failed/replacement nodes:
    $A_{I_f,I}$, $P_{I_f,I}$, $b_{I_f}$, $\gamma^{(sk-1)}, \rho^{(sk-1)}, \mu^{(sk-1)}, \mu^{(sk-s)}, \dots, \mu^{(sk-2)}$ or $G_k$ (if already
    computed), $T^{(k-1)}$, $r^{(sk)}_{I_f}$, $u^{(sk)}_{I_f}$, $x^{(sk)}_{I_f}$, $x^{(sk-1)}_{I_f}$, $R^{(k-1)}_{I_f}$, $U^{(k-1)}_{I_f}$
1: Retrieve static data $A_{I_f,I}$, $P_{I_f,I}$ and $b_{I_f}$
2: Gather $x^{(sk)}_{I\setminus I_f}$, $x^{(sk-1)}_{I\setminus I_f}$, $r^{(sk)}_{I\setminus I_f}$, $R^{(k-1)}_{I\setminus I_f}$ and $U^{(k-1)}_{I\setminus I_f}$
3: Retrieve $\gamma^{(sk-s+2)}, \dots, \gamma^{(sk-1)}$, $\rho^{(sk-s+2)}, \dots, \rho^{(sk-1)}$, $\mu^{(sk-1)}$,
       $\gamma^{(sk-s)}, \gamma^{(sk-s+1)}$, $\rho^{(sk-s)}, \rho^{(sk-s+1)}$,  ▷ only necessary if $T^{(k-1)}$ is recomputed or
   if $r^{(sk-s-1)}$ and not $r^{(sk-1)}$ was stored redundantly
       $G^{(k)}$ or $\mu^{(sk-s)}, \dots, \mu^{(sk-2)}$  ▷ Depending on whether $G^{(k)}$ has been computed yet
4: Retrieve or recompute $T^{(k-1)}$
5: Retrieve $u^{(sk)}_{I_f}$, $u^{(sk-s)}_{I_f}$ and $r^{(sk-1)}_{I_f}$ or $r^{(sk-s-1)}_{I_f}$
6: Compute $v = u^{(sk)}_{I_f} - P_{I_f,I\setminus I_f} r^{(sk)}_{I\setminus I_f}$
7: Solve $P_{I_f,I_f} r^{(sk)}_{I_f} = v$ for $r^{(sk)}_{I_f}$
8: Compute $s = u^{(sk-s)}_{I_f} - P_{I_f,I\setminus I_f} r^{(sk-s)}_{I\setminus I_f}$
9: Solve $P_{I_f,I_f} r^{(sk-s)}_{I_f} = s$ for $r^{(sk-s)}_{I_f}$
10: if $r^{(sk-s-1)}$ was stored redundantly then
11:     for $j = 0, \dots, s-2$ do
12:         if $j = 0$ then                      ▷ $u^{(sk-s+j)} = u^{(sk-s)}$ was stored redundantly
13:             Compute $u^{(sk-s+j)}_{I_f} = P_{I_f,I} r^{(sk-s+j)}_I$      ▷ Recover $j^{th}$ column of $U^{(k-1)}$
14:         end if
15:         Compute          ▷ Recover $(j+1)^{th}$ column of $R^{(k-1)}_{I_f}$ using Equation (5.7)
                $r^{(sk-s+j+1)}_{I_f} = \rho^{(sk-s+j)} \left( r^{(sk-s+j)}_{I_f} - \gamma^{(sk-s+j)} A_{I_f,I} u^{(sk-s+j)}_I \right) + \left(1 - \rho^{(sk-s+j)}\right) r^{(sk-s+j-1)}_{I_f}$
16:     end for
17:     Compute $u^{(sk-1)}_{I_f} = P_{I_f,I} r^{(sk-1)}_I$                ▷ Recover last column of $U^{(k-1)}$
18: else                                        ▷ If $r^{(sk-1)}$ was stored redundantly
19:     for $j = s-1, \dots, 2$ do
20:         Compute $u^{(sk-s+j)}_{I_f} = P_{I_f,I} r^{(sk-s+j)}_I$          ▷ Recover $j^{th}$ column of $U^{(k-1)}$
```

```
21:         Compute          ▷ Recover $(j-1)^{th}$ column of $R_{I_f}^{(k-1)}$ using Equation (5.8)
```

$$\begin{aligned} r_{I_f}^{(sk-s+j-1)} =& \frac{1}{1-\rho^{(sk-s+j)}} r_{I_f}^{(sk-s+j+1)} \\ &- \frac{\rho^{(sk-s+j)}}{1-\rho^{(sk-s+j)}} \left(r_{I_f}^{(sk-s+j)} - \gamma^{(sk-s+j)} A_{I_f,I} u_I^{(sk-s+j)} \right) \end{aligned}$$

```
22:     end for
23:     Compute $u_{I_f}^{(sk-s+1)} = P_{I_f,I} r_I^{(sk-s+1)}$          ▷ Recover second column of $U^{(k-1)}$
24: end if
25: Recover $x_{I_f}^{(sk)}$ and $x_{I_f}^{(sk-1)}$ using relation $b - Ax^{(i)} = r^{(i)}$
                                             ▷ (cf. lines 6, 7, 10 and 11 of Algorithm 8)
```

parts on the replacement nodes and to restart the outer iteration on the surviving nodes. No memory storage overhead is required for the corresponding residual vectors since they are stored in $R^{(k-1)}$ and $U^{(k-1)}$.

Communication overhead

Non-resilient CA-PCG communicates two vectors in each outer iteration, the preconditioned search direction and the preconditioned residual. These vectors are stored redundantly in resilient CA-PCG.

Non-resilient CA-PCG3 only communicates the current preconditioned residual in each outer iteration. Resilient CA-PCG3 also communicates the previous residual in every outer iteration when using Method 2 and in every second outer iteration if Method 3 is used to store redundant copies of these vectors. When Method 1 is used, the last $s-1$ columns of $R^{(k-1)}$ are communicated together with the current preconditioned residual. Sending multiple vectors during the communication phase(s) of the Matrix Powers Kernel does not increase latency since they can be communicated together in the same messages. Not every neighbour that communicates with a node has to receive its part of the additional vectors, only the neighbours that store its redundant copies.

Non-resilient CA-PCG and CA-PCG3 have the same latency cost per outer iteration in the communication phase(s) of the Matrix Powers Kernel invocations, see Section 3.7. However, the message sizes are larger in non-resilient CA-PCG since two vectors are communicated instead of one as in non-resilient CA-PCG3.

The choice of neighbours that store redundant copies depends on the sparsity structure of A and the number of redundant copies ϕ of each vector, see Sections 4.1.2 and 4.2. Thus, resilient CA-PCG and resilient CA-PCG3 using Methods 1, 2 and 3 have the same communication latency costs, but the sizes of the sent messages differ.

Not every neighbour a node communicates with has to receive its parts of the additional vectors in resilient CA-PCG3, only the neighbours that store its redundant copies. If the nodes were to send the same data to the neighbours with which they communicate, resilient CA-PCG3 using Method 1 or 2 would send at least the same size of messages as resilient CA-PCG. When Method 3 is used, where an additional vector is only sent every second iteration, resilient CA-PCG3 would still have an advantage over resilient CA-PCG regarding the size of send messages.

Table 5.1.: Cost of Exact State Reconstruction (ESR). The cost for retrieving the scalars and $G^{(k)}$ and of recovering $T^{(k-1)}$ is neglected. We assume that $r^{(sk-1)}$ instead of $u^{(sk-1)}$ is stored redundantly in resilient CA-PCG3 using Methods 2 and 3.

	PCG	CA-PCG	PCG3	CA-PCG3		
operation				M1	M2	M3
global vectors gathered	2	3	4	$s+3$	4	$2s+2$
redundant vector copies retrieved	2	2	2	$s+1$	2	3
local vector updates	4	4	6	6	5	$2s+4$
applying $A_{I_f,I}$ or $A_{I_f,I\setminus I_f}$	1	1	2	2	2	$s+1$
applying $P_{I_f,I}$ or $P_{I_f,I\setminus I_f}$	1	2	2	$s+1$	2	$s+1$
solving small systems with A_{I_f,I_f}	1	1	2	2	2	2
solving small systems with P_{I_f,I_f}	1	2	2	2	1	2

We assume that the number ϕ of redundant copies of each vector is small. If we send the additional vectors only to the neighbours that store these vectors redundantly, resilient CA-PCG3 using Methods 1 and 2 might communicate less data than resilient CA-PCG.

As described in Pachajoa et al. (2019) and Section 4.2, the ϕ neighbours that store redundant copies for a node are selected to minimize the latency overhead for resilience. Since this latency overhead does not increase with s, the communication-avoiding properties of MPK-PA1 and MPK-PA2 are preserved. Thus, these two Matrix Powers Kernel algorithms are able to reduce communication latency cost by a factor of $\mathcal{O}(s)$ compared to MPK-PA0, both when used for the non-resilient and when used for the resilient algorithms.

5.4.2. Overhead for recovery in case of node failures

In this section, we analyze the cost for recovery in case of node failures. We assume that $s > 1$ for the communication-avoiding methods since their recovery cost is almost identical to the cost of recovery for resilient PCG and PCG3 if $s = 1$.

Cost of Exact State Reconstruction (ESR)

Table 5.1 summarizes the operations performed by the replacement nodes for the Exact State Reconstruction (ESR). The cost of retrieving the scalars as well as the small matrices $G^{(k)}$ and $T^{(k-1)}$ respectively the recomputation of $T^{(k-1)}$ is neglected. We consider the operations for resilient CA-PCG3 using Methods 2 and 3 when redundant copies of $r^{(sk-1)}$ are stored instead of $u^{(sk-1)}$. We also take into account the calculation of the right-hand sides for the small systems of equations solved on the replacement nodes.

We define a local vector update as an operation comparable to the Level 1 BLAS operation axpy, i.e. computing a vector-scalar product and adding the result to another vector (Blackford et al., 2002).

ESR for CA-PCG uses the same steps as ESR for PCG, plus reconstructs the unpreconditioned search direction. Therefore, they are almost as expensive and the cost of ESR for CA-PCG is independent of s.

Since ESR for PCG3 recovers the lost parts of the vectors of the last two iterations, it is more expensive than ESR for PCG. ESR for CA-PCG3 using Method 2 reconstructs the state of a PCG3 iteration instead of an outer CA-PCG3 iteration. The lost parts of $R^{(k-1)}$ and $U^{(k-1)}$ are not reconstructed, hence the cost of reconstruction does not increase with s. The difference to the cost of ESR for PCG3 in Table 5.1 arises from the fact that we assume that resilient CA-PCG3 using Method 2 stores redundant copies of $r^{(sk-1)}$ instead of $u^{(sk-1)}$.

The cost of ESR for CA-PCG3 using Method 1 or 3 increases with s since the lost parts of $R^{(k-1)}$ and $U^{(k-1)}$ are recovered. In ESR for CA-PCG3 using Method 1, the replacement nodes retrieve redundant copies of the lost parts of $u^{(sk)}$, $u^{(sk-s)}$ and of the last $s-1$ columns of $R^{(k-1)}$. The lost parts of the last $s-1$ columns of $U^{(k-1)}$ are recovered with preconditioner applications.

Method 3 recovers the lost parts of $R^{(k-1)}$ and $U^{(k-1)}$ using the three-term recurrence relation of the unpreconditioned residual in Equation (5.7) or its rearranged version in Equation (5.8), depending on whether $r^{(sk-s-1)}$ or $r^{(sk-1)}$ is stored redundantly. In Table 5.1, the cost of the first case is listed, which is slightly more expensive since in the second case the last column of $R^{(k-1)}$ is stored redundantly and its lost parts do not need to be reconstructed.

In Method 3, not all columns of $U^{(k-1)}_{I\setminus I_f}$ have to be gathered on the replacement nodes in line 2 of Algorithm 11. When Equation (5.7) is used, the first $s-1$ columns are needed, while the last $s-2$ columns are needed when using Equation (5.7).

Cost of recomputing the lost data of the current outer iteration after ESR

Since no communication occurs in the inner iterations, ESR for CA-PCG and CA-PCG3 using Method 1 and 3 recover the state at the beginning of the current outer iteration. ESR for CA-PCG3 using Method 2 recovers a subset of this state that enables the algorithm to execute s PCG3 iterations instead of the restarted outer CA-PCG3 iteration.

Node failures are typically detected during communication operations (Losada et al., 2020). In resilient CA-PCG and CA-PCG3, this can either be the communication phase(s) of the Matrix Powers Kernel or the global reduction operation for computing the Gram matrix. If the node failures happen after the global reduction operation, they are detected at the beginning of the next outer iteration during the first communication phase of the Matrix Powers Kernel. In this case, the whole outer iteration must be recomputed.

Recovering a PCG3 iteration is less expensive than recovering an outer CA-PCG3 iteration, see Table 5.1. However, computing s PCG3 iterations is more expensive than one outer CA-PCG3 iteration in terms of communication. To obtain a fair comparison, we also consider the cost of the restarted outer iteration respectively of the s PCG3 iterations, which is summarized in Table 5.2.

We assume that the monomial basis and MPK-PA1 are used. If another basis is used, the amount of local vector updates increases. When using MPK-PA0, s communication phases during the Matrix Powers Kernel are necessary, i.e. the number of global vectors to be gathered increases. When using MPK-PA2, locally computable parts of the basis matrices are communicated, i.e. more than one vector is communicated during the Matrix Powers Kernel, see Section 3.6. However, MPK-PA2 has only one communication phase and the same amount of data is communicated as in MPK-PA1 (Demmel et al., 2008). Therefore, the cost when using MPK-PA2 is not larger than the cost shown in Table 5.2.

We first consider resilient CA-PCG and CA-PCG3 using Method 1 or 3. At the beginning of an outer iteration, the basis matrices are computed. If the node failures happened

Table 5.2.: Cost of the restarted outer iteration respectively of the s PCG3 iterations after ESR in the worst case if the monomial basis and MPK-PA1 are used. The cost arising from redundant local computation during MPK-PA1 is neglected.

	CA-PCG	CA-PCG3	
operation		M1/M3	M2
global vectors gathered	2	1	$2s$
local vector updates	$10s+1$	$4s^2+6s$	$6s$
applying $A_{I_f,I}$	$2s-1$	s	s
applying $P_{I_f,I}$	$2s-1$	s	s
global reduction operations	1	1	s

during or after this step, the parts of the basis matrices owned by the failed nodes that have already been computed must be recomputed by the replacement nodes.

In Table 5.2, we summarize the operations for the restarted outer iteration when MPK-PA1 is used. We assume that the sparsity structures of A and P are suitable for MPK-PA1, i.e. communication between neighbour nodes during MPK-PA1 does not increase significantly with s and has the same asymptotic latency cost as a single matrix-vector product and preconditioner application, see Section 3.6 as well as Demmel et al. (2007) and Hoemmen (2010).

Moreover, we assume that the redundant computations in MPK-PA1 can be neglected. Thus, computing the basis matrices is as computationally expensive as $2s-1$ respectively s single matrix-vector products and preconditioner applications for CA-PCG respectively CA-PCG3.

In the restarted outer iteration, resilient CA-PCG gathers the global vectors $p^{(sk)}$ and $u^{(sk)}$ while resilient CA-PCG3 gathers $u^{(sk)}$. The surviving nodes still hold the redundant vector copies stored for their neighbours. Thus, we only have to communicate the redundant vector copies as described in Section 5.4.1 that where stored by the failed nodes.

After the computation of the basis matrices, the Gram matrix $G^{(k)}$ is formed. In both CA-PCG and CA-PCG3, this matrix is of size $(2s+1) \times (2s+1)$, its elements have the same values on all nodes. Thus, it can be retrieved together with the scalars from any surviving node if the node failures happened after the completion of the reduction operation for $G^{(k)}$.

If the node failures happen before the computation of $G^{(k)}$ is started, no overhead arises from the computation of $G^{(k)}$ in the restarted outer iteration. In the worst case, the reduction operation is almost finished at the time of the node failures. Then, the lost local scalar products that have been computed on the failed nodes have to be recomputed by the replacement nodes, followed by a global reduction operation.

In resilient CA-PCG, we neglect the cost of the inner iterations since their operations only involve small vectors of length $2s+1$ and small matrices of size $(2s+1) \times (2s+1)$. Applying the basis matrices to these small vectors at the end of an outer iteration in lines 15 to 19 of Algorithm 5 is as expensive as $5s+1$ local vector updates.

For resilient CA-PCG3 using Methods 1 and 3, we neglect the computation of the small vectors $d^{(sk+j)}$ and $g^{(sk+j)}$ of length $2s+1$ as well as the local computation of the scalars in the inner iterations. Computing the results of the matrix-vector products

Table 5.3.: Cost of total recovery: cost of ESR in Table 5.1 and cost of the restarted outer iteration respectively of the s PCG3 iterations after ESR in Table 5.2. The same assumptions are used as in Tables 5.1 and 5.2.

	CA-PCG	CA-PCG3		
operation		M1	M2	M3
global vectors gathered	5	$s+4$	$2s+4$	$2s+3$
redundant vector copies retrieved	2	$s+1$	2	3
local vector updates	$10s+5$	$4s^2+6s+6$	$6s+5$	$4s^2+8s+4$
applying $A_{I_f,I}$ or $A_{I_f,I\setminus I_f}$	$2s$	$s+2$	$s+2$	$2s+1$
applying $P_{I_f,I}$ or $P_{I_f,I\setminus I_f}$	$2s+1$	$2s+1$	$s+2$	$2s+1$
solving small systems with A_{I_f,I_f}	1	2	2	2
solving small systems with P_{I_f,I_f}	2	2	1	2
global reduction operations	1	1	s	1

and preconditioner applications without communication using $d^{(sk+j)}$ and $g^{(sk+j)}$ are as expensive as $2 \cdot 2s$ local vector updates in each inner iteration. Additionally, 6 local vector updates are performed at the end of each inner iteration for the three-term recurrence updates in lines 19 to 21 of Algorithm 4.

Resilient CA-PCG3 using Method 2 does not recompute the basis matrices or the Gram matrix. Instead, s iterations of PCG3 are executed. Thus, this method is the only one in Table 5.2 that requires more than one additional global reduction operation in the worst case. If $G^{(k)}$ has already been computed before the node failures happened, it can be retrieved from any surviving node. In this case, the scalars of the s iterations can be computed using $d^{(sk+j)}$ and $g^{(sk+j)}$, whose computation is negligible and does not involve communication.

The two global vectors for resilient CA-PCG in Table 5.2 are gathered together and therefore require the latency cost of gathering a single global vector. Resilient CA-PCG3 using Method 2 has to gather the $2s$ global vectors for the s matrix-vector products and preconditioner applications individually.

The worst case for resilient CA-PCG and CA-PCG3 using Method 1 and 3 occurs if the whole outer iteration must be recomputed. However, for resilient CA-PCG3 using Method 2, s global reduction operations must be executed if the Gram matrix has not yet been computed at the time of the node failures. Since global communication is considered a major bottleneck of performance in Conjugate Gradient methods, this case can also be considered as the worst case for resilient CA-PCG3 using Method 2.

Cost of total recovery

Table 5.3 summarizes the operations needed for the total recovery consisting of the cost of ESR and the overhead for the restarted outer iteration respectively the s PCG3 iterations in the worst case. We assume that MPK-PA1 and the monomial basis are used.

The redundant copies of multiple vectors can be sent together during ESR (see row 2 of Tables 5.1 and 5.3), leading to the same latency cost as for the copies of a single vector. During ESR, the global vectors that are gathered on the replacement nodes can be communicated together (see row 1 of Table 5.1), which requires the same latency cost as gathering a single global vector. Thus, the communication latency cost of ESR is independent of s for all considered methods.

This also applies to the global vectors to be gathered in the restarted CA-PCG or CA-PCG3 iteration after ESR in Table 5.2. The exception is the recovery of CA-PCG3 using Method 2, where $2s$ global vectors are gathered in the s PCG3 iterations after ESR. Since these vectors cannot be communicated together, communication latency of s matrix-vector products and preconditioner applications is necessary.

Therefore, in terms of communication latency, CA-PCG3 using Method 2 is the most expensive recovery method in Table 5.3. Moreover, it is the only method that requires s global reduction operations in the worst case.

Unlike the cost of ESR for CA-PCG in Table 5.1, the cost of the total recovery of resilient CA-PCG increases with s in Table 5.3. However, the cost of communication for recovery is still independent of s.

Resilient CA-PCG3 using Method 1 has the most expensive overhead in the failure-free case among all considered methods since $\mathcal{O}(s)$ vectors are communicated during the Matrix Powers Kernel in each outer iteration, see Section 5.4.1. In case of node failures, it is the least expensive method for recovery of CA-PCG3. Its ESR is less expensive than ESR for CA-PCG3 using Method 3, and communication latency during recovery of CA-PCG3 using Method 1 is less expensive than when using Method 2.

Resilient CA-PCG3 using Method 3 has the least expensive overhead among resilient CA-PCG and resilient CA-PCG3 in the failure-free case. However, in case of node failures, it requires the most matrix-vector products and preconditioner applications among all considered methods. Thus, it is the most expensive in terms of local computation during recovery of CA-PCG3. In this regard, resilient CA-PCG and CA-PCG3 using Method 3 are almost equally expensive.

CA-PCG computes $2s-1$ matrix-vector products and preconditioner applications in each outer iteration in contrast to PCG, PCG3 and CA-PCG3, which compute only s in each outer iteration respectively in s iterations. Therefore, an outer iteration of CA-PCG is more expensive than an outer iteration of CA-PCG3 in terms of local computation. However, it is assumed that the cost of this local computation does not significantly degrade performance in CA-PCG (Toledo, 1995). This also applies to the restarted outer iteration for CA-PCG in Table 5.2 and consequently also to the matrix-vector products and preconditioner applications during recovery of CA-PCG3 using Method 3 in Table 5.3.

The cost of ESR for CA-PCG3 using Method 2 is the least expensive in Table 5.1 among the costs of ESR for CA-PCG and CA-PCG3. Moreover, it is independent of s. However, in Table 5.3, most steps are comparable to the other methods. In the s PCG3 iterations after ESR for CA-PCG3 using Method 2, communication for s single matrix-vector products and preconditioner applications is necessary. If the Gram matrix has not been computed yet when the node failures occur, s global reduction operations are performed after ESR. In this worst case, to avoid this global communication overhead, we could alternatively reconstruct the lost parts of $R^{(k-1)}$ by using the rearranged three-term recurrence relation in Equation (5.8) as in Method 3.

As with Table 5.2, if MPK-PA2 is used instead of MPK-PA1, the total cost of recovery is not larger than the cost shown in Table 5.3. Thus, recovery of resilient CA-PCG3 using Method 2 remains the most expensive in terms of communication latency.

The communication-avoiding s-step methods CA-PCG and CA-PCG3 are able to reduce both global and local communication. In cases where it suffices to reduce communication of the global scalar products or if the system matrix is not suitable for MPK-PA1 or MPK-PA2, MPK-PA0 can be used. In this case, the restarted outer iteration of resilient CA-PCG and CA-PCG3 using Method 1 or 3 in Table 5.2 needs communication for $2s + 1$ respectively s single matrix-vector multiplications and preconditioner applications. Therefore, latency cost of gathering the global vectors in Table 5.3 increases with s for all methods. In this case, recovery of CA-PCG3 using Method 2 is the least expensive among all methods in Table 5.3 if $G^{(k)}$ has already been computed before the node failures.

5.4.3. Summary of the theoretical evaluation

In the failure-free case, resilient CA-PCG and resilient CA-PCG3 using Methods 1, 2 and 3 have the same communication latency overhead. Resilient CA-PCG3 using Method 1 is the only algorithm whose message sizes increase with s since redundant copies of the last $s - 1$ columns of $R^{(k-1)}$ are communicated. Thus, this is the algorithm with the largest overhead in the failure-free case. Since we assume that node failures are rare, we aim to reduce the overhead in the failure-free case rather than the overhead of recovery. Therefore, resilient CA-PCG and CA-PCG3 using Method 1 or 3 are more suitable for higher values of s. Resilient CA-PCG3 using Method 3 has the smallest message sizes and therefore the smallest overhead in the failure-free case.

Resilient CA-PCG3 using Method 1 has the least expensive recovery when using MPK-PA1 or MPK-PA2. Its ESR is less expensive than ESR for CA-PCG3 using Method 3. Less local computation is necessary than for recovery of resilient CA-PCG. Communication latency during recovery of resilient CA-PCG3 using Method 1 is less expensive than when Method 2 is used.

In terms of communication latency, resilient CA-PCG3 using Method 2 has the most expensive recovery if MPK-PA1 or MPK-PA2 are used. However, if MPK-PA0 is used and $G^{(k)}$ has already been computed before the node failures, resilient CA-PCG3 using Method 2 has the least expensive recovery. While resilient CA-PCG3 using Method 3 has the smallest overhead in the failure-free case, it is the most expensive in terms of local computation during recovery of resilient CA-PCG3.

6. Experimental evaluation

In this chapter, we describe our implementation and the experimental evaluation of resilient CA-PCG and CA-PCG3. We illustrate the scalability of both the non-resilient and resilient versions of the two algorithms and their basis algorithms PCG and PCG3. Moreover, we show that the speedup with respect to s is similar for the non-resilient and resilient versions. Furthermore, we show the relative overhead for resilience in CA-PCG and CA-PCG3 compared to the respective non-resilient algorithms both in the failure-free case and when failures occur.

6.1. Implementation details and experimental setup

We implemented the algorithms in C++ using the GNU Scientific Library (GSL) and MPI. We used GSL 2.5, Intel MPI 2019 Update 7 and compiled with the Intel C++ compiler 19.1.3 with compiler flag `-O3`. The computational results presented have been achieved using the Vienna Scientific Cluster (VSC4).

If a node fails, all of its data is lost and has to be recovered on a replacement node. The processes of a node share their memory. If a process failure occurs, a replacement process, which can be either a spare process or one of the surviving processes on the same node, takes over the work of the failed process. Since the shared memory stayed intact if at least one process on this node survived, the replacement process does not have to recover any data (Pachajoa et al., 2019).

On large-scale parallel computers, communication becomes a major bottleneck. The additional local computation in CA-PCG and CA-PCG3 compared to their basis algorithms PCG and PCG3 is usually negligible, see Chapter 3. To see this effect in our experiments, we increased the number of processes by using multiple processes per node.

In Chapters 4 and 5, we assumed that a parallel runtime environment that supports fault-tolerance features is available, such as the MPI extension ULFM (Bland et al., 2013; Message Passing Interface Forum, 2021b), see Section 1.3. In our experiments, we simulate failures instead of taking down nodes or processes. The likelihood of failures increases with the number of nodes or processes (Herault and Robert, 2015). Since we used far more processes than nodes, we simulated process failures instead of node failures to simulate a more realistic scenario. We treated these simulated process failures as node failures, i.e. all data of a failed process is set to zero and reconstructed on a replacement process.

If failures happen before the computation of the Gram matrix, the overhead for recovery of resilient CA-PCG3 using Method 2 includes s global reduction operations. In Section 5.4.2, we suggest to instead reconstruct the lost parts of $R^{(k-1)}$ using the rearranged three-term recurrence relation in Equation (5.8) as in Method 3 to avoid the overhead of s global communication operations.

We therefore do not show experiments for this worst case of resilient CA-PCG3 using Method 2. Instead, we simulated failures at the end of an outer iteration of resilient

CA-PCG and CA-PCG3 using Method 1, 2 and 3. Thus, the entire outer iteration must be recomputed, see Section 5.4.2.

We introduced process failures in iteration 300 of resilient PCG and PCG3 respectively at the end of outer iteration $\lfloor 300/s \rfloor$ of resilient CA-PCG and CA-PCG3. We simulated a single process failure on rank 21 and multiple process failures on ranks 21, 22, and 24.

We chose the solution x with entries $1/\sqrt{n}$, where n is the problem size, and used a zero vector as the initial guess $x^{(0)}$. The convergence criterion for PCG, PCG3, CA-PCG and CA-PCG3 is the reduction of the relative residual norm by a factor of 10^8 or 10^{10}, depending on the convergence rate of the solvers for the respective matrix. While we used 10^8 for the matrices in Table 6.1 obtained from the SuiteSparse Matrix Collection (Davis and Hu, 2011), 10^{10} was used for rand_n100000_b2, a generated matrix for which the solvers converged significantly faster than for the other matrices. When solving the small linear systems during recovery, we terminated the solver after the relative residual norm had been reduced by a factor of 10^{11}. In our experiments, we did not observe that solving these small systems had a significant impact on the convergence rate.

We used the monomial basis in CA-PCG and CA-PCG3, see Section 3.1, whose accuracy was sufficient for $s \leq 6$ in our experiments. The preferred types of preconditioners in communication-avoiding Krylov subspace methods are preconditioners that only need communication with local neighbours to preserve scalability (Hoemmen, 2010; Cools et al., 2019). We used the Jacobi preconditioner, whose application does not involve communication.

In resilient CA-PCG3 using Method 1, the last $s - 1$ columns of $R^{(k-1)}$ are stored redundantly. In our implementation, we chose to send all columns of $R^{(k-1)}$ along with $u^{(sk)}$ since this leaves the latency cost unchanged, but avoids copying columns of $R^{(k-1)}$. For resilient CA-PCG3 using Methods 2 and 3, we chose to communicate redundant copies of $r^{(sk-1)}$ instead of $u^{(sk-1)}$, see Section 5.3.2. Since we used the monomial basis, we did not explicitly store the unpreconditioned search direction $q^{(sk)}$ in CA-PCG. Therefore, we did not have to recover its lost parts after failures.

The matrices are block-row distributed and the vectors are distributed accordingly. To compute the basis matrices in CA-PCG and CA-PCG3, we used the Matrix Powers Kernel algorithms MPK-PA0 or MPK-PA1.

6.2. Test data

The matrices used in the experiments are listed in Table 6.1. LFAT5000 and apache2 are obtained from the SuiteSparse Matrix Collection (Davis and Hu, 2011). rand_100000_b2 is a randomly generated SPD pentadiagonal matrix, i.e. all matrix elements are zero except on the main diagonal and the first two upper and two lower diagonals.

For matrices with special structures, e.g. band matrices, MPK-PA1 has the same asymptotic latency cost as a single matrix-vector product and preconditioner application at the expense of redundant computation, see Section 3.6 as well as Demmel et al. (2007) and Hoemmen (2010). When using band matrices, the higher the bandwidth and the number of non-zeros of the system matrix and the preconditioner, the more redundant local computation must be performed during MPK-PA1. On large-scale parallel computers, this redundant computation does not have a significant impact on the performance since communication becomes a major bottleneck (Demmel et al., 2007; Carson, 2018).

Table 6.1.: Test matrices generated or obtained from the SuiteSparse Matrix Collection (Davis and Hu, 2011). NNZ: Number of non-zero entries.

Matrix name	Size	NNZ	Description/Problem type
LFAT5000	19994	79966	Model Reduction (Davis and Hu, 2011)
rand_n100000_b2	100000	499994	generated random SPD pentadiagonal matrix
apache2	715176	4817870	Structural (Davis and Hu, 2011)

Since we used up to 3072 processes respectively 64 nodes, local computation can have a significant impact on runtimes in our experiments. This is especially critical for CA-PCG, which performs almost twice as many matrix-vector multiplications and preconditioner applications as CA-PCG3.

The matrices listed in Table 6.1 have small numbers of non-zeros such that CA-PCG3 does not have a significant performance advantage over CA-PCG regarding local computation. We chose these matrices so that the absolute runtimes of the algorithms would not distort the comparison of the relative overheads for resilience in Section 6.4.

The matrices LFAT5000 and rand_n100000_b2 have small bandwidths and therefore fulfill the requirements of MPK-PA1 regarding the sparsity structure. The matrix apache2 has a larger bandwidth, therefore experiments for this matrix were executed using MPK-PA0.

The selection of neighbours that store the redundant copies of a node respectively process as suggested by Pachajoa et al. (2019) and described in Section 4.2 is suitable for matrices whose entries are mostly clustered around the diagonal. Thus, this strategy is suitable for the band matrices in Table 6.1.

6.3. Scalability and speedup with respect to the step size

In this section, we show that scalability of CA-PCG and CA-PCG3 is preserved when incorporating resilience. Moreover, we show that the speedups of non-resilient and resilient CA-PCG and CA-PCG3 with respect to s are similar.

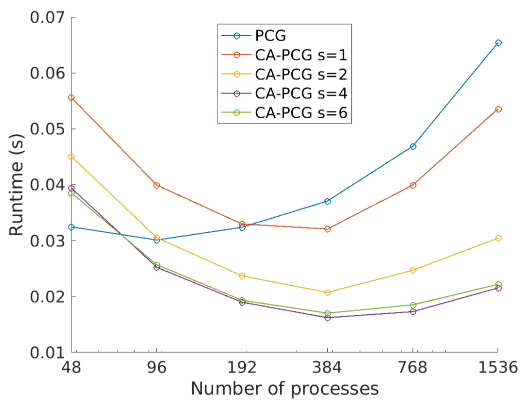
For each algorithm, we used the median of the runtimes of three test runs. For the resilient failure-free case, data redundancy to be able to tolerate three simultaneous process failures is ensured, but no process failures occur. For the case with failures, we simulated three simultaneous process failures.

We used 48 processes per node and up to 32 nodes. We terminated the algorithms once the relative residual had been reduced by a factor of 10^8.

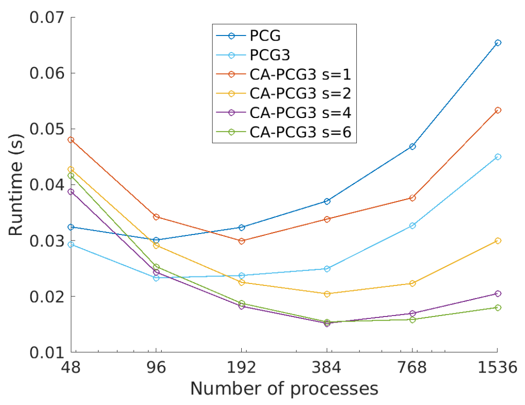
Non-resilient case

Figures 6.1a to 6.1c show the runtimes of the non-resilient algorithms for LFAT5000 for different values of s. For smaller numbers of processes, CA-PCG and CA-PCG3 are significantly slower than PCG and PCG3 since CA-PCG and CA-PCG3 perform more local computation than PCG and PCG3, e.g. the computation of the Gram matrix.

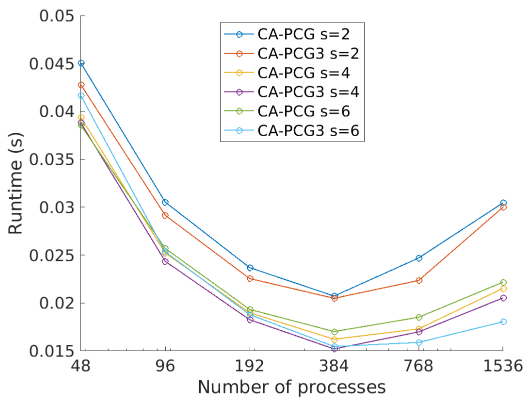
When using more than 384 processes, the runtimes increase for all algorithms and values of s. For these process amounts, communication becomes the dominant factor in



(a) Non-resilient PCG and CA-PCG



(b) Non-resilient PCG, PCG3 and CA-PCG3



(c) Non-resilient CA-PCG and CA-PCG3

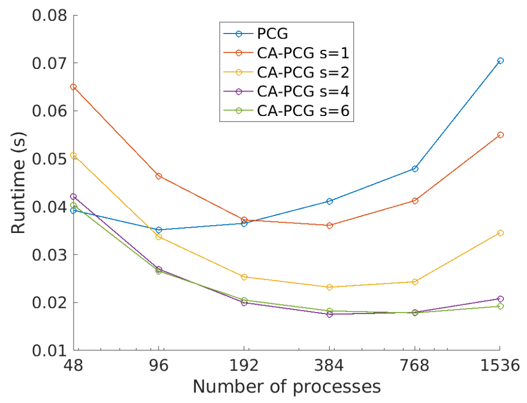
Figure 6.1.: Runtimes of the non-resilient algorithms for the matrix LFAT5000. Median runtimes of three test runs for each algorithm and value of s, using the Jacobi preconditioner and MPK-PA1. 48 processes per node and up to 32 nodes were used. The algorithms were terminated once the relative residual had been reduced by a factor of 10^8.

runtime for this matrix. The algorithms can no longer scale because the proportion of local computation is too small, even for higher values of s.

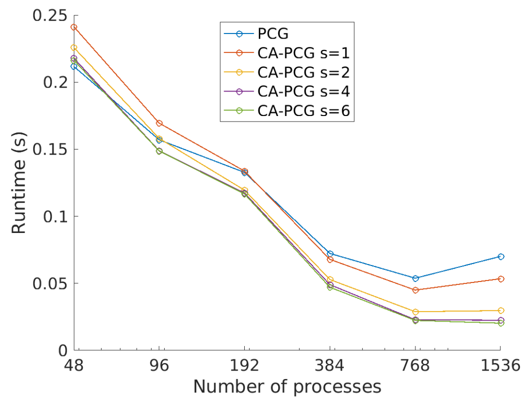
PCG performs two global reductions in each iteration while PCG3 is able to combine its two scalar products in one global reduction, making PCG3 a communication-avoiding algorithm and thus faster than PCG, see Section 2.3. CA-PCG and CA-PCG3 perform one global reduction per outer iteration and therefore show improvement over PCG even in the case $s = 1$ for larger amounts of processes since only half the number of global reduction operations is performed.

Since CA-PCG, CA-PCG3 and PCG3 have the same amount of global reduction operations if $s = 1$, CA-PCG and CA-PCG3 only have a performance advantage over PCG3 if $s > 1$. Their runtimes in the case $s = 1$ are larger than the runtimes of PCG3 since they have to perform more local computation, e.g. for the Gram matrix.

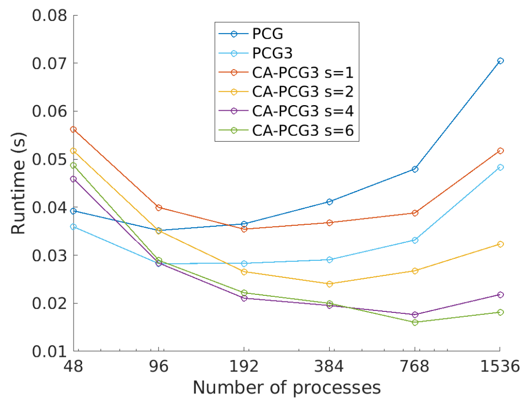
CA-PCG performs $2s-1$ and CA-PCG3 only s matrix-vector products and preconditioner applications in each outer iteration. Therefore, CA-PCG3 is slightly faster than CA-PCG



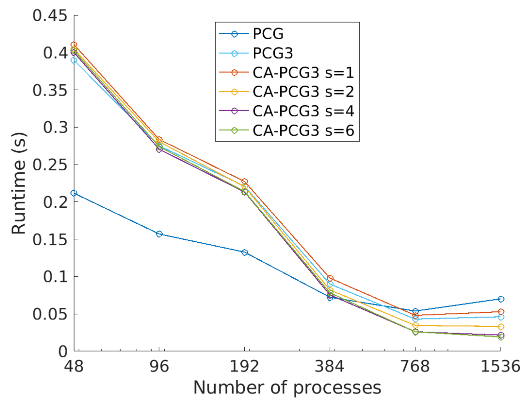
(a) Resilient PCG and CA-PCG in the failure-free case



(b) Resilient PCG and CA-PCG with three simultaneous process failures



(c) Resilient PCG, PCG3 and CA-PCG3 using Method 1 in the failure-free case



(d) Resilient PCG, PCG3 and CA-PCG3 using Method 1 with three simultaneous process failures

Figure 6.2.: Runtimes of the resilient algorithms for the matrix LFAT5000. The algorithms were executed using the same conventions and settings as in Figure 6.1. For the resilient failure-free case, data redundancy to be able to tolerate three simultaneous process failures is ensured, but no process failures occur. For the case with failures, three simultaneous process failures are simulated in iteration 300 respectively at the end of outer iteration $\lfloor 300/s \rfloor$ on rank 21, 22 and 24.

in Figure 6.1c, especially for $s = 6$. CA-PCG with $s = 6$ is not able to improve performance over CA-PCG with $s = 4$.

Resilient case

Figures 6.2a to 6.2d show the runtimes for resilient PCG, PCG3, CA-PCG and CA-PCG3 using Method 1. Data redundancy to be able to tolerate three simultaneous process failures is ensured. The figures show the failure-free case and when three simultaneous process failures occur. Since the figures for resilient CA-PCG3 using Method 1, 2 and 3 are similar, we show the runtimes of resilient CA-PCG3 using Methods 2 and 3 in Appendix A in

Figures A.1a to A.1d.

The resilient algorithms in the failure-free case in Figures 6.2a and 6.2c show similar behaviour in terms of scalability as the non-resilient algorithms in Figures 6.1a and 6.1b. The case of process failures shown in Figures 6.2b and 6.2d has significant larger runtimes for smaller amounts of processes since the proportion of failed processes is smaller for higher amounts of processes in these figures. However, the algorithms still show similar behaviour as in the failure-free and in the non-resilient case, e.g. PCG and CA-PCG with $s = 1$ have almost equal runtimes in Figures 6.1a, 6.2a and 6.2b when using 192 processes, and with higher process amounts, CA-PCG is faster than PCG for all values of s.

Recovery after process failures is more expensive for PCG3 and CA-PCG3 than for PCG and CA-PCG for smaller amounts of processes, see Figures 6.2b and 6.2d. PCG and CA-PCG are based on two-term recurrences while PCG3 and CA-PCG3 are based on three-term recurrences. In the latter case, not only the vectors of the last but also of the second last iteration must be reconstructed to be able to continue the three-term recurrence vector updates after recovery.

With increasing numbers of processes, the data owned by the three failed processes becomes a smaller proportion of the state, and the cost of recovery decreases. For high process numbers, PCG3 is faster than PCG as in the failure-free and the non-resilient case. When using 1536 processes, the runtimes of all algorithms in Figures 6.2b and 6.2d are similar to the respective failure-free and non-resilient versions.

In contrast to non-resilient CA-PCG in Figure 6.1a, resilient CA-PCG with $s = 6$ improves performance over smaller values of s in Figure 6.2a since the overhead for data redundancy decreases with s, which is shown in Section 6.4.

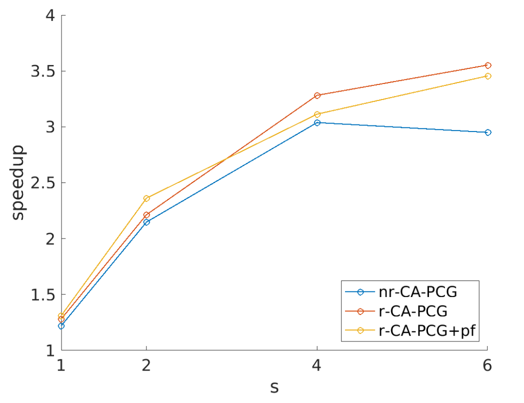
Speedup

Figures 6.3a and 6.3b show the speedup of the different versions of CA-PCG and CA-PCG3 over the respective version of PCG for 1536 processes, i.e. the speedup of non-resilient CA-PCG and CA-PCG3 over non-resilient PCG, the speedup of resilient CA-PCG and CA-PCG3 over resilient PCG in the failure-free case, etc. All algorithms are able to improve performance with increasing values of s, except for non-resilient CA-PCG with $s = 6$, which is consistent with Figure 6.1a. The resilient algorithms both in the failure-free case and in the case where three process failures happen, have similar speedups as the non-resilient algorithms.

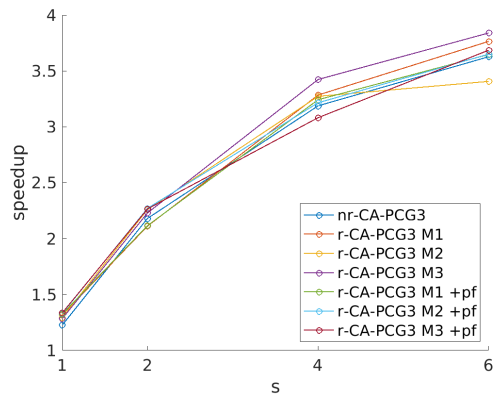
The runtime of non-resilient PCG with 1536 processes is approximately twice as large as the runtime with 96 processes, see Figure 6.1a. Thus, we show the speedup using the best runtime over all numbers of processes for each algorithm in Figures 6.3c and 6.3d. The versions with process failures have a higher speedup since resilient PCG with process failures does not have smaller runtimes when using fewer processes.

6.4. Relative runtime overheads for resilience

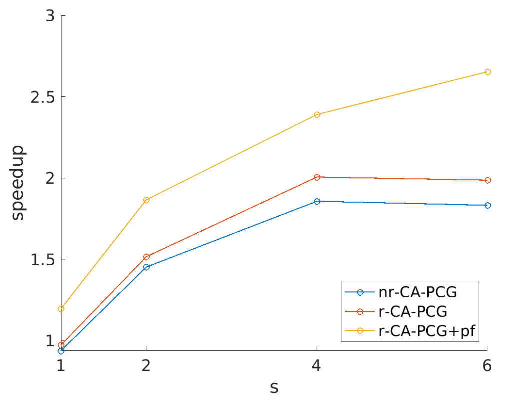
In this section, we show the relative runtime overheads for resilience in CA-PCG and CA-PCG3. We used the median runtimes over three test runs of non-resilient CA-PCG and CA-PCG3 for each value of s as baselines. We executed three test runs for the resilient failure-free case, i.e. data redundancy to be able to tolerate ϕ simultaneous process failures



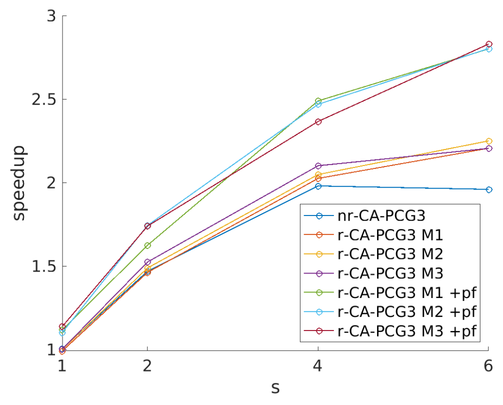
(a) Speedup of CA-PCG for 1536 processes



(b) Speedup of CA-PCG3 for 1536 processes



(c) Speedup of CA-PCG using the best runtime over all process amounts for each algorithm version and value of s in Figures 6.1 and 6.2

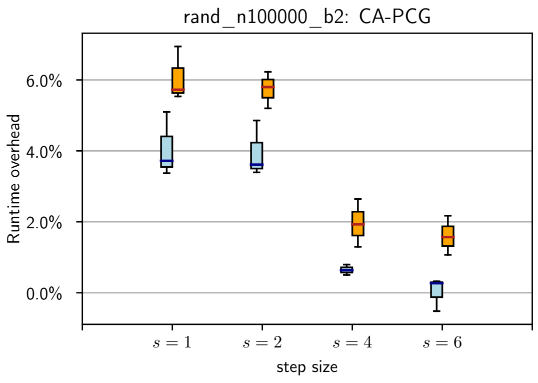


(d) Speedup of CA-PCG3 using the best runtime over all process amounts for each algorithm version and value of s in Figures 6.1 and 6.2

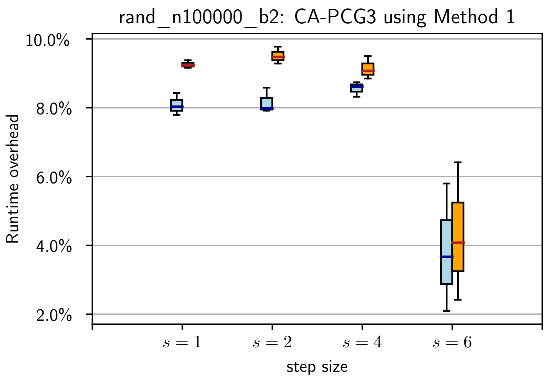
Figure 6.3.: Speedup of the runtimes shown in Figures 6.1 and 6.2 with respect to the step size s for the matrix LFAT5000. The subfigures show the speedup of the different versions of CA-PCG and CA-PCG3 over the respective version of PCG, i.e. the speedup of non-resilient CA-PCG (nr-CA-PCG) and CA-PCG3 (nr-CA-PCG3) over non-resilient PCG, the speedup of resilient CA-PCG (r-CA-PCG) and CA-PCG3 (r-CA-PCG3) over resilient PCG in the failure-free case, and the speedup in the case of three simultaneous process failures ("+pf"). The three different methods of resilient CA-PCG3 are indicated with "M1", "M2" and "M3".

is ensured, but no failures occur. Finally, we executed three test runs with ϕ simulated process failures.

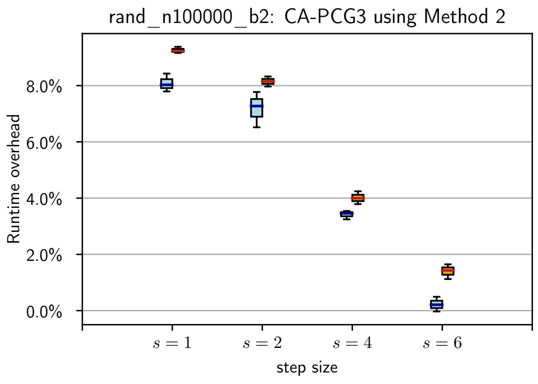
In Figures 6.4, 6.5 and 6.6, the blue boxplots (to the left of a group) represent the relative overheads in the failure-free case, i.e. the overhead for data redundancy. The orange boxplots (to the right of a group) represent the relative overheads in the case of ϕ simultaneous process failures.



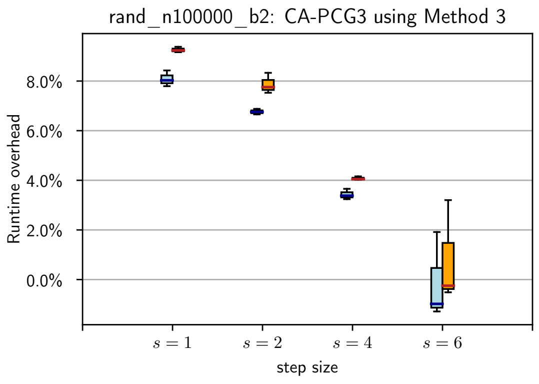
(a) CA-PCG



(b) CA-PCG3 using Method 1



(c) CA-PCG3 using Method 2



(d) CA-PCG3 using Method 3

Figure 6.4.: Relative overheads for resilience for the matrix rand_n100000_b2, using the Jacobi preconditioner and MPK-PA1. 48 processes per node and 64 nodes were used. The algorithms were terminated once the relative residual had been reduced by a factor of 10^{10}. The baseline for each algorithm and value of s, with respect to which the relative overheads are measured, is the median runtime of three test runs for non-resilient CA-PCG respectively CA-PCG3 for the respective value of s. The blue boxplots (to the left of a group) are the relative overhead of three test runs in the failure-free case for ensuring data redundancy to be able to tolerate $\phi = 1$ process failure. The orange boxplots (to the right of a group) indicate the relative overheads of three test runs in the case where $\phi = 1$ process failure is simulated in iteration 300 respectively outer iteration $\lfloor 300/s \rfloor$ on rank 21.

rand_n100000_b2

In Figures 6.4a to 6.4d, the relative overheads for rand_n100000_b2 with $\phi = 1$ are shown. We executed the algorithms on 64 nodes with 48 processes per node, i.e. 3072 processes in total. Since the algorithms needed far fewer iterations until convergence for rand_n100000_b2 than for the other two matrices in Table 6.1, the solvers were terminated once the residual had been reduced by a factor of 10^{10}.

The relative overheads in the failure-free case decrease with increasing s. Redundant vector copies can only be communicated at the beginning of each outer iteration, i.e. only every s steps, leading to lower communication latency overheads for higher values of s. Thus, the communication-avoiding properties of CA-PCG and CA-PCG3 also apply to the communication overhead for data redundancy.

Moreover, at the beginning of each outer iteration, each process has to store its part of the current (and in CA-PCG3 also the last) approximate solution vector, leading to one respectively two additional local vector copy operations per outer iteration, i.e. every s steps, see Section 5.4.1. Similarly to the local computation of the additional matrix-vector products in CA-PCG (Toledo, 1995), we expect this local computation overhead to become negligible with larger numbers of nodes and processes. The overheads for CA-PCG3 are larger than for CA-PCG in the failure-free case since two instead of one approximate solution vectors are copied in each outer iteration.

While resilient CA-PCG and CA-PCG3 using Method 1, 2 and 3 have the same communication latency cost in the failure-free case, resilient CA-PCG3 using Method 1 is the only method discussed in this thesis whose amount of redundantly stored vectors and thus the size of messages increases with s, which can be observed in Figure 6.4b. For $s = 6$, redundant copies are only stored every 6 steps, which compensates the overhead due to the larger message sizes.

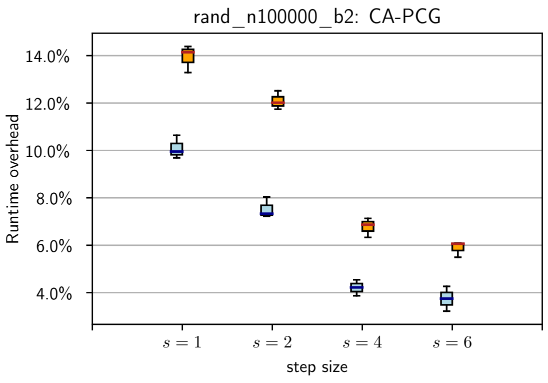
The difference between the blue and the orange boxplots is the time spent for recovery. The overhead for recovery is roughly the same for all algorithms and all values of s in Figures 6.4a to 6.4d. For resilient CA-PCG and CA-PCG3 using Method 2, the cost of reconstruction is independent of s, only the cost of the restarted outer iteration increases with s. For resilient CA-PCG3 using Method 1 or 3, the cost of reconstruction increases with s. However, our experiments show that this does not significantly impact the runtime overhead when using this matrix and this amount of processes.

The part of the total recovery cost that increases with s mainly consists of matrix-vector products and preconditioner applications for both resilient CA-PCG and CA-PCG3, see Table 5.3. Since we used the Jacobi preconditioner and rand_n100000_b2 has few nonzeros, we do not observe an increase in the recovery cost for higher values of s in Figures 6.4a to 6.4d.

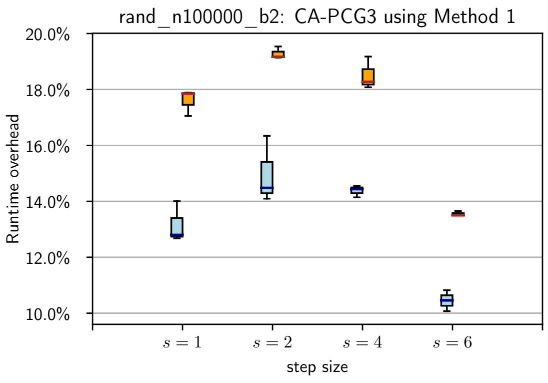
Figures 6.5a to 6.5d show the relative overheads for resilience for rand_n100000_b2 with $\phi = 3$ while using the remaining settings as in Figures 6.4a to 6.4d. Compared to the case $\phi = 1$, the overheads in the failure-free case are larger since more redundant copies are communicated and stored in each outer iteration. The cost of recovery (the difference between the blue and the orange boxplots) is more expensive since three times as much data is reconstructed. Moreover, while the small systems of equations are solved locally on a single replacement process if $\phi = 1$, they are solved in parallel on three replacement processes if $\phi = 3$.

Overall, the behaviours are similar for the cases $\phi = 1$ and $\phi = 3$ for this matrix. In both cases, the overheads in the failure-free case decrease with higher values of s, except for resilient CA-PCG3 using Method 1, whose overhead only decreases for $s = 6$. Moreover, the cost of recovery does not increase with s in both cases. In the case $\phi = 3$, the runtime overheads in the failure-free case are slightly smaller for resilient CA-PCG3 using Method 3 than when using Method 2 for $s > 1$. When using Method 2, redundant copies of two vectors are communicated in each outer iteration while on average only 1.5 vectors must be stored redundantly when using Method 3, see Section 5.4.1.

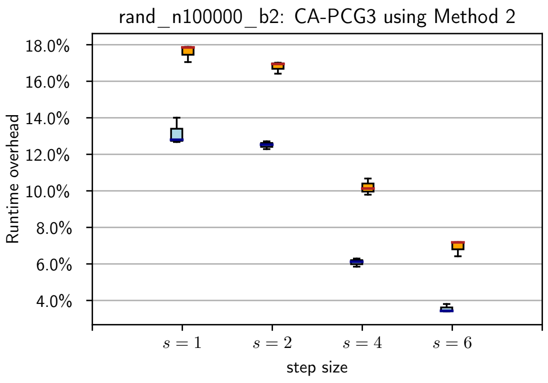
While the overheads for $\phi = 1$ are at most 10%, overheads for $\phi = 3$ for smaller values of



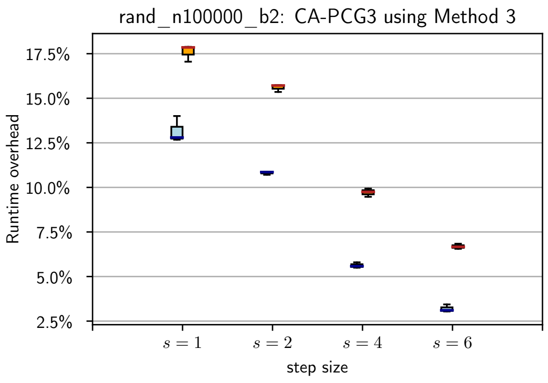
(a) CA-PCG



(b) CA-PCG3 using Method 1



(c) CA-PCG3 using Method 2



(d) CA-PCG3 using Method 3

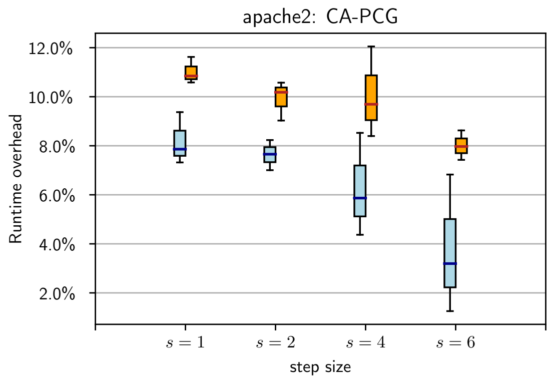
Figure 6.5.: Relative overheads for resilience for the matrix rand_n100000_b2 with $\phi = 3$ while otherwise using the same conventions and settings as in Figure 6.4. In the resilient failure-free case data redundancy to tolerate $\phi = 3$ simultaneous process failures is ensured. In the case of failures, process failures are simulated in iteration 300 respectively outer iteration $\lfloor 300/s \rfloor$ on rank 21, 22 and 24.

s are larger. For $s = 6$, all overheads are well below 10% with the exception of CA-PCG3 using Method 1 for $\phi = 3$.

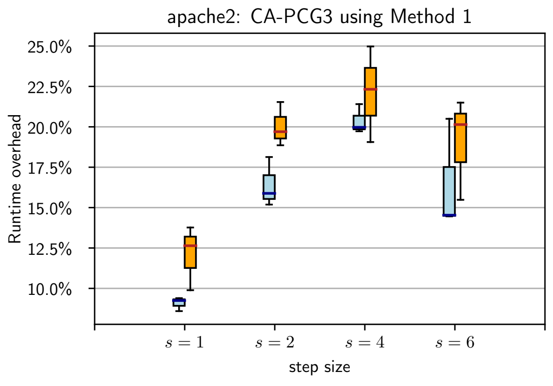
apache2

Figures 6.6a to 6.6d show the relative overheads for resilience for the matrix apache2 with $\phi = 3$. The solvers were terminated once the residual had been reduced by a factor of 10^8. The algorithms were executed on 32 nodes with 32 processes per node since PCG3 and CA-PCG3 were not able to converge for higher numbers of processes for this matrix.

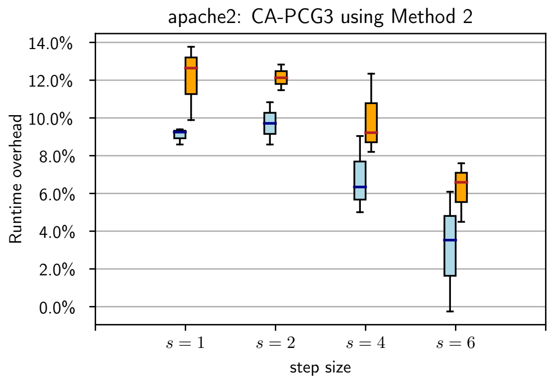
When executing PCG, PCG3, CA-PCG or CA-PCG3 in parallel, the local computations of the matrix-vector products, the preconditioner applications and the local vector update operations are distributed among processes. The computation of each element of the result vectors is computed as in the sequential case if the matrices are block-row distributed and the vectors are distributed accordingly. Thus, in finite-precision arithmetic, the number of processes does not influence the rounding error accumulation of these operations.



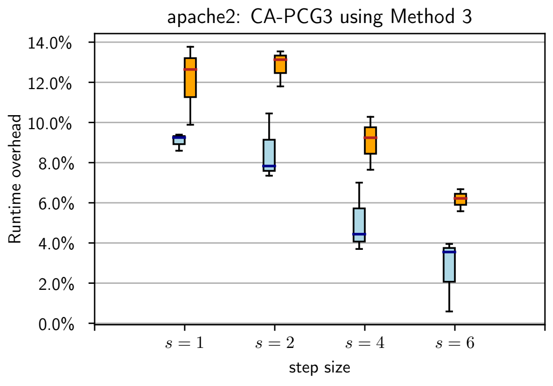
(a) CA-PCG



(b) CA-PCG3 using Method 2



(c) CA-PCG3 using Method 2



(d) CA-PCG3 using Method 3

Figure 6.6.: Relative overheads for resilience for the matrix apache2 with $\phi = 3$, using the Jacobi preconditioner and MPK-PA0. 32 processes per node and 32 nodes were used. The algorithms were terminated once the relative residual had been reduced by a factor of 10^8. The same conventions as in Figure 6.4 were used. Process failures are simulated in iteration 300 respectively outer iteration $\lfloor 300/s \rfloor$ on rank 21, 22 and 24.

However, the rounding error accumulation of the computation of the global scalar products respectively the Gram matrix differs depending on the number of processes.

Each process computes its part of the scalar products locally before the local results are summed up using a global reduction operation. Depending on the number of processes, the order of these additions vary. In exact arithmetic, the addition of scalars is commutative. However, in finite precision, different numbers of processes and thus different orders of the additions may lead to different rounding error accumulation. Since three-term recurrence relations are less accurate than two-term recurrence relations in Krylov subspace methods (Gutknecht and Strakos, 2000), this different rounding error accumulation affected the convergence behaviour of PCG3 and CA-PCG3 in our experiments with apache2 for higher numbers of processes.

In Figures 6.6a to 6.6d, resilient CA-PCG and CA-PCG3 show a similar behaviour for apache2 as for rand_n100000_b2. However, unlike for rand_n100000_b2, resilient CA-PCG3 using Method 1 with $s = 6$ is not able to sufficiently compensate the larger message sizes for apache2, indicating that this method is not suitable for this matrix if these values of s are used. Except for resilient CA-PCG3 using Method 1, all overheads are below 14%. Similar to rand_n100000_b2, the overheads for apache2 in the failure-free case are slightly smaller for resilient CA-PCG3 using Method 3 than when using Method 2.

6.5. Summary of the experimental evaluation

Our experiments showed that scalability of CA-PCG and CA-PCG3 is preserved when incorporating resilience. Moreover, the ability to reduce communication and thus runtime with increasing s is preserved.

Resilient CA-PCG and CA-PCG3 using Methods 1, 2 and 3 have the same communication latency cost in the failure-free case. Nevertheless, resilient CA-PCG3 using Method 1 showed significantly larger overheads in the failure-free case than the other methods. This is consistent with the theoretical evaluation in Section 5.4.1 since resilient CA-PCG3 using Method 1 is the only algorithm discussed in this thesis whose message sizes increase with s.

In our experiments, the overhead for recovery of resilient CA-PCG and CA-PCG3 did not increase with s. In Section 5.4.2 (see Table 5.3), we theoretically analyzed that recovery is less expensive for resilient CA-PCG3 using Method 1 than for resilient CA-PCG and CA-PCG3 using Methods 2 and 3 when using MPK-PA1 or MPK-PA2. Thus, we expect resilient CA-PCG3 using Method 1 to have a performance advantage over the other resilient communication-avoiding s-step methods when recovery is more expensive, i.e. when using matrices with more non-zeros than for the matrices in Table 6.1 and a more expensive preconditioner.

7. Conclusion

After reviewing the Preconditioned Conjugate Gradient (PCG) method as well as its three-term recurrence variant PCG3, we gave an overview of two existing communication-avoiding s-step Conjugate Gradient methods, CA-CG based on the unpreconditioned Conjugate Gradient (CG) method and CA-PCG3 based on PCG3. In both methods, the iteration loop of the respective basis algorithm is divided into an outer and an inner loop, where the inner loop has s iterations. Communication only happens at the beginning of each outer iteration, i.e. every s steps. To the best of our knowledge, an explicit formulation of a left-preconditioned version of CA-CG does not yet exist in literature. Therefore, we derived a communication-avoiding s-step method based on PCG (CA-PCG) analogously to CA-PCG3.

Next, we gave an overview of different existing approaches to make PCG resilient against node failures. We described in detail how this algorithm can be made fault-tolerant using inherent data redundancy of the matrix-vector product and how the lost data can be reconstructed exactly in case of node failures using Exact State Reconstruction (ESR).

Based on similar approaches as for resilient PCG, we developed novel resilient versions of CA-PCG and CA-PCG3. Resilient CA-PCG uses the inherent data redundancy of the vectors communicated for the matrix-vector products with little overhead to ensure enough data redundancy to tolerate node failures. To reconstruct the state after node failures, almost the same ESR method as for resilient PCG can be used.

Contrary to resilient CA-PCG, the amount of data that has to be recovered after node failures during resilient CA-PCG3 increases with s. Moreover, to be able to efficiently reconstruct the lost data, it is not sufficient to store redundant copies of the vectors that are communicated for the matrix-vector products in non-resilient CA-PCG3. Thus, additional vector parts are communicated along with the original vectors, which leaves the latency cost unchanged, but increases the message sizes, leading to more overhead even in the failure-free case.

Three different methods to ensure data redundancy and to reconstruct the lost data after node failures were presented for CA-PCG3. Method 1 stores the last $s + 1$ residual vectors redundantly in each outer iteration. Method 2 only stores the last two residuals redundantly and reconstructs the state of a PCG3 iteration instead of an outer CA-PCG3 iteration. Thus, s PCG3 iterations must be executed after reconstruction before the algorithm can be continued with CA-PCG3 iterations. Method 3 stores the last one or two residuals alternately and thus has the least overhead in the failure-free case. However, Method 3 has a more expensive reconstruction than Method 1 since the rest of the last $s + 1$ residuals must be reconstructed.

We theoretically and experimentally evaluated the cost of incorporating resilience in CA-PCG and CA-PCG3 both in the failure-free case and in case of failures. The communication latency overhead for data redundancy is independent of s for both CA-PCG and CA-PCG3. Therefore, the communication-avoiding properties and thus scalability of CA-PCG and CA-PCG3 are preserved. Our experiments showed that the step size does not have a significant impact on the recovery cost for the band matrices used.

Since resilient CA-PCG3 using Method 1 redundantly stores $s + 1$ vectors, its overhead for resilience increases with s even in the failure-free case. Resilient CA-PCG3 using Method 2 and 3 show only slightly larger overheads as resilient CA-PCG both in the failure-free case and when failures occur. With the exception of resilient CA-PCG3 using Method 1, all methods had small overheads for resilience (below 18%) for all test cases with $s \leq 6$, in the case $s = 6$, their overheads were below 10%.

Since node failures are rare, we aim to minimize the overhead in the failure-free case rather than the cost of recovery. An interesting future task would be to develop an alternative method for resilience in CA-PCG3 that only stores the vectors redundantly that are communicated for the matrix-vector products in non-resilient CA-PCG3. While this method minimizes the overhead in the failure-free case, its ESR algorithm is more complicated and expensive than for Methods 1, 2 and 3.

This thesis focused on resilience in communication-avoiding s-step methods. An interesting subject of future work would also be a comparison with resilience of the communication-hiding Conjugate Gradient algorithm with deep pipelines proposed by Cools et al. (2019).

Bibliography

Agullo, Emmanuel, Luc Giraud, Abdou Guermouche, Jean Roman and Mawussi Zounon (July 2013). *Towards resilient parallel linear Krylov solvers: recover-restart strategies.* Research Report RR-8324. INRIA. URL: `https://hal.inria.fr/hal-00843992`.

Agullo, Emmanuel, Luc Giraud, Abdou Guermouche, Jean Roman and Mawussi Zounon (May 2016). 'Numerical recovery strategies for parallel resilient Krylov linear solvers'. In: *Numerical Linear Algebra with Applications* 23.5, pp. 888–905. DOI: `10.1002/nla.2059`.

Barrett, Richard, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Chris Romine and Henk Van der Vorst (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* 1st edition. SIAM. DOI: `10.1137/1.9781611971538`.

Blackford, L., James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Gwendolyn Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington and R. Whaley (June 2002). 'An updated set of basic linear algebra subprograms (BLAS)'. In: *ACM Transactions on Mathematical Software* 28.2, pp. 135–151. DOI: `10.1145/567806.567807`.

Bland, Wesley, Aurelien Bouteiller, Thomas Herault, George Bosilca and Jack Dongarra (Aug. 2013). 'Post-failure recovery of MPI communication capability: Design and rationale'. In: *International Journal of High Performance Computing Applications* 27.3, pp. 244–254. DOI: `10.1177/1094342013488238`.

Bosilca, George, Aurelien Bouteiller, Thomas Herault, Yves Robert and Jack Dongarra (May 2014). 'Assessing the Impact of ABFT and Checkpoint Composite Strategies'. In: *2014 IEEE International Parallel Distributed Processing Symposium Workshops.* Phoenix, Arizona, USA: IEEE, pp. 679–688. DOI: `10.1109/IPDPSW.2014.79`.

Bosilca, George, Aurelien Bouteiller, Thomas Herault, Yves Robert and Jack Dongarra (Mar. 2015). 'Composing resilience techniques: ABFT, periodic and incremental checkpointing'. In: *International Journal of Networking and Computing* 5.1, pp. 2–25. DOI: `10.15803/ijnc.5.1_2`.

Cappello, Franck, Geist Al, William Gropp, Sanjay Kale, Bill Kramer and Marc Snir (Apr. 2014). 'Toward Exascale Resilience: 2014 Update'. In: *Supercomputing Frontiers and Innovations* 1.1, pp. 5–28. DOI: `10.14529/jsfi140101`.

Carson, Erin (Aug. 2015). 'Communication-Avoiding Krylov Subspace Methods in Theory and Practice'. PhD thesis. EECS Department, University of California, Berkeley. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-179.html`.

Carson, Erin (2018). 'The Adaptive s-Step Conjugate Gradient Method'. In: *SIAM Journal on Matrix Analysis and Applications* 39.3, pp. 1318–1338. DOI: `10.1137/16M1107942`.

Carson, Erin (Feb. 2020). 'An Adaptive s-step Conjugate Gradient Algorithm with Dynamic Basis Updating'. In: *Applications of Mathematics* 65, pp. 123–151. DOI: `10.21136/AM.2020.0136-19`.

Carson, Erin and James Demmel (Jan. 2014). 'A Residual Replacement Strategy for Improving the Maximum Attainable Accuracy of s-Step Krylov Subspace Methods'. In: *SIAM Journal on Matrix Analysis and Applications* 35.1, pp. 22–43. DOI: `10.1137/120893057`.

Carson, Erin, Nicholas Knight and James Demmel (Oct. 2013). 'Avoiding Communication in Nonsymmetric Lanczos-Based Krylov Subspace Methods'. In: *SIAM Journal on Scientific Computing* 35.5, pp. 42–61. DOI: `10.1137/120881191`.

Carson, Erin, Nicholas Knight and James Demmel (Dec. 2014). 'An efficient deflation technique for the communication-avoiding conjugate gradient method'. In: *Electronic transactions on numerical analysis ETNA* 43, pp. 125–141. URL: `https://etna.math.kent.edu/vol.43.2014-2015/pp125-141.dir/pp125-141.pdf`.

Chen, Zizhong (Jan. 2011). 'Algorithm-based Recovery for Iterative Methods Without Checkpointing'. In: *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. HPDC '11. San Jose, California, USA: ACM, pp. 73–84. DOI: `10.1145/1996130.1996142`.

Chronopoulos, Anthony T. and C. William Gear (July 1989a). 'On the efficient implementation of preconditioned s-step conjugate gradient methods on multiprocessors with memory hierarchy'. In: *Parallel Computing* 11.1, pp. 37–53. DOI: `10.1016/0167-8191(89)90062-8`.

Chronopoulos, Anthony T. and C. William Gear (Feb. 1989b). 's-step iterative methods for symmetric linear systems'. In: *Journal of Computational and Applied Mathematics* 25.2, pp. 153–168. DOI: `10.1016/0377-0427(89)90045-9`.

Cools, Siegfried, Jeffrey Cornelis and Wim Vanroose (Nov. 2019). 'Numerically Stable Recurrence Relations for the Communication Hiding Pipelined Conjugate Gradient Method'. In: *IEEE Transactions on Parallel and Distributed Systems* 30.11, pp. 2507–2522. DOI: `10.1109/TPDS.2019.2917663`.

Cools, Siegfried, Emrullah Fatih Yetkin, Emmanuel Agullo, Luc Giraud and Wim Vanroose (Jan. 2016). *Analysis of rounding error accumulation in Conjugate Gradients to improve the maximal attainable accuracy of pipelined CG*. Research Report RR-8849. Inria Bordeaux Sud-Ouest. URL: `https://hal.inria.fr/hal-01262716`.

Cools, Siegfried, Emrullah Fatih Yetkin, Emmanuel Agullo, Luc Giraud and Wim Vanroose (Mar. 2018). 'Analyzing the Effect of Local Rounding Error Propagation on the Maximal

Attainable Accuracy of the Pipelined Conjugate Gradient Method'. In: *SIAM Journal on Matrix Analysis and Applications* 39.1, pp. 426–450. DOI: 10.1137/17M1117872.

D'Azevedo, Ed and Chris Romine (Sept. 1992). *Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors.* Tech. rep. Oak Ridge National Laboratory. DOI: 10.2172/7172467.

Davis, Timothy A. and Yifan Hu (Dec. 2011). 'The University of Florida Sparse Matrix Collection'. In: *ACM Transactions on Mathematical Software* 38.1, pp. 1–25. DOI: 10.1145/2049662.2049663.

Demmel, James, Mark F. Hoemmen, Marghoob Mohiyuddin and Katherine A. Yelick (Oct. 2007). *Avoiding Communication in Computing Krylov Subspaces.* Tech. rep. UCB/EECS-2007-123. EECS Department, University of California, Berkeley. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-123.html.

Demmel, James, Mark F. Hoemmen, Marghoob Mohiyuddin and Katherine A. Yelick (Apr. 2008). 'Avoiding communication in sparse matrix computations'. In: *2008 IEEE International Symposium on Parallel and Distributed Processing.* Miami, Florida, USA: IEEE, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536305.

Ghysels, Pieter and Wim Vanroose (July 2014). 'Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm'. In: *Parallel Computing* 40.7. 7th Workshop on Parallel Matrix Algorithms and Applications, pp. 224–238. DOI: 10.1016/j.parco.2013.06.001.

Gutknecht, Martin and Zdenvek Strakos (June 2000). 'Accuracy of Two Three-term and Three Two-term Recurrences for Krylov Space Solvers'. In: *SIAM Journal on Matrix Analysis and Applications* 22.1, pp. 213–229. DOI: 10.1137/S0895479897331862.

Herault, Thomas and Yves Robert (2015). *Fault-Tolerance Techniques for High-Performance Computing.* 1st edition. Springer International Publishing. DOI: 10.1007/978-3-319-20943-2.

Hestenes, Magnus R. and Eduard Stiefel (Dec. 1952). 'Methods of conjugate gradients for solving linear systems'. In: *Journal of research of the National Bureau of Standards* 49.6, pp. 409–435. DOI: 10.6028/JRES.049.044.

Hoemmen, Mark F. (Apr. 2010). 'Communication-avoiding Krylov subspace methods'. PhD thesis. EECS Department, University of California, Berkeley. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-37.html.

Idomura, Yasuhiro, Takuya Ina, Akie Mayumi, Susumu Yamada and Toshiyuki Imamura (Mar. 2018a). 'Application of a Preconditioned Chebyshev Basis Communication-Avoiding Conjugate Gradient Method to a Multiphase Thermal-Hydraulic CFD Code'. In: *Supercomputing Frontiers.* Singapore: Springer International Publishing, pp. 257–273. DOI: 10.1007/978-3-319-69953-0_15.

Idomura, Yasuhiro, Takuya Ina, Susumu Yamashita, Naoyuki Onodera, Susumu Yamada and Toshiyuki Imamura (Nov. 2018b). 'Communication Avoiding Multigrid Preconditioned Conjugate Gradient Method for Extreme Scale Multiphase CFD Simulations'. In: *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*. Dallas, Texas, USA: IEEE, pp. 17–24. DOI: `10.1109/ScalA.2018.00006`.

Kelley, Carl T. (1995). *Iterative Methods for Linear and Nonlinear Equations*. SIAM. DOI: `10.1137/1.9781611970944`.

Knight, Nicholas, Erin Carson and James Demmel (May 2013). *Exploiting Data Sparsity in Parallel Matrix Powers Computations*. Tech. rep. UCB/EECS-2013-47. EECS Department, University of California, Berkeley. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-47.html`.

Langou, Julien, George Bosilca and Jack Dongarra (Oct. 2007). 'Recovery Patterns for Iterative Methods in a Parallel Unstable Environment'. In: *SIAM Journal on Scientific Computing* 30.1, pp. 102–116. DOI: `10.1137/040620394`.

Levonyak, Markus, Christina Pacher and Wilfried N. Gansterer (Jan. 2020). 'Scalable Resilience Against Node Failures for Communication-Hiding Preconditioned Conjugate Gradient and Conjugate Residual Methods'. In: *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing (PP)*. Seattle, Washington, USA: SIAM, pp. 81–92. DOI: `10.1137/1.9781611976137.8`.

Losada, Nuria, Patricia González, María J. Martín, George Bosilca, Aurelien Bouteiller and Keita Teranishi (May 2020). 'Fault Tolerance of MPI Applications in Exascale Systems: The ULFM Solution'. In: *Future Generation Computer Systems* 106.3, pp. 467–481. DOI: `10.1016/j.future.2020.01.026`.

Mayumi, Akie, Yasuhiro Idomura, Takuya Ina, Susumu Yamada and Toshiyuki Imamura (Nov. 2016). 'Left-Preconditioned Communication-Avoiding Conjugate Gradient Methods for Multiphase CFD Simulations on the K Computer'. In: *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. Salt Lake City, Utah, USA: IEEE, pp. 17–24. DOI: `10.1109/ScalA.2016.007`.

Message Passing Interface Forum (June 2021a). *MPI: A Message-Passing Interface Standard*. `https://www.mpi-forum.org/docs/`.

Message Passing Interface Forum (Nov. 2021b). *User Level Failure Mitigation*. `https://fault-tolerance.org/`.

Mohiyuddin, Marghoob, Mark F. Hoemmen, James Demmel and Katherine A. Yelick (Sept. 2009). 'Minimizing Communication in Sparse Matrix Solvers'. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon, USA: ACM, pp. 1–12. DOI: `10.1145/1654059.1654096`.

Pachajoa, Carlos, Markus Levonyak and Wilfried N. Gansterer (Nov. 2018). 'Extending and Evaluating Fault-Tolerant Preconditioned Conjugate Gradient Methods'. In: *2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. Dallas, Texas, USA: IEEE, pp. 49–58. DOI: `10.1109/FTXS.2018.00009`.

Pachajoa, Carlos, Markus Levonyak, Wilfried N. Gansterer and Jesper Larsson Träff (Aug. 2019). 'How to Make the Preconditioned Conjugate Gradient Method Resilient Against Multiple Node Failures'. In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP '19. Kyoto, Japan: ACM, 67:1–67:10. DOI: `10.1145/3337821.3337849`.

Pachajoa, Carlos, Christina Pacher, Markus Levonyak and Wilfried N. Gansterer (Aug. 2020). 'Algorithm-Based Checkpoint-Recovery for the Conjugate Gradient Method'. In: *Proceedings of the 49th International Conference on Parallel Processing*. ICPP '20. Edmonton, Alberta, Canada: ACM, pp. 1–11. DOI: `10.1145/3404397.3404438`.

Plank, James S., Youngbae Kim and Jack J. Dongarra (June 1997). 'Fault-Tolerant Matrix Operations for Networks of Workstations Using Diskless Checkpointing'. In: *Journal of Parallel and Distributed Computing* 43.2, pp. 125–138. DOI: `10.1109/HPC.1997.592191`.

Rutishauser, Heinz (1959). 'Theory of gradient methods'. In: *Refined iterative methods for computation of the solution and the eigenvalues of self-adjoint boundary value problems*. Vol. 8. Mitteilungen aus dem Institut für Angewandte Mathematik. Birkhäuser, Basel, pp. 24–49. DOI: `10.1007/978-3-0348-7224-9_2`.

Saad, Yousef (2003). *Iterative Methods for Sparse Linear Systems*. 2nd edition. SIAM. DOI: `10.1137/1.9780898718003.ch4`.

Schöll, Alexander, Claus Braun, Michael A. Kochte and Hans-Joachim Wunderlich (Oct. 2015). 'Low-overhead fault-tolerance for the preconditioned conjugate gradient solver'. In: *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. Amherst, Massachusetts, USA: IEEE, pp. 60–65. DOI: `10.1109/DFT.2015.7315136`.

Shantharam, Manu, Sowmyalatha Srinivasmurthy and Padma Raghavan (June 2012). 'Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution'. In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: ACM, pp. 69–78. DOI: `10.1145/2304576.2304588`.

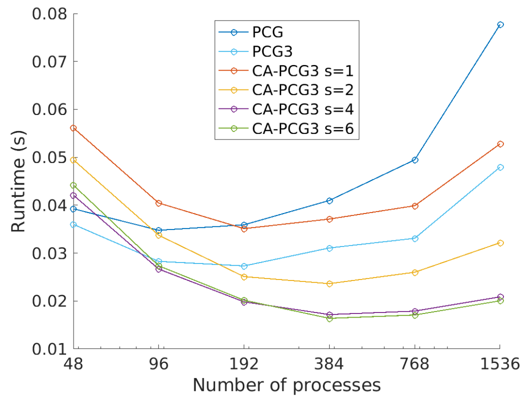
Tiwari, Manasi and Sathish Vadhiyar (Dec. 2020). 'Pipelined Preconditioned Conjugate Gradient Methods for Distributed Memory Systems'. In: *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. Pune, India: IEEE, pp. 151–160. DOI: `10.1109/HiPC50609.2020.00029`.

Toledo, Sivan (June 1995). 'Quantitative Performance Modeling of Scientific Computations and Creating Locality in Numerical Algorithms'. PhD thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science. URL: `http://hdl.handle.net/1721.1/37768`.

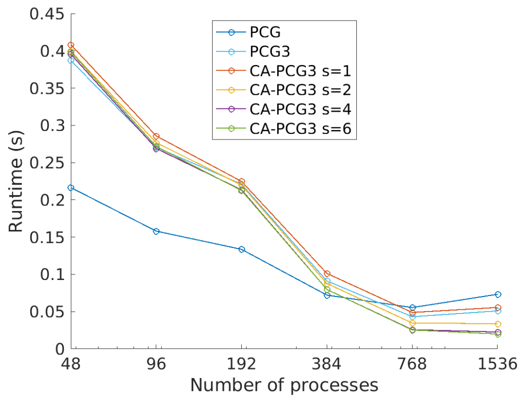
Zhang, Weizhe and Hui He (Mar. 2013). 'Fault Tolerance for Conjugate Gradient Solver Based on FT-MPI'. In: *Studies in Informatics and Control* 22.1, pp. 51–60. DOI: `10.24846/v22i2y101306`.

A. Additional Figures

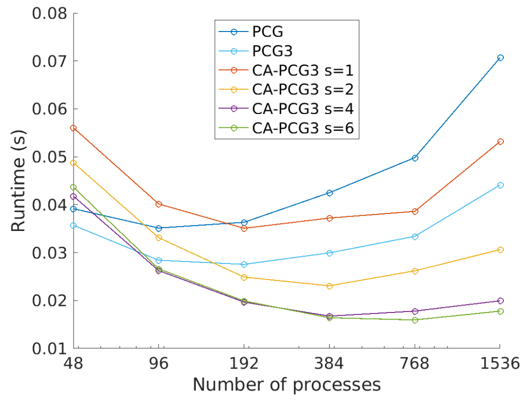
In addition to Figures 6.2a to 6.2d in Section 6.3, we show the runtimes for resilient CA-PCG3 using Methods 2 and 3 for the matrix LFAT5000 in the failure-free case and when three simultaneous process failures occur in Figures A.1a to A.1d.



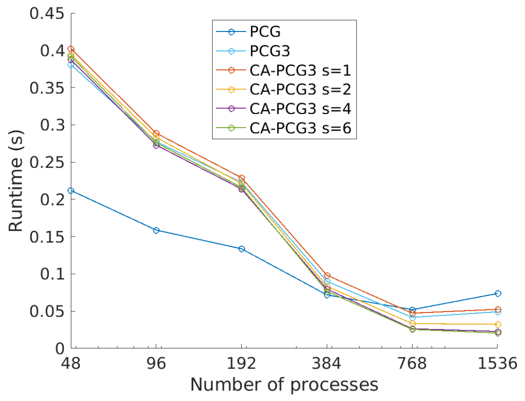
(a) Resilient PCG, PCG3 and CA-PCG3 using Method 2 in the failure-free case



(b) Resilient PCG, PCG3 and CA-PCG3 using Method 2 with three simultaneous process failures



(c) Resilient PCG, PCG3 and CA-PCG3 using Method 3 in the failure-free case



(d) Resilient PCG, PCG3 and CA-PCG3 using Method 3 with three simultaneous process failures

Figure A.1.: Runtimes of resilient CA-PCG3 using Method 2 and 3 for the matrix LFAT5000. The algorithms were executed using the same conventions and settings as in Figure 6.2. For the resilient failure-free case, data redundancy to be able to tolerate three simultaneous process failures is ensured. For the case with failures, three simultaneous process failures are simulated in iteration 300 respectively outer iteration $\lfloor 300/s \rfloor$ on rank 21, 22 and 24.