Algorithm-Based Checkpoint-Recovery
for the Conjugate Gradient Method

Carlos Pachajoa
University of Vienna
Faculty of Computer Science
Vienna, Austria
carlos.pachajoa@univie.ac.at

Christina Pacher
University of Vienna
Faculty of Computer Science
Vienna, Austria
christina.pacher@univie.ac.at

Markus Levonyak
University of Vienna
Faculty of Computer Science
Vienna, Austria
markus.levonyak@univie.ac.at

Wilfried N. Gansterer
University of Vienna
Faculty of Computer Science
Vienna, Austria
wilfried.gansterer@univie.ac.at

ABSTRACT

As computers reach exascale and beyond, the incidence of faults will increase. Solutions to this problem are an active research topic. We focus on strategies to make the preconditioned conjugate gradient (PCG) solver resilient against node failures, specifically, the *exact state reconstruction* (ESR) method, which exploits redundancies in PCG.

Reducing the frequency at which redundant information is stored lessens the runtime overhead. However, after the node failure, the solver must restart from the last iteration for which redundant information was stored, which increases recovery overhead. This formulation highlights the method's similarities to checkpoint-restart (CR). Thus, this method, which we call *ESR with periodic storage* (ESRP), can be considered a form of *algorithm-based checkpoint-restart*. The state is stored implicitly, by exploiting redundancy inherent to the algorithm, rather than explicitly as in CR. We also minimize the amount of data to be stored and retrieved compared to CR, but additional computation is required to reconstruct the solver's state. In this paper, we describe the necessary modifications to ESR to convert it into ESRP, and perform an experimental evaluation.

We compare ESRP experimentally with previously-existing ESR and application-level in-memory CR. Our results confirm that the overhead for ESR is reduced significantly, both in the failure-free case, and if node failures are introduced. In the former case, the overhead of ESRP is usually lower than that of CR. However, CR is faster if node failures happen. We claim that these differences can be alleviated by the implementation of more appropriate preconditioners.

1 INTRODUCTION

In order to cover the demand for solving contemporary computational problems in reasonable times, modern computer clusters reach unprecedented levels of parallelism. The mean time between failures (MTBF) in clusters formed of increasingly numerous nodes and components will continue to drop. In these circumstances, there is good reason to start thinking of parallel computers as unreliable machines [23], and to come up with strategies to work around this problem.

The solution of linear equations for symmetric, positive-definite (SPD) matrices is a problem of great importance in science and engineering. These matrices often arise from the discretization of elliptic differential equations, describing phenomena such as heat conduction and elastic deformation of materials. More detailed simulations require a finer grid and larger matrices. The solution of these problems often requires computer clusters and, if the scale of the machine is large enough, the solver is prone to encounter faults in the computer it runs on. To solve linear systems defined by SPD matrices, the choice of solver often is the *preconditioned conjugate gradient* (PCG) method (cf. Section 2.1). For very large matrices, running PCG on large numbers of computer nodes is warranted. Thus, it is worth to develop resilience strategies for this algorithm.

A particular mode in which faults may occur in a cluster are *node failures*: events where one or more nodes that were working on the solution of the system become inaccessible and the information contained in them is lost. To cope with them, checkpoint-restart (CR) is currently the most extensively used strategy. In CR, the state of the application is periodically stored in safe storage (this is the

checkpointing part of CR), and in the event of a node failure, the cluster can revert to a previously stored state and continue from there (the restart part of CR). Checkpointing has a runtime cost and since, in the absence of errors, this operation does not contribute to the results the application is producing, it is desirable to keep the checkpointing frequency as low as possible. Returning to the previous state, however, incurs the runtime cost of discarding the iterations performed since the checkpoint. Therefore, there is also pressure to checkpoint as often as possible. Finding the optimal checkpointing period is discussed in the literature, and the value will depend on the incidence of errors in the machine [8, 11, 28].

The application of CR is straightforward: The state can be the content of relevant variables selected by the programmer (application-level CR), or simply the contents of all the memory used by it (full-memory CR). However, there are drawbacks that the method described in this paper improves upon:

- It has been pointed out that CR will not scale up well on future exascale machines, particularly in the full-memory case [4, 5].
- CR strategies are generally algorithm-agnostic. The technique consists of storing data and reverting to it, ignoring any resilience arising from the algorithm itself, thus potentially being suboptimal for the task at hand.

In contrast to these algorithm-agnostic approaches, there is a category of strategies to deal with computer unreliability called *algorithm-based fault-tolerance* (ABFT), a concept first introduced in [13] where it is applied to dense matrix multiplication, that exploits the properties of the algorithms to endow them with resilience.

In this work, we focus on the PCG method used to solve the linear system $Ax = b$, where A is an sparse SPD matrix. PCG runs on a cluster that is vulnerable to node failures. We expand on a previously introduced strategy: *exact state reconstruction* (ESR) [7, 20], which provides ABFT resilience against node failures for PCG. Our contributions in this work reduce its runtime overhead considerably, particularly in the case of multiple simultaneous node failures.

1.1 Terminology and notation

In this section, we introduce some terms that are used in this paper.

Lost nodes are nodes that stop working in the event of a node failure: They become inaccessible and the information contained in them is lost. A *spare node* is in standby until a node failure occurs, at which point it replaces one of the lost nodes by reconstructing its information and continue iterating in its place. A spare node that, upon occurrence of a node failure, takes the place of a lost node is called a *replacement node*. A *surviving node* is still working after a node failure, its information remains accessible, and it also continues working after the reconstruction. In this paper, events in which a single node or multiple nodes fail simultaneously are called single-node failure and multiple-nodes failure, respectively.

We use the notation of [7] to refer to indices of vectors and matrices. The set of all indices is referred to as I. For a problem of size M, the cardinality of I is M. A subindex will restrict this index set to the indicated node or set of nodes: For example, the entries corresponding to node s will be denoted as I_s. If f is the set of nodes that fail simultaneously, the set of indices corresponding to the lost elements is denoted as I_f, and we can refer to the set of indices for elements in surviving nodes as $I \setminus I_f$. We use index sets as subscripts to refer to entries of vectors and matrices, for example, if we have a distributed vector x, we refer to the surviving vector elements as $x_{I \setminus I_f}$, and for a matrix A, the rows corresponding to the lost nodes can be written as $A_{I_f, I}$. We say that a node *owns* an index, and thus the corresponding vector entry and matrix row, if the index is in the set assigned to it. In the context of the PCG method, a superscript (as in $x^{(j)}$) indicates the iteration number of a vector or scalar.

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

The *state* of an iterative solver refers to all of the solver's dynamic data, i.e. the vectors and scalars whose values change in every iteration, distributed among all nodes. It does not include the static data (system matrix, preconditioner and right-hand-side vector). The *trajectory* followed by the solver is the sequences of states that it goes through until convergence. A given state fully defines the trajectory subsequently followed by the solver: All future states are completely determined by the current state.

1.2 Problem definition

In this paper, we work on the solution of sparse, symmetric, positive definite linear systems in a distributed-memory setting using the PCG method. The solver data is distributed, and the problem is solved in N nodes of a computer cluster. Disjoint subsets I_s of consecutive indices are distributed among the nodes, such that their union forms the set of all indices, I. Node s is assigned the rows of the matrix, and entries of the vectors, corresponding to the indices in its subset I_s. This *block row distribution* is common, and is used in prominent libraries such as PETSc [2]. The scalars used by the solver are replicated in all nodes.

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

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

1.3 Related work

Here we present literature concerned with the recovery of linear solvers from node failures.

Concerning the checkpoint-restart approach, the currently most common way to deal with this problem, we highlight work by Tao [25]. In this thesis, the author proposes techniques for lossy compression of checkpoint data, using prediction formulas based on spatial proximity.

In work on the solution of linear systems originating from partial differential equations (PDE), Ltaief et al. present a strategy for resilience for parabolic PDEs against node failures [17]. Here, forward and backward-stepping strategies are described to reconstruct data in the physical domain associated to the lost node, avoiding a more

expensive checkpoint-restart approach. The method reconstructs the iterand exactly, like the methods described in this paper, but its application is a time-stepping solver with finite-differences discretization, whereas ours is the conjugate gradient algorithm. In [14], Huber et al. present an algorithm for the reconstruction of the subdomain after a node failure for a multigrid solver for the Laplace equation. The affected values are reconstructed approximately by solving a linear system local to the lost node.

Langou et al. [15] work with a wider variety of iterative linear solvers. After a node failure, approximations to the lost entries of the iterand are found using the system matrix, the right-hand-side vector and the surviving data of the iterand itself by solving a small linear system. They can bound the new residual norm as less than the residual norm before the node failure times a constant factor. This method incurs no overhead in the absence of node failures.

Agullo et al. [1] improve upon this strategy. As in [15], they approximate the lost entries of the iterand from its surviving information, but use least-squares minimization instead of solving a linear system. As a result, the residual norm of the new vector will be less than or equal to the residual norm before the node failure.

Chen [7] introduces a way to perform exact state reconstruction (ESR) for multiple iterative methods, including PCG. They present a strategy to exploit the sparse matrix-vector product (SpMV) product to store redundant information for the input vector, so that the full state of the vector can be reconstructed. In [20], Pachajoa et al. extend the algorithm in [7] to combine ESR with differently formulated preconditioners (the preconditioner itself, its inverse or a split preconditioner), and also compare ESR to the linear interpolation algorithm from [15]. In [21], Pachajoa et al. extend the ESR approach by describing how to operate in the event of multiple, simultaneous node failures.

In [16], Levonyak et al. extend the concept of ESR to the pipelined PCG algorithm, while maintaining its communication-hiding properties.

The work mentioned so far supposes the availability of spare nodes. In [12], Hori et al. propose strategies for the allocation of these spare nodes, and the replacement of lost nodes, when runtime performance is of consideration.

Pachajoa et al. [22] introduce an ESR method which does not require spare nodes as replacements for failed nodes, but can reconstruct the lost information and continue on the surviving nodes.

1.4 Contributions of this paper

As the main contribution of this paper, we frame ESR for PCG as an instance of what we call *algorithm-based checkpoint-restart*, and describe how it can be restructured to enable decreased state-storage frequencies. We experimentally show that this approach reduces the runtime overhead in the absence of node failures. This is particularly beneficial in scenarios with multiple node failures, where the additional communication needs increase the overhead most drastically and, consequently, for which the runtime overhead reduction is greatest.

The rest of this paper is structured in the following manner: In Section 2, we describe the exact state reconstruction approach applied to PCG in more detail. In Section 3, we reframe ESR, as presented in [21], as a CR-like method, for which the state-storage

interval can be optimized, and which offers reduced runtime overheads. In Section 4, we describe the framework we use to obtain our experimental results. In Section 5, we present our experimental results, highlighting favorable scenarios for new methods. Finally, Section 6 concludes the paper and presents our perspectives on future work.

2 ALGORITHMIC BACKGROUND

In this section, we introduce the PCG method, describe the way we exploit the inherent data redundancies in the sparse matrix vector-product, and explain the exact state reconstruction method.

2.1 Preconditioned conjugate gradient

PCG is a linear solver for the system $Ax = b$, where A is an SPD matrix. The method is applied in conjunction with a preconditioner, which reduces the number of iterations until convergence at the price of the application of the preconditioner in every iteration.

The variables used in PCG are the following: x is the iterand vector, containing the current approximation to the solution. P is the preconditioner, here representing its action as a linear operator. The residual vector is represented with r, and the preconditioned residual vector with z. The search direction vector, p, determines the direction in which the iterand is modified in every iteration. β is a scalar used for the conjugation of the search directions, and α is a scalar determining the length of the step to be taken along the search direction towards the solution. PCG is presented in Alg. 1.

In exact arithmetic, supposing a naive selection of the initial guess, solving an SPD linear system of size M will take the conjugate gradient method M iterations to reach the solution. If the solver is restarted from the iterand at some point before convergence, reinitializing the search directions, reaching the solution might require performing M additional iterations from that point, thus wasting the work already performed. In [19], it is shown that this effect is also observed in floating-point arithmetic. This observation is the motivation for the *exact state reconstruction* (ESR) method (cf. Section 2.3).

Algorithm 1: Preconditioned conjugate gradient (PCG) method [24, Alg. 9.1]

---

1   $r^{(0)} := b - Ax^{(0)}, z^{(0)} := Pr^{(0)}, p^{(0)} := z^{(0)}$;

2   **for** $j = 0, 1, \ldots,$ *until convergence* **do**

3      $\alpha^{(j)} := r^{(j)\mathsf{T}}z^{(j)}/p^{(j)\mathsf{T}}Ap^{(j)}$;

4      $x^{(j+1)} := x^{(j)} + \alpha^{(j)}p^{(j)}$;

5      $r^{(j+1)} := r^{(j)} - \alpha^{(j)}Ap^{(j)}$;

6      $z^{(j+1)} := Pr^{(j+1)}$;

7      $\beta^{(j)} := r^{(j+1)\mathsf{T}}z^{(j+1)}/r^{(j)\mathsf{T}}z^{(j)}$;

8      $p^{(j+1)} := z^{(j+1)} + \beta^{(j)}p^{(j)}$;

9   **end**

2.2 Augmented sparse matrix-vector product

In order to reconstruct the entirety of the state of the PCG solver as it was before a node failure, enough redundant information has to be available. That is, there has to be redundancy that we can exploit.

The SpMV already provides some redundancy: In order to compute the product of matrix A and vector p, entries of p must be transmitted from their owner node to other nodes in the cluster, thus already creating copies of some entries in other locations. However, in order for the SpMV to provide full redundancy for the input vector, the matrix must fulfill the following condition [7]: For every node s, every column of the submatrix $A_{I \setminus I_s, I_s}$ contains at least one non-zero entry. For a matrix with this property, every entry of the vector is communicated from its owner to at least one other node. Most matrices do not fulfill this condition. Furthermore, this would only guarantee sufficient redundancy to recover from the failure of a single node.

To achieve the required redundancy, we use the extensions to the SpMV introduced in [7, 21] and name the concept *augmented sparse matrix vector product* (ASpMV). Entries that would not have been sent to any node with the ordinary SpMV are transferred to a neighbor anyway. With our chosen strategy, node s sends a given entry to node $(s + 1) \bmod N$, if this entry is not already being sent to some other node as part of the regular SpMV.

The exact communication overhead depends on the sparsity pattern of the matrix. In general, denser matrices will have lower overheads for ASpMV, since more information has to be sent anyway to compute the product. With the nodes sending information to their neighbors, it is convenient if the matrix is banded, with most of its entries close to the diagonal. That way, the amount of information that ASpMV has to send additionally to the neighbors is minimized.

2.2.1 Redundancy against multiple-nodes failures.
ASpMV can also guarantee the presence of several redundant copies of each input vector element. This is necessary in order to provide resilience against multiple-nodes failures. In order to describe this extension, we make use of the notation of [21, §3 and §4]: Let ϕ denote the target number of times each entry of the input vector must be replicated in the cluster (i.e. the number of simultaneous node failures that should be supported). Before, we defined I_s as the set of all indices owned by node s. Let $I_{s,l}$, with $l \in \{1..N\}$, be the subset of I_s with indices of the input vector p corresponding to entries that must be sent to node l for the computation of the product Ap. Node s does not send data to itself during this operation, so we define $I_{s,s} \coloneqq \emptyset$. Furthermore, let $d_{s,k}$ denote the designated destination nodes for resilient copies of the vector elements of node s, with $k \in \{1..\phi\}$. In this work, we select $d_{s,k}$ to be the ϕ nearest neighbors of node s. This can be achieved with the following strategy:

$$d_{s,k} \coloneqq \begin{cases} \left(s + \left\lceil \frac{k}{2} \right\rceil \right) \bmod N, & \text{if } k \text{ odd} \\ \left(s - \frac{k}{2}\right) \bmod N, & \text{if } k \text{ even} \end{cases} \quad (1)$$

For index i, which belongs to node s, we define its multiplicity, $m(i)$, as the number of subsets $I_{s,l}$, with $l \in \{1..N\}$, where i is present. That is, $m(i)$ is the number of nodes that the ith vector entry must be sent to in order to compute Ap. Additionally, we define $g(i)$ as the number of the subsets $I_{s,d_{s,k}}$, $k \in \{1..\phi\}$ in which i appears, that is, how many of the nodes $d_{s,k}$ already need the ith entry to compute Ap.

We can now describe the set $R^c_{s,k}$ of indices of entries, owned by node s, to be sent to $d_{s,k}$, in addition to the ones required to compute the product Ap:

$$R^c_{s,k} \coloneqq \left\{ i \in I_s \mid i \notin I_{s,d_{s,k}} \text{ and } m(i) - g(i) < \phi - k \right\}.$$

That is, the ith entry, if owned by node s, will be sent to $d_{s,k}$ if (1) it is not already being sent there and if (2) as we traverse the designated destination nodes by increasing k, the target number of copies for this entry has not been met yet. The approach for a single-node failure described earlier in this section is the same as the approach for a multiple-nodes failure, with ϕ set to 1.

After the ASpMV is complete, each entry of the vector will have been communicated by its owner to at least ϕ nodes, thus creating $\phi + 1$ copies (one of them in the owner). If up to ϕ nodes fail simultaneously, each entry will have survived in at least one node and can afterwards be transferred to a replacement node.

With this method, we send entries that are not necessary for the computation of the matrix-vector product. The communication of this additional information will cause iterations to take longer, and thus lead to an increased runtime until convergence. The exact overhead depends on factors such as the sparsity pattern of the matrix and the network topology of the cluster. Optimization of our strategies taking these factors into consideration is beyond the scope of this paper. Research in this direction is ongoing work.

2.2.2 Redundant copies.
After the ASpMV is executed, the redundant information of the search direction $p^{(j)}$ is not explicitly available. This means, even though the information of the copies is present and spread in the cluster, after a node failure, the data must be gathered in a replacement node before we can work with $p^{(j)}$ again. We introduce the concept of a *redundant copy*, designated with a prime symbol ($'$), as in $p'^{(j)}$, to abstractly represent the redundant vector data in the cluster, in whatever storage scheme the framework utilizes. We do not specify the number of copies per vector entry that a redundant copy represents, since we do not need this information for further descriptions. The concept of redundant copies will be used to explain our algorithms in Section 3.

2.3 Exact state reconstruction for PCG

The PCG algorithm performs a matrix-vector product in every iteration (Line 3 of Alg. 1): The system matrix A is multiplied with the search direction vector p, thus potentially providing redundancy for the latter. In [21], the authors explain how to reconstruct the state of the solver as it was prior to the node failure, save for small perturbations resulting from floating-point arithmetic. With the two latest search directions, and after retrieving the scalar β from one of the surviving nodes, it is possible to move backwards from Line 8 of Alg. 1 and reconstruct every vector involved in the computation. The reconstruction procedure, run on replacement nodes, is presented in Alg. 2. Note that the reconstruction procedure assumes that the static solver data (system matrix, preconditioner and right-hand-side vector) can be retrieved from safe storage.

Algorithm 2: ESR reconstruction phase for the PCG method on the replacement nodes [20, Alg. 2]

1　Retrieve the static data $A_{I_f,I}$, $P_{I_f,I}$, and b_{I_f};

2　Gather $r^{(j)}_{I\backslash I_f}$ and $x^{(j)}_{I\backslash I_f}$;

3　Retrieve the redundant copies of $\beta^{(j-1)}$, $p^{(j-1)}_{I_f}$, and $p^{(j)}_{I_f}$;

4　Compute $z^{(j)}_{I_f} := p^{(j)}_{I_f} - \beta^{(j-1)}p^{(j-1)}_{I_f}$;

5　Compute $\boldsymbol{v} := z^{(j)}_{I_f} - P_{I_f,I\backslash I_f}r^{(j)}_{I\backslash I_f}$;

6　Solve $P_{I_f,I_f}r^{(j)}_{I_f} = \boldsymbol{v}$ for $r^{(j)}_{I_f}$;

7　Compute $\boldsymbol{w} := b_{I_f} - r^{(j)}_{I_f} - A_{I_f,I\backslash I_f}x^{(j)}_{I\backslash I_f}$;

8　Solve $A_{I_f,I_f}x^{(j)}_{I_f} = \boldsymbol{w}$ for $x^{(j)}_{I_f}$;

With the state of the solver as it was before the node failure, it is possible to reach convergence following the same trajectory as an undisturbed solver. As illustrated in [21], this method produces very low overheads, particularly if it protects only against single-node failures.

3　ESR WITH PERIODIC STORAGE

In this section, we extend ESR introduced in Section 2.3 to perform iterations with ASpMV with a reduced frequency, that is, not in every iteration, but two consecutive times every T iterations. We call T the *checkpointing interval* to keep with CR terminology, and we refer to the set of two iterations in which redundant information is stored as the *storage stage*. In the event of a node failure, the solver will return to the last time the search directions for two successive iterations were stored redundantly via ASpMV. It is then possible to reconstruct the state for the last of those iterations. We call the new approach *Exact state reconstruction with periodic storage* (ESRP), and contrast it with ESR which stores data in every iteration.

To describe ESRP, we introduce the concept of a *queue*, where the solver stores redundant copies (cf. Section 2.2.2). In the ESR algorithm, this queue has space for two positions: Every iteration, ASpMV will push a new redundant copy into the queue, and the oldest copy will be released. This queue thus contains the redundant copies of search directions for two successive iterations.

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

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

In addition to the redundant copies created during the ASpMV, the solver needs to duplicate some local data at each node during the storage stage. As can be seen in Line 4 of Alg. 2, to reconstruct the vector $z^{(j)}$, and subsequently the state of the solver for iteration j, the value of $\beta^{(j-1)}$ is needed. However, depending on when the node failure occurs, the scalar β may have changed since the last storage stage. It is therefore necessary to create a duplicate of the value of $\beta^{(j-1)}$. Similarly, the local entries of the residual r, the preconditioned residual z, the iterand x and the search direction p at iteration j must also be duplicated in all nodes, so that they can be used for the reconstruction process and so that the surviving nodes can reset their own parts of the solver state to match the state that is reconstructed at the replacement nodes. Entries of the vector $z^{(j)}$ corresponding to surviving nodes are not used during the reconstruction, and could also be recomputed from $r^{(j)}$ once the latter has been reconstructed. However, our solver stores a local copy instead of performing this operation. We mark these duplicate values with an asterisk: β_* is a scalar, and r_*, p_*, z_* and x_* are distributed vectors. There is no need to store the scalar α. It is not used during the reconstruction and it will be computed in Line 13 of Alg. 3 when the solver continues iterating. Since these copies are created locally by each node, they do not introduce any additional communication between nodes, and the runtime overhead they cause is therefore negligible. Note that since r_*, p_*, z_* and x_* are created by each node copying its own data, if a node fails, the copies contained in it are also lost. Therefore, these copies, by themselves, obviously cannot be used to reconstruct the state.

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

When the next storage stage is reached, after an additional T iterations, we use ASpMV again, and Q becomes $[p'^{(T)}, p'^{(T+1)}, p'^{(2T)}]$, and again, a copy of $\beta^{(2T)}$ is created. The newest entry of Q cannot be used in conjunction with the previous ones to reconstruct the state of the solver. In the event of a node failure at this point, the state would be recovered for iteration $T+1$. At this time, we still need the value of $\beta^{(T)}$ to reconstruct the state, so we may not overwrite β_* yet, and will overwrite it in the next iteration instead. After an additional successful iteration, the queue contains $[p'^{(T+1)}, p'^{(2T)}, p'^{(2T+1)}]$, and copies for $r^{(2T+1)}$, $z^{(2T+1)}$, $x^{(2T+1)}$ and $p^{(2T+1)}$ are created. From that point on, it is possible to reconstruct for iteration $2T+1$, which concludes the second storage stage. The process is presented in Alg. 3 and graphically represented in Fig. 1. For the description of the algorithms in Section 3, we refer to the ordinary SpMV as $\boldsymbol{\varrho} := \mathrm{SpMV}(A, p)$, a function that takes

matrix A and vector p as inputs and returns their product, ϱ. The augmented variant is represented as $\varrho := \mathsf{ASpMV}(A, p, \phi, Q)$, where the function additionally takes the number of desired redundant copies ϕ, and the queue Q where it will push a new *redundant copy* for p.

Algorithm 3: Preconditioned conjugate gradient (PCG) method with periodic redundant storage (for ESRP)

---

1   $r^{(0)} := b - Ax^{(0)}, z^{(0)} := Pr^{(0)}, p^{(0)} := z^{(0)}, j := 0,$

2   $Q := [\_, \_, \_];$

3   **repeat**

4   **if** $j \bmod T = 0$ *and* $j > 2$ **then**

5         $\varrho^{(j)} := \mathsf{ASpMV}(A, p^{(j)}, \phi, Q);$

6         $\beta ** = \beta^{(j)};$

7   **else if** $(j - 1) \bmod T = 0$ *and* $j > 2$ **then**

8         $\varrho^{(j)} := \mathsf{ASpMV}(A, p^{(j)}, \phi, Q);$

9         $x* = x^{(j)}, \; r* = r^{(j)}, \; z* = z^{(j)}, \; p* = p^{(j)};$

10        $\beta* = \beta **;$

11   **else**

12         $\varrho^{(j)} := \mathsf{SpMV}(A, p^{(j)});$

13      $\alpha^{(j)} := r^{(j)\mathsf{T}}z^{(j)}/p^{(j)\mathsf{T}}\varrho^{(j)};$

14      $x^{(j+1)} := x^{(j)} + \alpha^{(j)}p^{(j)};$

15      $r^{(j+1)} := r^{(j)} - \alpha^{(j)}\varrho^{(j)};$

16      $z^{(j+1)} := Pr^{(j+1)};$

17      $\beta^{(j)} := r^{(j+1)\mathsf{T}}z^{(j+1)}/r^{(j)\mathsf{T}}z^{(j)};$

18      $p^{(j+1)} := z^{(j+1)} + \beta^{(j)}p^{(j)};$

19      $j := j + 1;$

20 **until** $\|r\|_2/\|b\|_2 < rtol;$

From Fig. 1 we can see which checkpointing intervals make sense for ESRP. For $T > 2$, we proceed as explained above. For $T = 2$, it no longer makes sense to use this approach, since redundant copies are created in every iteration. In this case, it is better to use ESR (Section 2.3). For $T = 1$, we can no longer talk about creating two successive copies. Again, this corresponds to regular ESR.

3.1 ESR/ESRP and checkpointing

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

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

In-memory checkpoint-restart. In our experiments in Section 5 we compare the runtime of our ESRP approach to an in-memory buddy checkpoint-restart strategy (IMCR), which will now be described in detail.

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

There are further similarities between this checkpointing strategy and ESR/ESRP: For example, the data that can be retrieved from a checkpointing buddy is the same data that is reconstructed during the recovery phase of ESR, and the strategy for choosing the buddies is the same as the strategy for determining the destinations of redundant elements in the augmented sparse matrix-vector product, as defined by Eq. 1 in Section 2.2.1. An important difference, however, is that ESR mainly adds on to existing communication, while the checkpointing strategy introduces a completely new round of communication in each storage iteration.

4 IMPLEMENTATION

We implement our algorithms using our own framework, written in C. In this way we achieve the highest flexibility for our purpose. Elementary linear algebra functionality is provided by GSL [10], but parallelization of linear algebra operations, in particular of the matrix-vector product, is in-house code. Communication between nodes is realized with MPI. The framework is modularly structured, such that different strategies to achieve data redundancy, and to perform reconstruction and recovery, can be used. In particular, we can simulate ESRP, as well as in-memory CR.

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

In a real-world scenario, the replacement nodes would also have to reload the rows of the system matrix and the preconditioner, and the entries of the right-hand-side vector that they own; however, we have decided not to include this step in the measurement of the runtime overheads during our experiments. There are two main reasons for this decision. Firstly, this overhead depends too strongly on the individual use case for us to be able to make any generalized statements about it: Not only is it influenced by the matrix size and file system properties, but matrices might be stored in different file formats (e.g. plain text or binary), or the solver might be
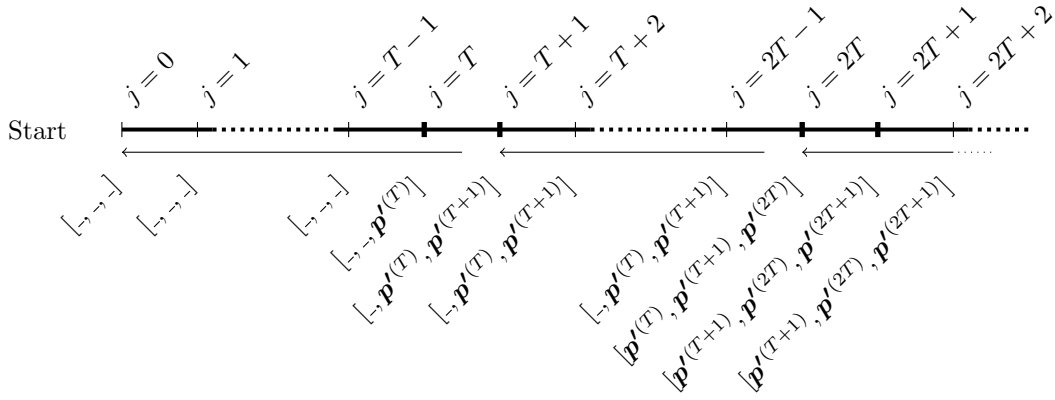
Figure 1: State of the redundancy queue for the search directions during the solution process. The lists below the line represent the state of the search direction queue, and which search directions are stored redundantly somewhere in the cluster. The thin arrows running leftwards show how far the solver has to revert in the event of a node failure.

working with a matrixless representation altogether. This changes the loading time considerably. Secondly, the reloading step is the same for both the ESR and CR versions we investigate. Therefore, we would not gain any valuable information about differences in the behaviour of these strategies from examining the time required for the reloading of static data.

Beyond node-failure simulation

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

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

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

A third task would be to setup the cluster to continue working. For ESR as as well as the CR strategy described in this paper, this involves providing a replacement node to take the place of the lost node and setting up a new communicator to continue iterating. In the case of ESR, it is also possible to proceed without a replacement node (cf. [22]), but this is beyond the scope of this paper. More generally speaking, depending on how exactly CR is implemented, it could make use of spare nodes, enabling the application to keep

using most of the nodes already allocated to it, but making it necessary to identify the identity of the lost node just as in the case of ESR; or the whole application could be restarted on newly-allocated nodes, although this is likely to be more costly that identifying the lost nodes, particularly at greater scales [6, 12, 26].

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

Although presently not in the MPI standard, there is ongoing work on tools to deal with node failures. The *User-Level Fault Mitigation* (ULFM) library [3, 18] offers functions for detection of node failures and identification of the affected nodes. In conjunction with standard MPI, it is possible to create a new communicator on which the solver can continue working.

5 EXPERIMENTS

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

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

If no protective measures are taken, node failures can cause the loss of all the computation invested in the solution of a linear system. However, they are a relatively rare occurrence. With the incidence estimations of [11] (mean time between failures of 9 hours for $100\,000$ nodes, and 53 minutes for $1\,000\,000$ nodes), a linear solver might be affected by at most a few of such events during its runtime. Thus, we consider that examining the behavior of the solver when a single node-failure event strikes at some point during its operation is a useful exercise.

We use a block Jacobi preconditioner, with non-overlapping blocks and all rows of a block belonging to a single node. The blocks are uniformly sized and we use as few of them as possible, with a maximum block size of 10. This preconditioner is used both for the linear system of the problem we are solving, and for the inner

Table 1: Test matrices from [9]

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

systems of the reconstruction (Lines 6 and 8 of Alg 2). The solver has converged once the relative residual $\|r\|_2/\|b\|_2$ is below 10^{-8}. The relative residual for the inner system for the reconstruction must reach 10^{-14} for convergence.

Our test problems are SPD matrices from the SuiteSparse Matrix Collection [9] (see Table 1). They were selected based on their size, to allow for comparisons with related work in [21]. With these matrices, we set up the test constellation as follows:

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

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

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

Experimental results

We measure the runtime t of the solver to reach convergence to evaluate our algorithms. We define the reference time, t_0, as the median time for runs of a reference, non-resilient PCG solver. The metric that we present in our results is the relative overhead over this reference time: realtive overhead $= (t - t_0)/t_0$. Our results are summarized in Tables 2 and 3, and shown in Fig. 2 and 3. Note that in all of our experiments the measured total runtime is less than a minute and, thus, well below an estimated mean time between failures of approximately an hour up to several hours in large-scale applications [11]. Since we have to recover from a node failure in that short runtime, the overhead due to this recovery is likely more severe than in a real-world application with less frequent recoveries. Therefore, we test a scenario that is *less favorable* for our approach. Reasonably low overheads in our experiments can hence be considered a proof of concept.

For the used test matrices, overheads of ESRP are usually much smaller than the ones of ESR (Section 2.3), and this advantage is more pronounced for larger numbers of redundant copies. In the case of ESRP, the columns for reconstruction overhead in Tables 2

and 3 show the cost of gathering the information and of recomputing the lost data. In the case of IMCR, these columns show the cost of communicating checkpointed data to the replacement nodes. For both test matrices, our experiments show a reconstruction overhead of basically zero for IMCR. This suggests that in our experimental setup the communication cost is considerably smaller than the computation cost. Keeping in mind the common understanding that communication tends to become more expensive than computation, especially in the large scale, this observation indicates that the experimental setup currently available to us possibly leads to underestimating the communication cost and does not allow for a solid comparison with IMCR. Consequently, we decided to depict experimental measurements for IMCR less prominently in Figs. 2 and 3 and leave more representative experiments at larger scales for future work.

The cluster that we use introduces a certain amount of variation to our measurements. We repeat each experiment to reduce the standard deviation of the tests, and we present their median. However, there are cases in which this standard deviation for the reference runtimes is not reduced below the overhead. Table 2 shows an instance of this: The median reference time for ESRP for three redundant copies is larger for a period of 50 that for a period of 20. In general, we expect larger checkpoint intervals to produce smaller overheads in the failure-free case, but these results can be affected by the variation in runtime from external factors. In this case, the overhead is so close to zero that it is overshadowed by the noise from the machine.

Table 2 shows that, for the matrix `Emilia_923`, the failure-free overhead for ESRP is lower than for IMCR, down to about half of the corresponding value for IMCR in some settings. For ESRP, reducing the frequency at which redundant copies are stored visibly reduces the overhead, especially in cases with multiple-nodes failures (Tables 2 and 3). As for matrix `audikw_1`, Table 3 shows that overheads for the failure-free cases for ESRP and IMCR are close, with some advantage for ESRP in cases with multiple-node failures.

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

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

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

| Strategy	T	Failure-free overhead [%]			Location	Overhead with node failures [%]			Reconstruction overhead [%]		
		$\phi = 1$	$\phi = 3$	$\phi = 8$		$\phi = \psi = 1$	$\phi = \psi = 3$	$\phi = \psi = 8$	$\phi = \psi = 1$	$\phi = \psi = 3$	$\phi = \psi = 8$
ESRP	1	0.5	1.3	9.1	Start	2.8	3.7	11.5	2.4	2.1	3.6
					Center	2.4	3.4	10.7	1.9	2.2	2.8
	20	0.1	0.4	1.7	Start	2.0	2.9	4.6	2.4	2.1	3.6
					Center	2.1	3.0	4.4	1.1	2.2	2.8
	50	0.4	0.7	1.3	Start	2.7	5.0	5.0	1.6	2.9	3.6
					Center	2.5	3.7	3.8	1.1	2.2	2.8
	100	0.3	0.2	1.1	Start	3.5	4.0	5.5	1.6	2.9	3.6
					Center	3.2	4.2	4.1	1.9	2.2	2.8
IMCR	20	1.1	2.2	5.3	Start	0.9	2.8	5.7	0.0	0.0	0.0
					Center	1.5	2.3	5.6	0.0	0.0	0.0
	50	0.5	1.4	2.3	Start	1.2	2.1	3.2	0.0	0.0	0.0
					Center	1.0	1.7	3.3	0.0	0.0	0.0
	100	0.4	1.2	1.3	Start	2.3	2.1	2.2	0.0	0.0	0.0
					Center	1.7	1.9	3.5	0.0	0.0	0.0

Table 3: Results for matrix `audikw_1`. Reference time t_0 = 23.22 s. The reference case takes C = 5543 iterations to reach convergence. Symbols and terms are explained in the caption of Table 2

| Strategy	T	Failure-free overhead [%]			Location	Overhead with node failures [%]			Reconstruction overhead [%]		
		$\phi = 1$	$\phi = 3$	$\phi = 8$		$\phi = \psi = 1$	$\phi = \psi = 3$	$\phi = \psi = 8$	$\phi = \psi = 1$	$\phi = \psi = 3$	$\phi = \psi = 8$
ESRP	1	4.4	4.6	7.4	Start	5.5	8.0	13.2	1.3	2.6	5.7
					Center	5.8	6.2	10.4	1.3	1.5	2.2
	20	0.9	0.9	1.4	Start	2.9	3.6	7.5	1.8	2.5	5.7
					Center	2.5	2.6	3.7	1.3	1.5	2.3
	50	0.7	0.4	0.4	Start	3.4	4.1	7.1	1.8	2.7	5.7
					Center	2.4	2.9	3.4	1.3	1.5	2.2
	100	0.1	0.2	0.4	Start	3.3	4.8	8.3	1.3	2.5	5.7
					Center	3.6	3.4	4.3	1.3	1.5	2.3
IMCR	20	0.3	0.8	2.1	Start	0.6	1.1	2.2	0.0	0.0	0.0
					Center	0.5	1.1	2.3	0.0	0.0	0.0
	50	0.1	0.4	0.9	Start	1.0	1.0	1.8	0.0	0.0	0.0
					Center	1.0	2.0	1.9	0.0	0.0	0.0
	100	0.0	0.2	0.7	Start	1.8	1.9	2.3	0.0	0.0	0.0
					Center	1.7	2.2	2.5	0.0	0.0	0.0

a method with a lower recovery cost is preferable if the probability of encountering a node failure is higher.

Accuracy of the experiments. In general, when working with PCG without residual replacement, there is some drift between the vector r and the vector $b - Ax$ [27]. In ESRP, solving the inner system with an iterative solver, and performing the reconstruction in floating-point arithmetic, cause the reconstructed vector r not to be exactly equal to its state before the node failure. To evaluate the accuracy
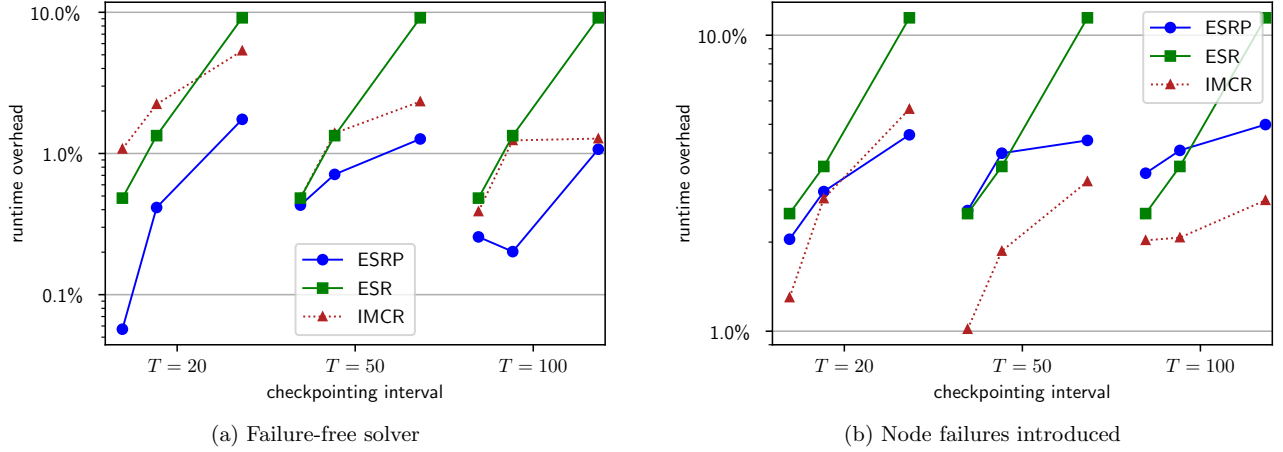
Figure 2: Median runtime overhead for the experiments with the matrix `Emilia_923`. In each plot, experiments for a check-point interval T are clustered together. In each cluster, there are three lines, representing experiments with ESRP, ESR and in-memory CR (IMCR). In each line, the three markers, from left to right, represent experiments with 1, 3, or 8 redundant copies, and also 1, 3 or 8 simultaneous node failures for Fig. 2b, to the right. ESR results are the same for all checkpointing intervals in each plot because they are equivalent to ESRP results with $T = 1$, and are displayed along data for ESRP and IMCR for comparison. The markers represent the median for results in all locations (cf. Tab 2) and repetitions.
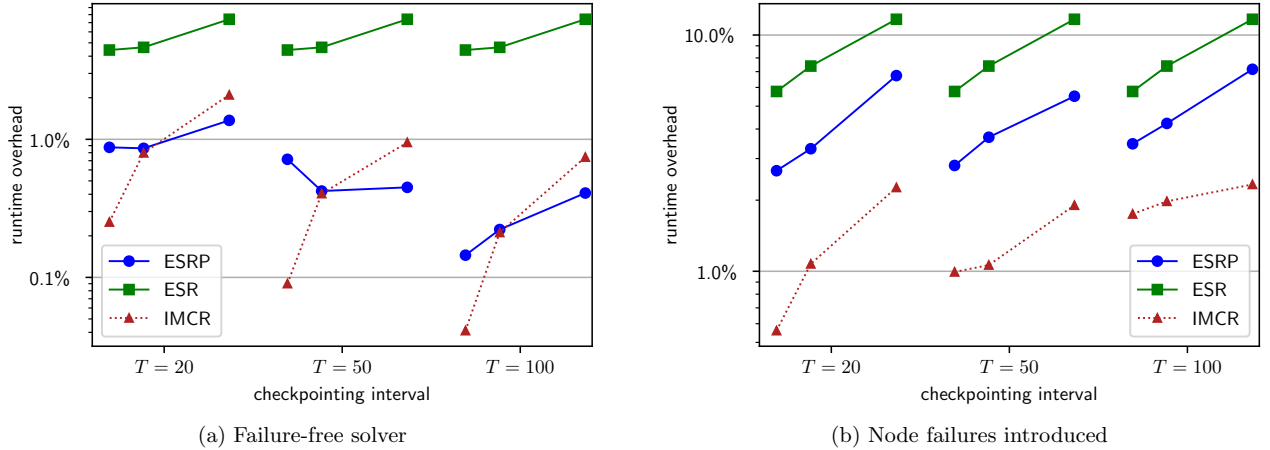
Figure 3: Median runtime overhead for the experiments with the matrix `audikw_1`. See the caption of Fig. 2 for details on the structure of the plot.

of ESRP, we compute the vector $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(End)}$, where the superscript *End* represents the vectors' state after convergence, and use it to compute the *residual drift* metric:

$$\frac{\|\boldsymbol{r}^{(End)}\|_2 - \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(End)}\|_2}{\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(End)}\|_2}. \qquad (2)$$

A more positive value of the residual drift indicates a smaller value of $\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(End)}\|_2$ and, thus, a more accurate result. This metric is not used to determine convergence of the solver; for this, we use the relative residual as described in Section 5. The residual drift is computed only after the solver has converged. We use this metric to ensure that ESRP is not generally less accurate than PCG.

PCG and ESRP experiments without node failures produce the same value for the metric since they always follow exactly the same trajectory. With node failures, the residual drift depends on the selection of affected nodes and on the iteration when the node failure occurs; in this case, we present the minimum and median for all experiments. Accuracy results are summarized in Table. 4. In the median, ESRP with node failures does not differ significantly from PCG. As for the minimum value, the results for the matrix `Emilia_923` show little accuracy loss with respect to PCG, but for the matrix `audikw_1`, there is a drift of close to 15.5%, where PCG has a drift of close to 8%. We do not consider this to be a significant issue. The slight advantage for ESRP in the median case

for audikw_1is explained by the fact that it reconstructs the iterand from the residual vector, thus making the residual and iterand consistent with each other.

Table 4: Residual drift observed in the experiments. *Reference*: **Residual drift for all failure-free cases.** *Median*: **Median residual drift over all experiments with node failures.** *Minimum*: **Minimum residual drift over all experiments with node failures, representing the greatest loss of accuracy during the reconstruction in ESRP.**

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

6 CONCLUSIONS AND PERSPECTIVES

In this paper, we introduce an extension to the exact state reconstruction (ESR) algorithm for node-failure resilience for PCG. Our approach reduces the runtime overhead of ESR by saving redundant copies not in every iteration, but only every T iterations. Given the relation of this approach to checkpoint-restart (CR), we call such a strategy an algorithm-based checkpoint-restart method. We introduce a framework for our experiments and evaluate this new strategy, which we call ESRP, comparing it to standard ESR and also to our implementation of in-memory buddy CR (IMCR). In our experimental results, the runtime overhead of ESRP turns out considerably lower than that of standard ESR and also lower than that of IMCR for the failure-free cases, When node failures occur, however, the recovery time is dominated by the solution of a smaller inner linear system and depends on the matrix itself.

An important step to take in future work is to evaluate ESRP using different preconditioners. Furthermore, we are working on producing larger test problems, so that we can observe a different regime in the computation/communication ratio for PCG. Another interesting direction is the study of ESRP working with partitioning algorithms, looking in particular for partitioning strategies that optimize for the matrix-vector product and simultaneously provide sufficient redundancy. In the future, we intend to produce an implementation of the algorithms on a framework that can detect node failures and provide replacements, using tools like ULFM [3] and similar, for example.

REFERENCES

[1] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Jean Roman, and Mawussi Zounon. 2016. Numerical recovery strategies for parallel resilient Krylov linear solvers. *Numer. Lin. Algebra. Appl.* 23, 5 (2016), 888–905.
[2] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan,

Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2019. *PETSc Users Manual.* Technical Report ANL-95/11 - Revision 3.12. Argonne National Laboratory. https://www.mcs.anl.gov/petsc
[3] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. 2013. Post-failure recovery of MPI communication capability: Design and rationale. *Int. J. High Perform. Comput. Appl.* 27, 3 (2013), 244–254.
[4] Jiajun Cao, Kapil Arya, Rohan Garg, Shawn Matott, Dhabaleswar K. Panda, Hari Subramoni, Jérôme Vienne, and Gene Cooperman. 2016. System-Level Scalable Checkpoint-Restart for Petascale Computing. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS).* 932–941.
[5] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov.* 1, 1 (2014), 4–27.
[6] Sourav Chakraborty, Hari Subramoni, Adam Moody, Akshay Venkatesh, Jonathan Perkins, and Dhabaleswar K. Panda. 2015. Non-Blocking PMI Extensions for Fast MPI Startup. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.* 131–140.
[7] Zizhong Chen. 2011. Algorithm-based Recovery for Iterative Methods Without Checkpointing. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing.* 73–84.
[8] John T. Daly. 2006. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future. Gener. Comp. Sy.* 22, 3 (2006), 303–312.
[9] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (2011), 1:1–1:25.
[10] Brian Gough. 2009. *GNU Scientific Library Reference Manual* (3rd ed.). Network Theory Ltd.
[11] Thomas Herault and Yves Robert. 2015. *Fault-Tolerance Techniques for High-Performance Computing* (1st ed.). Springer Publishing Company, Incorporated.
[12] Atsushi Hori, Kazumi Yoshinaga, Thomas Herault, Aurélien Bouteiller, George Bosilca, and Yutaka Ishikawa. 2020. Overhead of using spare nodes. *Int. J. High Perform. Comput. Appl.* 34, 2 (2020), 208–226.
[13] Kuang-Hua Huang and Jacob A. Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. Comput.* C-33, 6 (1984), 518–528.
[14] Markus Huber, Björn Gmeiner, Ulrich Rüde, and Barbara Wohlmuth. 2015. Resilience for Exascale Enabled Multigrid Methods. arXiv:1501.07400.
[15] Julien Langou, Zizhong Chen, George Bosilca, and Jack Dongarra. 2007. Recovery Patterns for Iterative Methods in a Parallel Unstable Environment. *SIAM J. Sci. Comput.* 30, 1 (2007), 102–116.
[16] Markus Levonyak, Christina Pacher, and Wilfried N. Gansterer. 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.* 81–92.
[17] Hatem Ltaief, Marc Garbey, and Edgar Gabriel. 2006. Parallel fault tolerant algorithms for parabolic problems. *Euro-Par 2006 Parallel Processing* (2006), 700–709.
[18] Message Passing Interface Forum. 2017. User Level Failure Mitigation. http://fault-tolerance.org/.
[19] Carlos Pachajoa and Wilfried N. Gansterer. 2017. On the Resilience of Conjugate Gradient and Multigrid Methods to Node Failures. In *Euro-Par 2017: Parallel Processing Workshops - Euro-Par 2017 International Workshops (Lecture Notes in Computer Science)*, Vol. 10659. Springer, 569–580.
[20] Carlos Pachajoa, Markus Levonyak, and Wilfried N. Gansterer. 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).* 49–58.
[21] Carlos Pachajoa, Markus Levonyak, Wilfried N. Gansterer, and Jesper Larsson Träff. 2019. How to Make the Preconditioned Conjugate Gradient Method Resilient Against Multiple Node Failures. In *Proceedings of the 48th International Conference on Parallel Processing.* Article 67, 10 pages.
[22] Carlos Pachajoa, Christina Pacher, and Wilfried N. Gansterer. 2019. Node-Failure-Resistant Preconditioned Conjugate Gradient Method without Replacement Nodes. In *9th IEEE/ACM Workshop on Fault Tolerance for HPC at eXtreme Scale, FTXS@SC 2019.* IEEE, 31–40.
[23] Daniel A. Reed and Jack Dongarra. 2015. Exascale Computing and Big Data. *Commun. ACM* 58, 7 (2015), 56–68.
[24] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics.
[25] Dingwen Tao. 2018. *Fault Tolerance for Iterative Methods in High-Performance Computing.* Ph.D. Dissertation. UC Riverside.
[26] Keita Teranishi and Michael A. Heroux. 2014. Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM. In *Proceedings of the 21st European MPI Users' Group Meeting.* 51–56.
[27] Henk A Van Der Vorst and Qiang Ye. 2000. Residual replacement strategies for Krylov subspace iterative methods for the convergence of true residuals. *SIAM J. Sci. Comput.* 22, 3 (2000), 835–852.
[28] John Young. 1974. A first order approximation to the optimum checkpoint interval. *Commun. ACM* 17, 9 (1974), 530–531.