Noname manuscript No.
(will be inserted by the editor)

A minimally intrusive low-memory approach to resilience for existing transient solvers

Chris D. Cantwell · Allan S. Nielsen

Received: date / Accepted: date

Abstract We propose a novel, minimally intrusive approach to adding fault tolerance to existing complex scientific simulation codes, used for addressing a broad range of time-dependent problems on the next generation of supercomputers. Exascale systems have the potential to allow much larger, more accurate and scale-resolving simulations of transient processes than can be performed on current petascale systems. However, with a much larger number of components, exascale computers are expected to suffer a node failure every few minutes. Many existing parallel simulation codes are not tolerant of these failures and existing resilience methodologies would necessitate major modifications or redesign of the application. Our approach combines the proposed user-level failure mitigation extensions to the Message-Passing Interface (MPI), with the concepts of message-logging and remote in-memory checkpointing, to demonstrate how to add scalable resilience to transient solvers. Logging MPI communication reduces the storage requirement of static data, such as finite element operators, and allows a spare MPI process to rebuild these data structures independently of other ranks. Remote in-memory checkpointing avoids disk I/O contention on large parallel filesystems. A prototype implementation is applied to Nektar++, a scalable, production-ready transient simulation framework. Forward-path and recovery-path performance of the resilience algorithm is analysed through experiments using the solver for the incompressible Navier-Stokes equations, and strong scaling of the approach is observed.

Keywords exascale · fault tolerance · message-logging · MPI · transient solvers · parallel computing

ExaFLOW Project: EU Horizon 2020 grant agreement number 671571

C. D. Cantwell
Imperial College London
Tel.: +44 20 759 45123
E-mail: c.cantwell@imperial.ac.uk

A. S. Nielsen
École polytechnique fédérale de Lausanne
Tel.: +41 21 69 35905
E-mail: allan.nielsen@epfl.ch

1 Introduction

One of the drivers for exascale computing is the requirement to solve tightly-coupled computational problems of unprecedented scale in order to gain new and improved insight into complex physical phenomena. This need spans scientific domains such as industrial fluid dynamics, climate modelling, energy and personalised medicine. Models in these application areas are often mathematically formulated in terms of time-dependent partial differential equations, and their computational implementations utilise domain decomposition techniques and message passing paradigms for communication in order to run on massively parallel distributed computing systems. A large number of, often highly-complex, production codes already exist to numerically solve these problems. Such simulations, by their transient nature, are frequently long-running, executing for several days or weeks. Exascale systems, when they arrive, have the potential to allow much larger, more accurate and scale-resolving simulations to be performed, the results of which will have significant scientific and societal impact. However, there are several major hurdles associated with practically using an exascale supercomputer, which need to be overcome if codes are to scale efficiently on very large numbers of processing cores so that such simulations remain cost-effective.

Algorithm and software resilience is now one of the greatest concerns in striving towards exascale and interruption, due to component failure, is now considered a major barrier to effectively using an exascale system with current numerical codes [9, 28]. Both hardware and software errors, such as component failures or operating system crashes, may interrupt simulations or lead to non-deterministic results [27]. This is further exacerbated by the trend towards heterogeneous computing where nodes are composed of multiple processing components and additional system-level software layers. Failures typically necessitate a restart of the computation and results in wasted time, energy and resources. Even with the use of high-quality hardware, the number of components necessary to reach this level of throughput leads to an overall system failure rate of once every few hours.

The need for resilience is already evident in the current generation of petascale supercomputers with some recent systems having a mean-time-between-interrupts (MTBI) of just a few hours. For example, the CPU-only portion of the Cray hybrid Blue Waters system (22,640 nodes, 5.66 petaflops) is reported to have a MTBI of 8.6 hours [13], while the MTBI of 8192 nodes of the Tianhe-2 supercomputer (equivalent to 17.33 PF) is just 2 hours [10]. In contrast, the Cray XK6/XK7 (Titan) at Oak Ridge National Laboratory (10-20 / 27 petaflops) achieves a MTBI of 132/173hrs [2]. The anticipated failure rate of an exascale machine is likely to be higher than present systems [8, 9, 28, 23] and therefore application resilience is critical in maintaining the usefulness of any future exascale system.

Minimising data movement at all levels within a system is an increasingly important consideration. Exascale machines are likely to be characterised by high flop-rates, high-parallelism and high costs to moving data within or between nodes (in terms of performance and energy). Memory will also be increasingly hierarchical, incorporating newer technologies such as NVRAM, but with limited memory close to the CPU. This consequently constrains the nature of resilience mechanisms which can be employed.

A number of techniques already exist to improve the resilience of application codes in the event of system degradation or failure, including checkpoint/restart, redundant computing and application-based resilience. These methods can be classified as either forward recovery or backward recovery. In the former, the algorithm continues and corrects errors introduced by failures. Examples of this include redundant computing or some algorithm-specific approaches. Forward-correcting algorithms exist for some sub-components of the transient solvers considered, such as conjugate gradient solvers [1], but these approaches do not typically provide a comprehensive solution in the case of an error occurring outside of these components. Furthermore, they require a significant intrusion into the application code. In contrast, backward-recovery rolls back to the last previously recorded globally consistent state and repeats calculations. A check-point is a snapshot of the application state at a point during an application's execution. Coordinated checkpointing to stable storage is the most common backward-recovery technique employed in current production transient simulation codes. A globally consistent system state is written to disk periodically, allowing the application to be restarted and continue from the saved state in the event of a failure. This can be achieved at an application level through writing sufficient state data to disk to allow restart, or more transparently at an operating system level by directly dumping memory pages. The second approach, while simple, is typically more costly and many finite element codes, for example, write only the solution state to disk on the basis that the remaining state can be reconstructed easily when the simulation is restarted. Even in this case, the volume of data and resulting I/O contention means that highly parallel codes on current petascale systems might spend more than 25% of their execution time performing check-pointing [27]. Dedicated nodes have been employed for non-blocking check-pointing to minimise the effect of this bottleneck and allow the computation to continue [26].

In-memory checkpoint-restart records a snapshot of the application memory and allows recovery of the simulation in the event of an application fault or detected but unrecoverable error [29,25]. It alleviates the disk I/O contention during normal execution by storing check-points in volatile memory, potentially only writing out to stable storage in the event an error occurs. Remote in-memory checkpoint-restart further allows recovery in the event of a complete node failure. For example, such an approach has been demonstrated with a molecular dynamics code on a large-scale supercomputer in which checkpoints were stored both locally and on a remote node, and showed over two orders of magnitude decrease in checkpoint time and significant reduction in recovery time [31]. Cross-node resilience can also be provided through erasure codes and check-sums, built into certain algorithms, to reduce the storage overhead and these have recently been applied in the case of linear solvers [20,32]. Finally, multi-level check-pointing attempts to balance the performance and resilience capabilities of the above techniques [14,12]

Application-based resilience avoids the cost of restarting a simulation by detecting failures and recovering failed processes. For message-passing interface (MPI) programs, this may involve shrinking existing MPI communicators, or invoking a spare process to take over the failed process. The latter is preferred in the context of domain decomposition methods to avoid the expensive redistribution of work necessary to maintain load balancing. A number of proposals and prototype implementations have already been reported. User-Level Failure Mitigation (ULFM) [3,4] is a proposed extension to the MPI 4.0 standard which adds fault tolerance semantics. ULFM provides three key additions to the MPI API. The `MPIX_Comm_revoke` method invalidates a communicator and allows a process to notify other processes that a failure has occurred, for example to initiate recovery. The MPIX_Comm_shrink method then reconstructs a revoked communicator containing failed processes into a working communicator with those failed processes omitted. Finally, MPIX_Comm_agree implements an agreement algorithm, performing a logical AND operation on the boolean parameter across processes; this succeeds even if there are failed processes.

Other fault tolerance efforts include Reinit, which aims to improve on ULFM for the case of bulk-synchronous codes to allow global backward non-shrinking recovery [22]. However, most of this capability has since been incorporated into the latest ULFM specification. FENIX [18,17] is a library which provides fault tolerance without application shutdown and is built upon the ULFM capabilities. ACR implements Automatic Checkpoint/Restart using replication of processes in order to handle both soft and hard errors. FA-MPI proposes non-blocking transactional operations as an extension to the MPI standard to provide scalable failure detection, mitigation and recovery [19]. Finally, FT-MPI was an earlier precursor to ULFM for adding fault tolerance to the MPI standard [16].

Existing approaches to incorporating resilience in scientific codes involve substantial modifications to the application code in order to add protection mechanisms to all the necessary data structures and, in some cases, may require a complete redesign. Many of the demonstrators of these approaches are stencil applications, written specifically for illustrating the resilience algorithm and are not necessarily representative of production codes. In this paper, we instead outline a novel low-intrusion application-based resilience approach, building on ULFM, specifically for tightly-coupled transient solvers. We illustrate our strategy through a prototype implementation in Nektar++, a production-ready high-order spectral/hp element framework for the solution of a wide range of partial differential equations [5]. Our approach meets the following objectives:

- Necessitate minimal application code intervention or code redesign to allow resilience to be easily added to existing codes;
- Allow the simulation to continue in the event of one or more concurrent hard failures with limited interruption to surviving processes;
- Incur minimal forward-path overhead on execution performance to effect resilience;
- Ensure strong scalability of the resilience algorithm on massively parallel systems.

2 Algorithm and Implementation

We are particularly interested in exploiting the properties of algorithms used for the solution of time-dependent initial value problems. Such problems involve the evolution of one or more solution variables under the action of a (time-dependent) partial differential equation beginning from some initial state. Without the addition of resilience, the failure of a single MPI process would typically lead to the termination of the entire simulation, requiring a complete restart and backward-recovery from the last checkpoint.

The resilience approach we describe here is independent of the particular numerical discretisation used (finite difference, element, volume, etc), time-integration scheme and the specific time-dependent PDE to be solved. In general, the implementation fo solvers for PDEs can be broken down into two distinct phases. The first is a set-up phase in which the necessary discrete operators or stencils are constructed. These are typically in the form of matrices which remain fixed throughout the simulation. The cumulative storage of such operators is often large, compared to solution vectors, and their construction requires global communication in order to associate neighbouring degrees of freedom across partitions. The second phase is the time-advancement of the initial conditions to the final state. The value of the solution variables change during time integration but the discrete operators do not. An exception might be if adaptivity of the computational discretisation is employed, which is discussed later. The data generated during the first phase will be referred to as *static* data, while the time-evolving data in the second phase will be referred to as *dynamic* data.

In the event of a process failure (e.g. due to a hardware fault) we would like to avoid the complete restart of the simulation on all processes, avoid checkpointing to disk and instead substitute the failed process with a *spare* process which recovers from data provided by a surviving process in order to continue the calculation.

2.1 Initialisation of spare processes

To leverage the capabilities of ULFM, we provide a custom error handler on all communicators, rather than allowing MPI to automatically defer to calling `MPI_Abort` when a failure occurs. Execution of the application proceeds as normal but with a number of additional ranks allocated. The set of all processes are partitioned into *worker* and *spare* ranks immediately after MPI is initialised using `MPI_Comm_split`. This creates a sub-communicator in which the worker processes participate. Spare processes are assigned to a null communicator and wait until a failure occurs. They should proceed only on two events: a failure occurring, or; the application terminating normally in which case `MPI_Finalise` must be called. To achieve this on spare processes we call `MPI_Comm_agree` with a value of *true* immediately after `MPI_Init`. On worker threads, the same call is made immediately prior to `MPI_Finalise` with a value of *true* or, in the event of an error being detected, from the error handler with a value of *false*. Therefore, if the resulting conjugation evaluates to true, the application code must have completed successfully. If it evaluates to false, this implies a process has failed, identifying that recovery is required.

2.2 State Protection

In order to enable recovery on a replacement MPI process, the data structures within the code must be protected in a way which allows their reconstruction on a spare process. This is traditionally achieved in transient codes by check-pointing the dynamic data to disk. Static data is not stored as this is regenerated when the simulation is restarted. Checkpoints must be written out periodically and a balance sought between anticipated failure rate and checkpoint frequency [11]. However, disk checkpointing is not scalable and will be infeasible for the purpose of resilience at exascale.

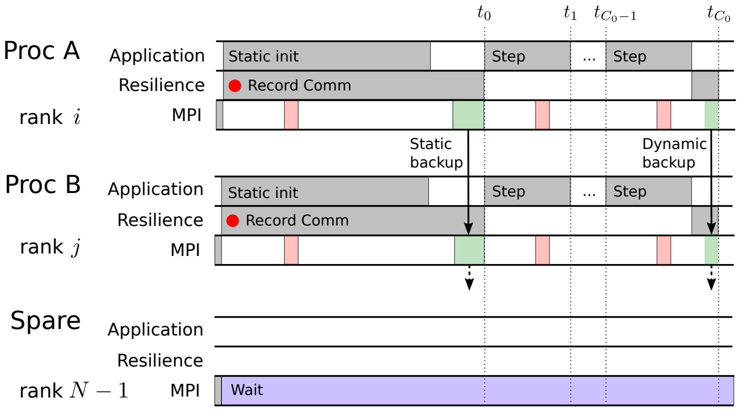

Fig. 1 Diagrammatic representation of the protection algorithm. Initialisation of solver, showing three processes – two active and one spare – with ranks i, j, and N, respectively. Rank i communicates static recovery data to rank j. After a number of steps, at time t_{C_0}, a remote in-memory checkpoint occurs. The spare rank, N remains idle throughout. Red MPI regions denote collective communication, while green regions denote pairwise communication.

State protection is depicted diagrammatically in Fig. 1. In our approach, rather than writing to disk, we opt for remote in-memory check-pointing. Data on each process is backed up to a partner process which requires only pairwise communication and is denoted by green blocks in Fig. 1. This 'buddy' process is chosen to balance resilience and performance and is typically on an adjacent node. Depending on network topology and cluster configuration, a more distant node may improve resilience to some less common failures, such as power distribution faults[14]. However, such optimisations are beyond the scope of this study.

To perform on-the-fly independent recovery on a spare node, we will need both the static and dynamic components of the data. However, depending on the simulation code design and particularly for object-oriented codes, the static data may be scattered throughout the code requiring extensive code modification or redesign to enable backup and restoration of these data structures. We are initially faced with two challenges: how do we backup the static data efficiently; and, how do we do so with minimal code intervention? We can address both concerns using the concept of message-logging during the initialisation phase of the solver. We do not store the initialised static data structures themselves, but rather the outcome of any MPI communications performed by the application during the static phase, indicated by *Record Comm* in Fig. 1. This can be achieved by intercepting calls to the MPI API and logging the result, if applicable. This provides two key advantages: the volume of data is anticipated to be smaller than the fully initialised data structures whose generation invoked the communication, since exchanges occur along partition boundaries rather than within the partition volume, and; very little modification is required to the existing application code. To complete this aspect, we must annotate the code (through function calls) to mark the beginning and end of the initialisation phase.

Dynamic data checkpointing is performed at regular intervals after the end of the initialisation phase. These data typically consist of the solution vectors at the time of checkpointing only and are relatively small in size compared to static data.

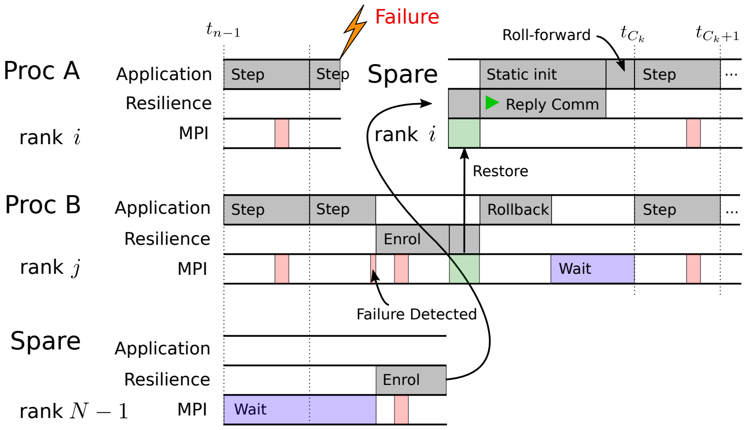

Fig. 2 Diagrammatic representation of the recovery algorithm. Process A with rank i fails and, after enrolment and rank translation during recovery, process S is assigned rank i and receives recovery data from rank j. Static and dynamic data is subsequently recovered to the last checkpoint at time t_{C_k} without requiring any further communication with surviving ranks. The simulation then continues. Red MPI regions denote collective communication, while green regions denote pairwise communication.

These are protected through duplication to the memory of a partner node. Further optimisation or compression of this data is not considered within the scope of this study.

2.3 MPI Communicator Recovery

ULFM allows for several recovery models, namely, *shrink*, *spare* and *respawn*. Shrinking involves reducing the number of processes participating in a simulation, thereby necessitating a redistribution of the work from failed processes. While this can be efficient for some types of problems (e.g. molecular dynamics codes), it is less efficient, for example, in finite element codes utilising domain decomposition due to the reconstruction of operators and mappings. Respawning does not align with the current use of queuing systems with fixed resource allocations, used on most HPC clusters. We therefore pursue the spare invocation approach.

The first task is to modify the behaviour of the MPI routines in the event of a failure. We specify our own error handler function to be called by MPI in the event that any process detects another process has failed through any communicator. The primary role of this handler is: to revoke the communicator, thereby ensuring all other processes become aware of the failure, and; to throw an exception which propagates up the call tree to a suitable point in the application code in which backward-recovery can be managed. For transient simulation codes, this is typically the outer time-integration loop.

The second task is to enrol spare processes to replace those which have failed and rebuilding all communicators used in the application code. A call to `MPI_Comm_agree` unlocks the spare processes (see above) so they can participate in the enrolling process. The set of all processes is then shrunk to omit those which have failed, MPI group operations are used to determine the failed ranks, and spare processes are reassigned the ranks of the failed processes using `MPI_Group_translate_ranks`. A complete set of worker processes is then selected using `MPI_Comm_split` and any other sub-communicators are similarly reconstructed.

2.4 State Recovery

The application must now achieve a globally consistent state on all processes. Surviving processes simply rollback their dynamic data to the last in-memory checkpoint. Spare processes must recover both the static and dynamic data. This aspect of the algorithm is shown in Fig. 2.

Recall that spare processes wait at the beginning of execution, shortly after the initialisation of MPI, until a failure occurs or the application terminates normally. Recovery of the static data on an initialising spare process then proceeds as follows. The MPI message log is first retrieved from the state backup, located on the surviving "buddy" process. The application code then proceeds with the initialisation phase as normal. When communication operations are performed, they are instead intercepted and the result of the call is returned directly from the message log, rather than invoking calls to the MPI library itself. Surviving worker processes are at a different point in the code execution and deadlock would occur if collective communication was attempted. This strategy allows the recovering process to initialise the static data structures completely independently of all other surviving processes with very minimal code changes.

Upon commencing time-integration, the dynamic data is rolled forward from the dynamic data checkpoint, resulting in a working replacement and an application which is in a globally consistent state, whereby execution can continue.

2.5 Implementation

A prototype implementation of the algorithm has been developed in Nektar++. This package comprises of a set of libraries, written in C++, which implement the spectral/hp element method in an efficient manner which aligns with their mathematical formulation and current computer architectures [30,7,6,24], along with a collection of physics solvers which build upon these libraries. These transient solvers can tackle a wide range of problems involving incompressible and compressible fluid dynamics, combustion, oceanic models using shallow water equations and biomedical problems in arterial flow and cardiac electrophysiology. Many of these applications require high-resolution scale-resolving simulations which are ideally targeted towards massively parallel distributed clusters.

Nektar++ is heavily object-oriented and much of the static data is encapsulated within classes and other rich data structures. The static data generated by Nektar++ includes elemental high-order basis functions, integration weights, geometric information, per-element matrices representing the necessary finite element operators and the global data structures required for applying the conjugate gradient algorithm to the complete problem. In particular, these global structures include an assembly operator for each solution field, represented as a surjective map, which associates each local degree of freedom to a corresponding degree of freedom in the global system. The construction of each of these maps requires global collective communication in the form of a gather-scatter operation, implemented by an external library. Other collective communication includes mesh partitioning, construction of finite element operators and preconditioners and ensuring consistent enforcement of boundary conditions in parallel, as well as auxiliary functions such as collecting time-series data from specific coordinates in the domain.

One advantage of the design of Nektar++ is that all direct MPI operations are managed through a single C++ class, which allowed rapid prototyping of the resilience algorithms. Code was added to perform the message logging, message replay and exchange of static- and dynamic-data remote in-memory checkpoints to effect the resilience capabilities described above. Command-line parameters enable the user to specify the number of spare processes, S to be reserved dynamically at run-time from the total of N ranks requested and the rank offset k to be used for storing state preservation data. The last S ranks are, by default, assigned to be spares while the first $W = N - S$ ranks are allocated as workers. Each worker rank r sends state preservation data to rank $(r + k) \mod W$, where $0 < k < W$ is the stride. The value of k can be set to the number of ranks on a node or chassis, to improve the resilience based on specific cluster configuration. Rank r also receives state preservation data from rank $(r + W - k) \mod W$.

Performance is analysed using the solver for the incompressible Navier-Stokes equations, given by

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \frac{1}{Re}\nabla^2 \mathbf{u},$$
$$\nabla \cdot \mathbf{u} = 0.$$

In summary, these equations are solved using a velocity-correction high-order splitting scheme in which the pressure is first solved as a Poisson problem and the velocity then adjusted to enforce the incompressibility constraint through a series of Helmholtz problems [21]. The solutions to the pressure and velocity systems are obtained using the preconditioned conjugate gradient method. Full details of the implementation are described elsewhere [5].

3 Performance Analysis

To demonstrate the efficacy of our approach we measure performance characteristics of the resilience algorithm on a UK Tier-2 High-Performance Computing system. We first outline our test problem and environment and then discuss the memory usage and performance considerations of our implementation.

3.1 Test problem and environment

The specific test problem we consider is laminar flow in a rectangular duct of height D, streamwise length $20D$ and of width $10D$. This is an intentionally trivial problem designed to demonstrate the effectiveness and scalability of the resilience algorithm under a relatively non-intensive computational load. A plug inlet velocity profile is prescribed with a Reynolds number of 10, based on the duct height and inlet velocity. The domain is discretised into regular hexahedral spectral/hp elements of size $0.4D \times 0.4D \times 0.04D$ in the streamwise, spanwise and cross-stream directions, respectively. The computational mesh consists of 31,249 hexahedral elements, each using a polynomial order of 3 in each coordinate direction, giving 64 local modes per element. Accounting for three velocity components and a pressure field, this gives a total of 3.37M degrees of freedom. A low-energy block preconditioner is applied to the velocity and pressure systems to accelerate the convergence of the conjugate gradient solver. Disk checkpointing in Nektar++ is achieved by writing a compressed binary file per process to the parallel filesystem.

The test system used is an SGI ICE XA system with 10,080 cores. Each of the 280 nodes is equipped with dual 2.1Ghz, 18-core Intel Xeon processors with hyper-threading enabled and 256GB RAM. All nodes are connected using a single Infiniband fabric and share a common Lustre parallel filesystem. Tests are performed across a range of working process counts, spanning two nodes, up to sixty-four nodes, or $NP_{\max} = 2304$ processes. One process was assigned per hardware core (up to 36 per node) and MPI bind-to-core was used to reduce the effects of inter-socket memory bandwidth contention and lower-level cache thrashing. For all fault tolerance tests, an additional node was allocated to ensure the number of working processes remained the same and the offset value was set at $k = 36$ to ensure state preservation data was stored on a different node.

In the results presented below timings are given as mean ± standard deviations, measured across all processes for five independent simulations. State protection size measurements are given as mean ± standard deviation across all processes for one simulation, since memory usage is identical for repeated simulations.

3.2 Memory overhead

Fig. 3(a) shows a breakdown of per-process memory usage for the major components of the fault-tolerant simulation code. *Static data* refers primarily to the matrix operators and element mappings which are constructed during initialisation of the solver and remain unchanged throughout the time-integration phase. This is the most significant component of the solver memory usage, particularly for lower core counts. At higher core counts, this quantity begins to saturate due to the emerging dominance of data structures whose size is independent of the sub-problem size being tackled by that process.

The *static backup* component is the memory occupied (and subsequently transferred over the network) in providing resilience for the static data on a partner process. Specifically, this comprises the message logs from all MPI communications which were undertaken during the initialisation phase of the originating process. This decreases steadily with process count up to the limit of the number of available cores. In particular, for higher core counts, this quantity continues decreasing. The relative magnitude of the *static backup* size to the original *static data* size is presented in Fig. 3(b) as a percentage. The difference in storage requirements drops considerably at higher core-counts and is $\approx 6\%$ at $NP_{\max}$.

The *dynamic backup* component represents the memory occupied by the copy of the solution vectors which are being advanced in time; therefore this data changes at each time-step. Since this is only vector data, the size is over two orders of magnitude smaller than the size of the static backup. The cost of storing the process's dynamic data and the backup of a partner process's dynamic data backup is approximately equal since, in the current implementation, no compression or

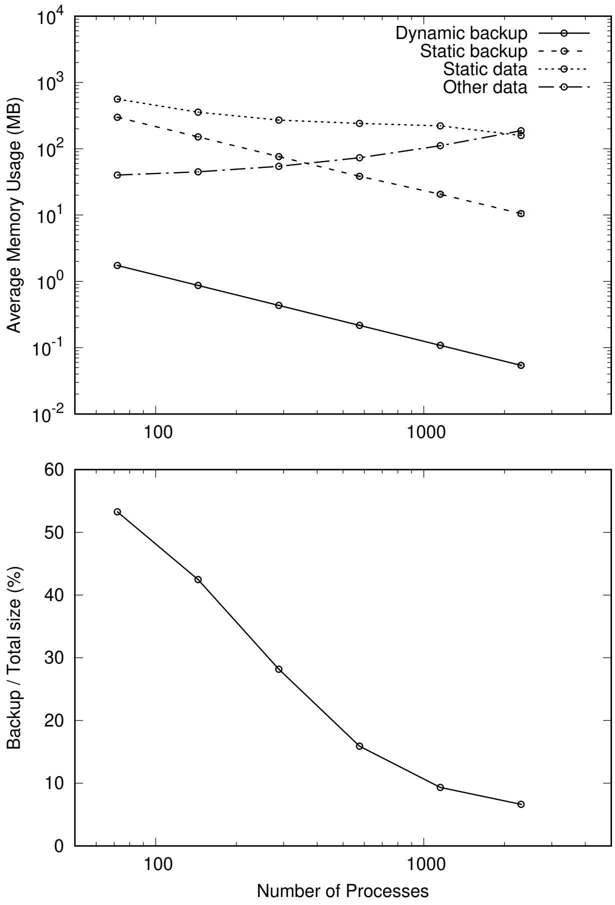

Fig. 3 (a) Per-process memory usage as a function of number of processes. (b) Percentage size of static data backup vs full backup.

erasure codes are applied to the checkpointing process and the work is equally distributed.

Finally, the *other data* component relates to the memory occupied by code and MPI initialisation. The latter increases with increasing core counts and becomes one of the dominant memory costs for the largest core counts.

3.3 Checkpointing Performance

Remote in-memory checkpoint time for the dynamic data was measured on all processes. Fig. 4 (solid line) shows strong scaling results of measured in-memory checkpoint time, as a function of the number of processes. Checkpoint time monotonically decreases with increasing parallelism, due to reduced data volume per process, with little sign of saturation up to the limit of the available cores. In contrast, disk checkpointing (dashed line in Fig. 4) saturates around 288 cores and

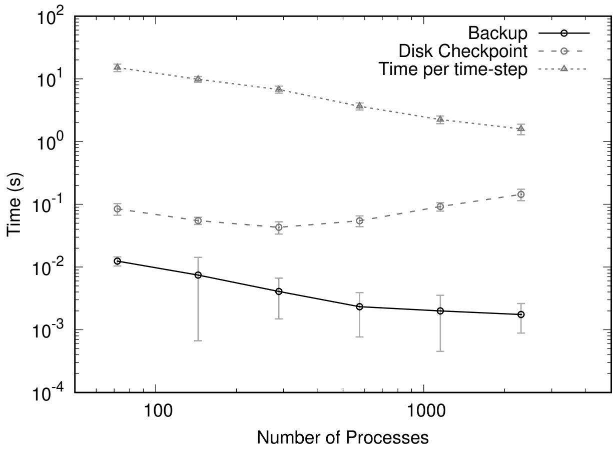

Fig. 4 Remote in-memory dynamic data checkpointing time for test problem (solid line), compared with disk checkpointing (dashed line). For comparison the execution time per time-step, without any form of checkpointing, is shown (dotted line).

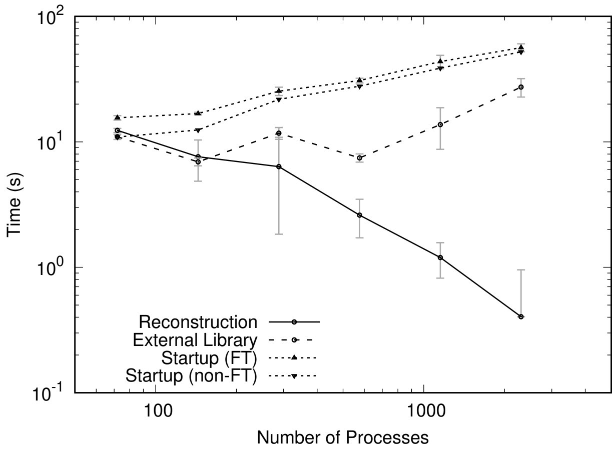

Fig. 5 Recovery time as a function of number of processes, broken down into the time to repair communication and regenerate static data and, separately, the time taken to reinitialise the external gather-scatter library. Time for initial simulation start-up is shown for comparison, both for the original and fault-tolerant versions of the code.

increases for larger core counts. Above 2k cores, disk-checkpoint time is approximately two orders of magnitude greater than remote in-memory check-pointing. As a point of comparison, we also show the execution time per time-step, measured as the wall-time of advancing the PDE by one time-step without any form of checkpointing.

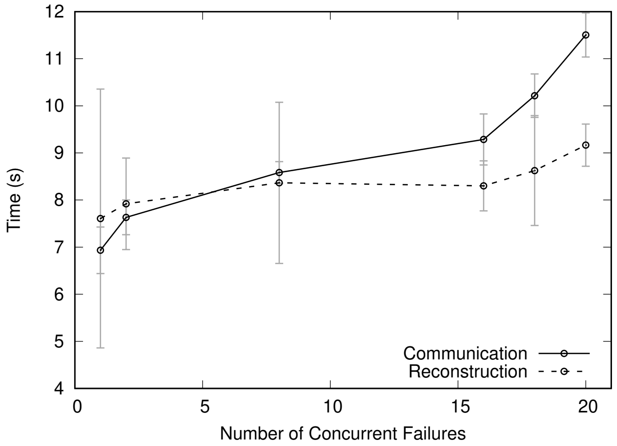

Fig. 6 Restore time as a function of number of processes, broken down into communication and static data regeneration for multiple concurrent failures.

3.4 Recovery Performance

Recovery time describes the time from the detection of a process failure to the operational recovery of the system, at the point where the simulation can continue. This duration is typically a function of a number of variables, including problem size per process, number of cores and frequency of check-pointing. To simplify analysis of the resilience mechanism, the latter has been eliminated.

Fig. 5 shows the time taken to recover following a process failure, as a function of the number of cores. Reconstruction (solid line) includes the communication of the state preservation data, reinitialisation of MPI communicators and execution of the initialisation phase of the application code to regenerate the static data. During initialisation, MPI operations utilise the result from the message log rather than performing actual communication and therefore the process recovers completely independently. This part of the algorithm scales well and takes less than half a second at $NP_{\max}$. Since MPI communication during the initialisation of the gather-scatter external library objects could not be logged by the prototype implementation in Nektar++, these are reinitialised exactly as for the normal start-up and shown separately (dashed line) in Fig. 5. Until such communication can be included in the message log, this is the dominant cost of recovery at larger core counts.

As a point of reference, we also show in Fig. 5 the start-up time for the simulation using both the fault-tolerant implementation and the original unmodified version. The overhead of message-logging and the exchange of state preservation data can be seen to have limited impact on the start-up performance.

3.5 Concurrent failures

It is often standard practice, particularly for those codes which do not support hybrid parallelism of the form MPI+X, to execute multiple MPI ranks on a single node. This leads to a situation in which a fault of the node would cause multiple ranks to fail concurrently. This scenario is supported by the ULFM paradigm and multiple spares will be enrolled to replace failed ranks. Figure 6 shows experiments performed to examine the performance of recovery in the event of multiple concurrent failures. All failed ranks are co-located on the same node. Communication costs increase slightly with the number of failed ranks due to the increasing volume of recovery data being exchanged. Variability due to network contention is more evident with the linear scale. While the performance of static data reconstruction is mostly independent of the number of failures, there is a slight increase in the time taken to complete restoration with increasing numbers of failures due to contention within the node.

4 Discussion

This study outlines a minimally intrusive, efficient and scalable approach to adding resilience to existing time-dependent solvers, in which process data can be partitioned into static and dynamic components. Our approach is demonstrated within the Nektar++ spectral/hp element framework. In contrast to the more general design of existing approaches, we tailor our approach specifically to target time-dependent solvers, thereby checkpointing only dynamically evolving data and using message-logging [15] to allow the static data to be locally reconstructed. This moves significantly less data than a direct recovery of finite element operators, and with minimal disruption to surviving processes. We have demonstrated promising performance characteristics of this strategy for use with massively parallel simulations, where fault tolerance will become an essential ingredient of numerical algorithms and software. There are several key advantages to the approach described, which are discussed below.

Static data are protected through retention of the outcome of MPI communication exchanges during the initialisation phase. This brings two benefits. The first is that the volume of recovery data is typically less than that of the original data, reducing storage requirements and consequently improving intra-node and inter-node performance by reducing data movement costs. The reason for this in part stems from the fact that communication typically occurs on the boundary of partitions for which the data is of one dimension lower than the volumetric data. Consequently, for three-dimensional problems, the static data grows as $\mathcal{O}(n^3)$ while the boundary data only grows as $\mathcal{O}(n^2)$. This trend can be clearly observed in Fig. 3(a).

The second advantage is that existing codes require very little modification in order to augment them with this resilience capability. Since MPI communication occurs through the MPI API, these calls can be intercepted to inject the resilience layer - logging outputs during the initial execution, and replying those outputs during any subsequent recovery. This is a significant advantage for large object-oriented codes, such as Nektar++, in which raw data are often encapsulated within layers of class hierarchies and other language semantics, and are not easily accessible from a single central location without substantial rewriting of the application.

Performance and efficiency of the algorithm is critical to minimise the impact of the resilience layer on the normal execution of the code. All state preservation operations which occur during the normal error-free state are pairwise and do not involve the use of any collective communications. This allows the algorithm to scale with the volume of data per process, independent of the number of processes. Coordinated communication is only required during the restoration process in order for communicators to be repaired consistently and is independent of the volume of data per process. Following restoration of the communicators, the spare process receives all the preserved state data in a single pairwise communication and recovers independently of all other processes. These aspects of the algorithm therefore scale with the number of processes and data per process respectively, but independent of the total number of degrees of freedom.

The implementation was assessed through a medium-sized test problem of viscous flow in a square duct. For the largest runs considered up to $NP_{\max}$, this corresponded to an average of just 13 elements per process, or 1430 degrees of freedom. At this level of granularity, the parallel efficiency of the simulation itself drops considerably. Dynamic check-pointing is the most critical performance consideration as it occurs frequently throughout the simulation. It was shown to strong-scale up to $NP_{\max}$ and, as expected, compares favourably against traditional parallel file-system check-pointing (Fig. 4). In contrast, the static data backup occurs only once, and static data restoration occurs only in the event of a failure. Reconstruction of the static data was also shown to scale well up to $NP_{\max}$ (Fig. 5).

4.1 Topology-aware algorithms

The prototype implementation described here places the partner process at a rank-offset specified at runtime. Assuming ranks are allocated sequentially by core and then by node, this allows the user to ensure data is backed up to a node which is different to the one on which the original process resides. This also allows for increasing resilience when there are higher-level clusterings of processes, allowing backups to span multiple chassis or racks. However, for complex multi-layered network topologies, where communication between all nodes is non-uniform, a balance between performance and resilience may need to be sought. In such cases, a grouping of ranks based on topology may be more appropriate.

This consideration also applies to the placement of spare ranks. The current implementation assumes uniform communication performance between nodes and the highest S ranks are reserved as spares. For non-uniform topologies, distributing spares throughout the range of ranks may be a more performant strategy and allow for the selection of a topologically nearest spare to be chosen in the event of failure.

4.2 Limitations

A number of assumptions are inherent in the algorithm developed. The most significant of these is that a large portion of the memory required during simulation contains static data, such as finite element operators, which are created at the beginning of the simulation and subsequently do not change during execution. This inherently creates challenges for applying this approach to simulations where the domain or computational mesh is modified in time, such as arbitrary Lagrangian-Eulerian (ALE) methods, r-adaptation (mesh movement) or h-adaptation (mesh adaptivity). In the first two cases, the geometry-dependent component of the finite element operators is time-dependent and that portion would need to be included as part of the dynamic data. For mesh adaptive codes, message-logging could be used and the static backup could be regenerated with each mesh adaptation. The efficacy of this would depend on the frequency of mesh adaptation and the likelihood of a failure occurring. However, neither ALE, r-adaptivity or mesh adaptation are currently supported by Nektar++ so these cases have not been explored further.

Related to this, non-determinism of simulations is only supported where the non-determinism forms part of the dynamic data (for example, the initial condition) or is incorporated into static backup as a consequence of MPI communication. The algorithm could be extended to allow non-deterministic components to be manually included with the static backup at the cost of increasing the invasiveness of the approach.

A further limitation arose due to the external nature of the gather-scatter library, GSLib, used for assembling distributed finite element operators. Since this code is a third-party library, it directly calls the MPI API without passing through the prototype resilience layer developed within Nektar++. The recovery of communication within the application code includes repairing all MPI communicators, but necessarily required reinitialising the gather-scatter library, which involves collective communication. This was therefore shown separately in Fig. 5.

4.3 Future Work

The resilience algorithm was prototyped through modification of the communication classes within Nektar++. While this allowed all implementation to be contained within a single section of code, it prevented third-party libraries taking advantage of the algorithm. Future work will focus on extracting the resilience algorithm to an independent library which will intercept MPI API calls for both application and any third-party library codes. This should eliminate the performance bottleneck introduced by the gather-scatter library during recovery (dashed line, Fig. 5) and allow strong scaling of the approach to be retained to many thousands of processes.

Acknowledgements This work was funded by the EU Horizon 2020 project ExaFLOW. We are grateful to the other partners in the ExaFLOW project for their support and helpful suggestions. This work used EPCCs Cirrus HPC Service
(https://www.epcc.ed.ac.uk/cirrus).

References

1. Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Jean Roman, and Mawussi Zounon. Towards resilient parallel linear krylov solvers: recover-restart strategies. Technical Report RR-8324, INRIA, 2013.
2. Ashley D. Barker, David E. Bernholdt, Arthur S. Bland, James J. Hack, Douglas L. Hudson, James H. Rogers, T. P. Straatsma, Kevin G. Thach, Sudharshan S. Vazhkudai, Jack C. Wells, and Julia C. White. High performance computing facility operational assessment

2013 oak ridge leadership computing facility. Technical report, Oak Ridge National Laboratory, 2014.

3. Wesley Bland, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. A proposal for user-level failure mitigation in the MPI-3 standard. Technical report, 2 2012.
4. Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of MPI communication capability. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 1 2013.
5. Chris D. Cantwell, David Moxey, Andrew Comerford, Alessandro Bolis, Gabriele Rocco, Gianmarco Mengaldo, Daniele De Grazia, Sergey Yakovlev, Jean-Eloi. Lombard, Dirk Ekelschot, Bastien Jordi, Hui Xu, Yumnah Mohamied, Claes Eskilsson, Blake Nelson, Peter Vos, Cristian Biotto, Robert M. Kirby, and Spencer J. Sherwin. Nektar++: An open-source spectral/ hp element framework. *Computer Physics Communications*, 192:205–219, 7 2015.
6. Chris D. Cantwell, Spencer J. Sherwin, Robert M. Kirby, and Paul H. J. Kelly. From h to p efficiently: Selecting the optimal spectral/hpdiscretisation in three dimensions. *Mathematical Modelling of Natural Phenomena*, 6(3):84–96, 1 2011.
7. Chris D. Cantwell, Spencer J. Sherwin, Robert M. Kirby, and Paul H.J. Kelly. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers & Fluids*, 43(1):23–28, 1 2011.
8. Franck Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *The International Journal of High Performance Computing Applications*, 23(3):212–226, 1 2009.
9. Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations*, 1(1), 1 2014.
10. Cheng Chen, Yunfei Du, Ke Zuo, Jianbin Fang, and Canqun Yang. Toward fault-tolerant hybrid programming over large-scale heterogeneous clusters via checkpointing/restart optimization. *The Journal of Supercomputing*, 1 2017.
11. John Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *Computational Science ICCS 2003*, volume 2660 of *Lecture Notes in Computer Science*, pages 3–12, 1 2003.
12. Sheng Di, Mohamed Slim Bouguerra, Leonardo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1181–1190. IEEE, 2014.
13. Catello Di Martino, William Kramer, Zbigniew Kalbarczyk, and Ravishankar Iyer. Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 25–36. IEEE, 2015.
14. Catello Di Martino, William Kramer, Zbigniew Kalbarczyk, Ravishankar Iyer, Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.
15. E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 1 2002.
16. Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1908 of *Lecture Notes in Computer Science*, pages 346–353, 1 2000.
17. Marc Gamell, Daniel S. Katz, Keita Teranishi, Michael A. Heroux, Rob F. Van der Wijngaart, Timothy G. Mattson, and Manish Parashar. Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 346–355. 1 2016.
18. Marc Gamell, Rob F. Van der Wijngaart, Keita Teranishi, and Manish Parashar. Specification of fenix MPI fault tolerance library version 1.0.1. Technical report, 9 2016.
19. Amin Hassani, Anthony Skjellum, and Ron Brightwell. Design and evaluation of FA-MPI, a transactional resilience scheme for non-blocking MPI. In *2014 44th Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks, pages 750–755. IEEE, 2014.

20. Xuejiao Kang, David F. Gleich, Ahmed Sameh, and Ananth Grama. Distributed fault tolerant linear system solvers based on erasure coding. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2478–2485. IEEE, 2017.
21. George Em Karniadakis, Moshe Israeli, and Steven A Orszag. High-order splitting methods for the incompressible navier-stokes equations. *Journal of Computational Physics*, 97(2):414–443, 1 1991.
22. Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, Bronis R de Supinski, Kathryn Mohror, and Howard Pritchard. Evaluating and extending user-level fault tolerance in MPI applications. *The International Journal of High Performance Computing Applications*, 30(3):305–319, 1 2016.
23. Esteban Meneses, Xiang Ni, Gengbin Zheng, Celso L. Mendes, and Laxmikant V. Kalé. Using migratable objects to enhance fault tolerance schemes in supercomputers. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):2061–2074, 2015.
24. David Moxey, Chris D. Cantwell, Robert M. Kirby, and Spencer J. Sherwin. Optimising the performance of the spectral/hp element method with collective linear algebra operations. *Computer Methods in Applied Mechanics and Engineering*, 310:628–645, 1 2016.
25. Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K. Panda. A 1 PB/s file system to checkpoint three million MPI tasks. In *22nd international symposium on High-performance parallel and distributed computing*, pages 143–154, New York, New York, USA, 2013. ACM.
26. Kento Sato, Naoya Maruyama, Kathryn Mohror, Bronis R. de Supinski, Adam Moody, Todd Gamblin, and Satoshi Matsuoka. Design, modeling, and evaluation of a non-blocking checkpointing system. In *SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.
27. Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78:012022, 7 2007.
28. Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 1 2014.
29. Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S. Tanenbaum. Techniques for efficient in-memory checkpointing. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems - HotDep '13*, pages 1–5, 1 2013.
30. Peter E.J. Vos, Spencer J. Sherwin, and Robert M. Kirby. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *Journal of Computational Physics*, 229(13):5161–5181, 1 2010.
31. Gengbin Zheng, Xiang Ni, and Laxmikant V. Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6. IEEE, 2012.
32. Yao Zhu, David F. Gleich, and Ananth Grama. Erasure coding for fault-oblivious linear system solvers. *SIAM Journal on Scientific Computing*, 39(1):C48–C64, 1 2017.