A Fault-Tolerant Gyrokinetic Plasma Application using the Sparse Grid Combination Technique

Md Mohsin Ali, Peter E Strazdins  
Research School of Computer Science  
Australian National University  
Canberra, Australia

Brendan Harding, Markus Hegland, Jay W Larson  
Mathematical Sciences Institute  
Australian National University  
Canberra, Australia

Abstract—Applications performing ultra-large scale simulations via solving PDEs require very large computational systems for their timely solution. Studies have shown the rate of failure grows with the system size and these trends are likely to worsen in future machines as less reliable components are used to reduce the energy cost. Thus, as systems, and the problems solved on them, continue to grow, the ability to survive failures is becoming a critical aspect of algorithm development. The sparse grid combination technique (SGCT) is a cost-effective method for solving time-evolving PDEs, especially for higher-dimensional problems. It can also be easily modified to provide algorithm-based fault tolerance for these problems. In this paper, we show how the SGCT can produce a fault-tolerant version of the GENE gyrokinetic plasma application, which evolves a 5D complex density field over time. We use an alternate component grid combination formula to recover data from lost processes. User Level Failure Mitigation (ULFM) MPI is used to recover the processes, and our implementation is robust over multiple failures and recovery for both process and node failures. An acceptable degree of modification of the application is required. Results using the SGCT on two of the fields’ dimensions show competitive execution times with acceptable error (within 0.1%), compared to the same simulation with a single full resolution grid. The benefits improve when the SGCT is used over three dimensions. Our experiments show that the GENE application can successfully recover from multiple process failures, and applying the SGCT the corresponding number of times minimizes the error for the lost sub-grids. Application recovery overhead via ULFM MPI increases from \( \sim 1.5s \) at 64 cores to \( \sim 5s \) at 2048 cores for a one-off failure. This compares favourably to using GENE’s in-built checkpointing with job restart in conjunction with the classical SGCT on failure, which have overheads four times as large for a single failure, excluding the backtrack overhead. An analysis for a long-running application taking into account checkpoint backtrack times indicates a reduction in overhead of over an order of magnitude.

Keywords—fault tolerance; ULFM; process failure recovery; PDE solver; sparse grid combination; approximation error; gyrokinetic plasma

I. INTRODUCTION

Today’s largest High Performance Computing (HPC) systems consist of thousands of nodes that are capable of concurrently executing up to millions of threads to solve complex problems within a feasible period of time. Significant effort is required to exploit the full performance of these systems. Extracting this performance is essential in different research areas such as climate, environment, physics and energy which can be characterized by complex scientific models.

In the near future, exploiting the full performance of such large systems will be critically hampered by the increased number of component failures [1]. The current widely used standard Message Passing Interface (MPI) [2] library for parallel and distributed programming of HPC systems lacks methods to deal with component failures at run-time. FT-MPI [3] introduced some methods for process failure recovery (see [4] for details), but it was discontinued due to the lack of standardization. Recently, the MPI Forum’s Fault Tolerance Working Group began work on a draft standard for User Level Failure Mitigation (ULFM) to allow the application writers to design recovery methods and control them from the user level [5].

However, there is a lack of practical examples which demonstrate the range of issues encountered during the development of fault-tolerant applications. Moreover, the amount of literature detailing the implementation and performance of the proposed standard is very limited. Some of the work that is available assumes a fail-stop process failure scenario, i.e., a failed process is permanently stopped without recovering and the application continues working with the remaining processes [6]. However, this scenario is not adequate for all applications. For example, some applications do not tolerate a reduction of the size of the MPI communicator and, thus, require recovery of the failed processes in order to finish the remaining computation successfully.

There appears to be an even greater lack of research on how to make existing, complex and widely used parallel applications fault-tolerant. In this paper, we will demonstrate how an existing application, the GENE gyrokinetic plasma simulation code, can be made fault-tolerant using ULFM MPI and a form of algorithm-based fault tolerance obtained via modification of the sparse grid combination technique (SGCT). The recovery from faults includes the restoration of failed processes and MPI communicators on either existing or new (spare) nodes.

The contributions of this paper are to (i) detail how a highly scalable SGCT algorithm can be integrated into an existing and complex application to make it highly fault-tolerant, and (ii) perform a detailed experimental evaluation of this work including scalability, an analysis of result errors in terms of
number of failures, and an analysis of recovery overheads. The latter includes a comparison with checkpointing on a GENE simulation using a non-fault-tolerant SGCT.

The paper is organized as follows. Section II describes the SGCT and how it may be extended to support fault tolerance. Section III describes the existing GENE application. Details of how GENE was adapted to become fault-tolerant using a highly scalable SGCT algorithm are given in Section IV. Experimental results are given in Section V, including an analysis of recovery time for longer computations. Related work is discussed in Section VI, and finally conclusions are given in Section VII.

II. THE SPARSE GRID COMBINATION TECHNIQUE

Numerical solution of Partial Differential Equations (PDEs) are usually performed on a full isotropic grid. With uniform discretization across all its dimensions, the number of grid points increases exponentially with the increase of dimensionality. This behavior makes high-dimensional PDE solvers computationally expensive. In order to address this issue, high-dimensional PDEs may be solved on a sparse grid [7] having substantially fewer grid points than the regular isotropic grid. An example of a sparse grid is shown in Fig. 1, where the sparse grid is represented as the union of regular grids.

A numerical method called the Sparse Grid Combination Technique (SGCT) [8] is employed to approximate the solutions of PDEs on the sparse grid. Instead of solving a PDE on a full isotropic grid, it is solved on several anisotropic grids with substantially fewer grid points, called the sub-grids or component grids, as shown in Fig. 1. Solutions on these sub-grids are then linearly combined to approximate the solution on the sparse grid.

Suppose, for the 2D case, each sub-grid $G_{i,j}$ is assumed to have $(2^i + 1) \times (2^j + 1)$ grid points with a grid spacing of $h_1 = 2^{-i}$ and $h_2 = 2^{-j}$ in the x- and y-directions, respectively, where $i,j \geq 0$. If we consider a square domain, then the grid points of $G_{i,j}$ are $\{(x, y) | x = 0, 1, \ldots, 2^i, y = 0, 1, \ldots, 2^j\}$. In the more general case, the index space for the grids will be some finite $I \subset \mathbb{N}^d$, and the set of grids of interest can be denoted by $G_{i,j} \in I$. If $u_{i,j}$ denotes the approximate solution of a PDE on $G_{i,j}$, the combination solution $u_I^c$ generally takes the form

$$u_I^c = \sum_{i,j \in I} c_{i,j} u_{i,j},$$

where $c_{i,j} \in \mathbb{R}$ are the combination coefficients. Clearly, the accuracy of the combination technique approximation depends on the choice of the index space $I$ of the sub-grids and their respective coefficients. For the 2D case, good choices of the coefficients are $\pm 1$. For instance, in the classical case, we have for level $l$ the set $I = \{(i, j)|i,j \geq 0, l-1 \leq i+j \leq l\}$ and the combination coefficients are $c_{i,j} = 1$ if $i+j = l$ and $c_{i,j} = -1$ if $i+j = l-1$. This provides the following combination formula

$$u_I^c = \sum_{i+j=l} u_{i,j} - \sum_{i+j=l-1} u_{i,j},$$

A fault-tolerant adaptation of the SGCT has been studied in [9]. In this paper, we refer to this adaptation as SGCT+ABFT. It was observed that the solution on even smaller sub-grids can be computed at little extra cost and that this added redundancy allows combinations with alternative coefficients to be computed. When a process failure affects one or more processes involved in the computation of one of the larger sub-grids, the entire sub-grid is discarded. In the event that some sub-grids have been discarded one must modify the combination coefficients such that a reasonable approximation is obtained using solutions computed on the remaining sub-grids. In 2D, this involves finding $c_{i,j}$ for the formula (1) for which $c_{i,j} = 0$ for each $u_{i,j}$ which was not successfully computed. For a small number of failures this is typically done by starting with the formula 2 and subtracting hierarchical surplus approximators of the form $u_{i',j'} - u_{i'-1,j'} - u_{i',j'-1} + u_{i'-1,j'-1}$ such that the undesired $u_{i,j}$ drop out of the formula possibly introducing some of the smaller sub-grids which were also computed. After a combination, all sub-grids may be restarted from the combined solution, including those which had previously failed. An approach for the general computation of combination coefficients is described in [10].

For the 2D fault-tolerant SGCT computations in this paper, two extra layers (or diagonals) of sub-grid solutions $u_{i,j}$ were computed satisfying $i+j = l-2$ and $i+j = l-3$ with levels $l-2$ and $l-3$, respectively. An example of the default combination and an alternative using one of the extra sub-grids is depicted in Fig. 2. For the 3D fault-tolerant SGCT computations, we computed one extra layer (or diagonal) of sub-grids with level $l-3$. 

![Fig. 1. A sparse grid with its components.]

![Fig. 2. A depiction of the 2D SGCT combination coefficients $c_{i,j}$, where +, − and blank represent values of +1, −1 and 0, respectively. On the left is a combination of the form (2). On the right is a modified combination of the form (1) which avoids 3 of the solutions required in the original combination instead making use of one smaller solution. Thus this combination serves as an alternative when a failure affects one or more of the 3 solutions which were avoided.]

500
III. THE GENE GYROKINETIC PLASMA APPLICATION

GENE (Gyrokinetic Electromagnetic Numerical Experiment) [11], [12] is a plasma micro-turbulence application. It contains a multi-dimensional solver of gyrokinetic equations on a field comprising of a fixed grid in a five-dimensional phase space \((x, y, z, v, u)\). The physical space is toroidal (e.g. a tokamak) with a magnetic field whose lines move around the torus. \(x, y, \) and \(z\) represent the spatial dimensions in the radial, perpendicular, and parallel to this field. \(v\) and \(u\) are the velocities in the \(z\) and \(y\) dimensions, respectively. GENE imposes a minimum resolution of 16 in the velocity dimensions.

This field is one of the main outputs of GENE; from it, it computes gyroradius-scale fluctuations and transport coefficients.

The code base [11] is written in Fortran 90 and has hybrid MPI/OpenMP parallelization. High scalability to 10K cores has been reported [12]. Due to the relative smoothness of the solution, the SGCT has yielded good results in producing a relatively accurate solution on important problem sets [13].

Internally, GENE uses a complex precision array \((g_{-1})\) representing the density of each particle (species) of interest in phase space. The number of species \(s\) is typically in the range \(1 \leq s \leq 4\). As well as the dimensions above, \(s\) adds a sixth dimension to the array. GENE is capable of generating this field from initial conditions data or from reading \(g_{-1}\) from a previously stored checkpoint (made by an MPI parallel I/O call).

All processes in a running GENE instance read in the same parameter file \(parameters\), which include things such as the sizes of each grid dimension, number of timesteps, maximum \(\Delta t\) for each timestep etc.

For performing the ‘initial value’ computation, the main subroutine \(rungene()\) initializes the communicator for the simulation, reads in parameters and checkpoint data (when applicable), sets the maximum timestep and calls the \(initial\_value()\) function, which contains the time evolution loop. In each timestep, electromagnetic fields are computed from \(g_{-1}\), and the gyrokinetic equations are applied to produce an update for \(g_{-1}\) for that timestep.

IV. IMPLEMENTATION

In this section, we give an overall description of our 2D/3D SGCT algorithm, with a focus on the parallel process organization imposed and its extensions for fault tolerance. We then describe how it can be integrated into applications with grids of higher dimensions, and finally detail how GENE was modified to accommodate the SGCT.

A. SGCT Algorithm Overview and Process Organization

Referring to (1), each PDE instance whose solution is \(u_{i_j}\) will be run on a distinct set of processes denoted by \(P_{i_j}\) arranged in a logical \(d\)-dimensional grid. The SGCT algorithm consists of first a gather stage, where each process in \(P_i\) sends its portion of \(u_{i_j}\) to each of the corresponding (in terms of physical space) processes in a logical \(d\)-dimensional grid \(P^c\).

For reasons of efficient resource utilization, \(P^c\) is made up of a (normally near-maximal) subset of all processes in \(\cup_{l \in L} P_l\). Each process in \(P^c\) then gathers the \(|I|\) versions of each point of the full grid (using interpolation where necessary), and performs the summation according to (1) to get the sparse grid solution \(u_{i_j}^g\), which can be used as an approximation to the full grid solution. The use of interpolation in turn requires that a ‘halo’ of neighbouring points (in the positive direction, for our implementation) have been filled by a halo exchange operation by each process in each \(P_l\) and is also sent in the gather stage.

In the scatter stage, each process in \(P^c\) sends a down-sample of its portion of \(u_{i_j}^g\) to the corresponding process in \(P_{i_j}\) iteratively, for each \(i_j \in I\).

Further details on the algorithm can be found in [14].

Our SGCT algorithm supports so-called ‘truncated’ combinations [15], where, for the 2D case, each component grid has \((2^l + 2^{l-1} + 1) \times (2^l + 2^{l-1} + 1)\) points, for some \(i_l, j_l \geq (i, j)\). This avoids the problem of minimum dimension size imposed by GENE (section III). Furthermore, it allows us to avoid the use of highly anisotropic grids (e.g. \(G_{1,2}\)) which have been known to contribute least towards the accuracy of the sparse grid solution or cause convergence problems [15], enabling us to concentrate process resources on more accurate sub-grids. In this context, we use a different notion of level to that described in Section II, which describes how much smaller the sub-grids are relative to some full grid \(G_{i_l, j_l}\). In particular, a level \(l \leq \min\{i_l, j_l\}\) in non-fault-tolerant combination in this context consists of sub-grids from the index set

\[
I = \left\{(i, j) : (i_l - l, j_l - l) < (i, j) \quad i_l + j_l - l \leq i + j \leq i_l + j_l + 1 - l \right\}
\]

In terms of load balancing, we allocate the same number \((p \in N)\) of processes on each of the \(P_l\) on the uppermost diagonal in the grid index space (see Fig. 2). The next lower diagonal receives \(\lfloor p/2 \rfloor\) processes. This strategy balances the amount of data points and hence work across each processes, which approximates to a first order to the amount of load for that process. To support the alternate combination technique, four diagonals/planes of grids are used.

Failure of computing nodes or application processes causes the loss of some processes on some grid \(G_{l}\). It is tolerated as follows. Before the SGCT algorithm is applied, the loss of some processes in \(P_l\) is detected using ULFM MPI (see [16] for details). Replacement processes are then created (with the same process grid size as \(P_{l}\)) on the same node when node failure is not happened. Otherwise, they are created on the spare nodes. Then, an alternate combination formula (see Section II) is applied which sets a combination coefficient of \(c_{l} = 0\) for the lost sub-grid solutions \(u_{i_l}\). Note that this formula can be computed on all current processes. In this case, the gather of \(u_{i_l}\) on the replaced \(P_{l}\) and \(P^c\) is not performed.
For the velocity dimensions, the padded elements \( u \) in the non-SGCT dimensions, we use a block factor of \( 1 \) instead. On each process, this corresponds to the \( g_{-1} \) array (a 3D complex density field of GENE).

The set of sub-grid solutions \( \{ u_z \} \), used in formula (1), will have an extra element padded out in the SGCT dimensions. On each process, the storage for \( \{ u_z \} \) will also have room for halo elements. Common elements are copied over from \( g_z \) to \( u_z \) (line 13). For the velocity dimensions, the padded elements in \( u_z \) are initialized to 0; for the spatial dimensions, boundary conditions are applied. Halo exchanges are also performed (line 14). For the \( z \) dimension in GENE, shifts need also to be applied [17], as flux lines traverse the tokamak in a helical fashion; we call into the GENE code itself to perform this operation.

However, process or node failure may happen at any of these stages since the program starts executing. In order to tolerate these failures, the faulty communicator is reconstructed (line 15, details are in [16]). Then the fault-tolerant SGCT is applied using the communicator \( W \), with the combined solution \( u_i^j \) being used to re-initialize the sub-grid solutions \( \{ u_i \} \) (lines 16–17).

The main program uses the SGCT implementation to determine the parallel domain decomposition to be used by GENE. Furthermore, each GENE instance will have different global sizes for the dimensions used in the combination (e.g., for a 3D SGCT, these might be \( N_x, N_y \) and \( N_u \)). The main program also creates a different directory for each GENE instance with a customized parameters file containing the above values for this instance. Only a limited number of modifications to the GENE source were required to apply the SGCT. A standard tool is utilized for interoperability between C and Fortran. We used the intrinsic module ISO_C_BINDING and the language-binding-spec attribute BIND for this interoperability. A C wrapper of C++ code is used to hide the C++ code from the inter-operation. A list of modifications that were made on GENE are as follows.

- The runGENE() function at line 12 of Algorithm 1 replaces the main subroutine of GENE. It calls into some top-level GENE subroutines including rungene(...). One extra parameter is added to rungene() to store the sub-grid communicator \( C_z \) to determine the process rank and communicator size with that parameter. The other subroutines are not modified.
- In the initial_value() subroutine, called by rungene(), a C++ function call c_get_g1() is made with Fortran’s INTERFACE block at the end of the time evolution loop. This passes \( g_{-1} \) (and other associated data) from GENE to the SGCT to initialize its sub-grid solution \( u_z \).
- \( u_z \) passed to initial_value() via rungene(), is used to initialize the \( g_{-1} \) field before entering the time-loop. This is required for repeated combinations over...

Algorithm 1: Main function of the modified application. Operations are assumed to be applied in parallel over all processes in the relevant communicator.

B. Integration of the SGCT Algorithm into Higher-Dimensional Grids

The field of GENE, regarded as a field of real numbers, can be thought of an array with a dimensionality of

\[
D = (2, N_x, N_y, N_z, N_u, N_s, s)
\]

The first element in \( D \) arises from the field being complex (note that the SGCT uses only additions and multiplications with real coefficients).

Our implementation performs the SGCT across such a field as follows. The dimensions for the SGCT must be a contiguous sub-vector of \( D \). Any remaining lower dimensions of \( D \) can be dealt with an extension to the algorithm to operate on blocks of \( b \geq 1 \) real elements. Any remaining higher dimensions can be dealt with by applying the SGCT iteratively over these dimensions. If the above proves restrictive, the grid can be transposed to get the desired ordering in \( D \).

For example, to perform a 2D combination on the \( N_y \) and \( N_u \) dimensions, we use a block factor of \( b = 2N_y N_u N_z \) and, if \( s = 2 \), iterate the SGCT over each of two array slices in the \( s \) dimension.

Our SGCT algorithm supports parallelization over arbitrary process grids in the SGCT dimensions. To support parallelization to a total factor of \( p \in \mathbb{N} \) in the non-SGCT dimensions, \( p \) independent SGCT computations can be performed in parallel. The implementation of this requires the careful construction of MPI communicators for each process sub-grid; the details of this are beyond the scope of this paper.

C. Modifications to GENE for the SGCT

Algorithm 1 describes the main program, written in C++, the same language as the implementation of our SGCT algorithm, calling into the Fortran code of GENE.

A global communicator \( W \) is used to create a set of sub-grid communicators \( \{ C_i \} \) to simultaneously run several (each \( i \in I \)) GENE instances (line 11), with GENE itself being called for the specified number of timesteps on line 12. The first time this is called, \( u_z \) is null, and runGENE() uses initial condition data to initialize \( g_z \); afterwards, it uses \( u_z \) instead.
time. For a single combination, this is not used (passed as null). GENE initializes $g_1$ by itself.

V. EXPERIMENTAL RESULTS

In this section, we describe the experimental setup used for benchmarking, analysis of execution performance, and memory consumption of both the SGCT and equivalent full grid computations. Following this, we discuss the approximation error of solutions computed with SGCT+ABFT, and compare recovery overheads of the SGCT+ABFT using ULFM with that of the SGCT implementation using Checkpoint/Rerstart [18]. Finally, we analyze the repeated failure recovery performance of ULFM.

A. Experimental Setup

All experiments were conducted on the Raijin cluster located at the Australian National University. It has a total of 57,472 cores distributed across 3,592 compute nodes each consisting of dual 8-core Intel Xeon (Sandy Bridge 2.6 GHz) processors (i.e., 16 cores) with Infiniband FDR interconnect, a total of 160 TB bytes (approx.) of main memory, and 10 PBytes (approx.) of usable fast filesystem [19].

We used git revision icldistcomp-ulfm-46b781a8f170 of ULFM under the development branch 1.7ft of Open MPI for implementation. The parameters for the collective communications for mpifortuned, ftbasic,basic,self. The value of the MCA parameter coll_ftbasic_method was set to 1 to choose the ‘Two-Phase Commit’ as an agreement algorithm for failure recovery. The ‘Log Two-Phase Commit’ option was more scalable than this, but could not be used now due to its instability. All the source code (including ULFM MPI) were compiled with GNU-4.6.4 compilers with optimization flag 

Experiments were conducted based on a problem from the GENE testsuite(testsuite/big/parameters_6). One was called 2d_big_6, with a full grid size $(N_v, N_u) = (2^8, 2^8)$ for the 2D SGCT, $N_x = 64, N_y = 4, N_z = 16$, and the level of the SGCT was $l = 5$. For a 2D fault-tolerant SGCT for $l = 5$ with power of two sizes for the component’s process grids, it turns out that the total number of processes was also a power of two, permitting a head-to-head comparison with the full grid results. The other was called 3d_big_6, with a full grid size $(N_z, N_v, N_u) = (2^8, 2^8, 2^8)$ for the 3D SGCT, $N_x = 32, N_y = 4$ and the level of the SGCT was $l = 4$. For both cases we set the number of species $x$ to be 1, the timesteps to 100, the maximum $\Delta t = 10^{-3}$, and the grid type for the $N_u$ dimension to be ‘equidist’.

Faults were injected into the application by aborting single or multiple MPI processes at a time (except process 0 as it was used for the controlling purposes) by the system call $\text{kill}$(getpid(), SIGKILL) at some point before the combination of the component grid solutions. MPI processes were also failed repeatedly (not at a single time) to examine the repeated failure recovery performance of the application.

B. Execution Performance

Fig. 3a gives execution time for the 2D SGCT, 3D SGCT, and the equivalent full grid computation. It is observed that the 2D SGCT is $\approx 2 \times$ faster than the equivalent full grid computation. The difference in this speed reflects the reduced amount of work enabled by the SGCT compared to the full grid, which for $l = 5$ on the 2D case is approximately one half. Execution time of the 3D SGCT shows a clear advantage in terms of reduced amount of work, which for $l = 4$ in this case is approximately one quarter. This causes 3D SGCT to be $\approx 4 \times$ faster than the equivalent full grid computation.

Fig. 3a also provides evidence that our SGCT implementation supports parallelization over non-SGCT dimensions. For the 2D case, we choose parallelization $p = 2$ on non-SGCT dimension $N_z$, where $N_v$ and $N_u$ holds process sub-grid for the SGCT.
component grids, and the 3D big_6 error component grid, inputs are used for the 2D and 3D SGCT computations, respectively, which are different.

\[ u \approx t \text{timesteps with } g_1 \]

\[ T \text{N} := T \text{timesteps each with checkpoint write, } T + T \text{and } T \text{are the times } \]

\[ \text{field of the full grid solution and } (T \text{and } T \text{is the system } \] for a single occurrence of failure. This excludes the \( T \text{ and } T \text{ are the system time of running SGCT and relative } \]

\[ T = (t_1 + t_2) - (t_3 + T \text{RD}), \quad (3) \]

where \( t_1 \) and \( t_2 \) are the system time of running SGCT with CR for 50 timesteps each with checkpoint write at the end (no checkpoint read) and with initializing the application from previously written checkpoint (no checkpoint write at the end), respectively. \( t_3 \) is the system time of running SGCT with CR for 100 timesteps with checkpoint write at the end (no checkpoint read).

E. Failure Recovery Overheads

Fig. 6a shows the component timings that are used for estimating the recovery overheads of the two approaches. First, the implementation of SGCT using ULFM, which uses the algorithm-based recovery (AB) to recover from failures. Second, the implementation of SGCT using Checkpoint/Restart, which uses a Checkpoint/Restart based recovery (CR) to recover from failures. The components are generated using 2d_big_6 input, and will be used for measuring the overheads of both the shorter and longer computations.

The notations in this figure are as follows.

- \( T_\text{RP} := T_\text{RP}(N) \) and \( T_\text{RN} := T_\text{RN}(N) \) are the times taken to reconstruct communicators on \( N \) nodes using ULFM for a single occurrence of a fault to recover processes and nodes failure, respectively.
- \( T_\text{WR} := T_\text{WR}(N) \) and \( T_\text{RD} := T_\text{RD}(N) \) are the times required to write a checkpoint on \( N \) nodes and read a checkpoint from \( N \) nodes, respectively.
- \( T_\text{RM} \) is a single MPI launch time (when restarts from checkpoint after failure), which can be calculated by

\[ T_\text{RM} = (t_1 + t_2) - (t_3 + T_\text{RD}), \quad (3) \]

Based on the above notations, it is possible to estimate the recovery overhead for the shorter computations. AB overheads of a one-off processes and nodes failure are \( T_\text{RP} \) and \( T_\text{RN}, \) respectively. With CR, the overhead is the sum of \( T_\text{WR}, T_\text{RM}, \) and \( T_\text{RD} \) for a single occurrence of failure. This excludes the overhead of backtrack time. It is observed that the one-off failure recovery overhead of the CR based technique is \( \approx 4 \times \) larger than that of the algorithm-based technique (both for processes and nodes failure). We expect this gap to increase in future mature ULFM MPI releases.

Using results gathered so far, we will estimate the overhead of our implementation for longer computations and different frequencies of faults. We will compare with a Checkpoint/Restart based recovery (CR) taking into account the overhead generated by having to backtrack to the previous checkpoint when failures occur. It is assumed that the oc-
currence of faults is independent and identically distributed on each compute node. Further, it is assumed that fault
are exponentially distributed and therefore the failure rate is
constant. The other variables we use are as follows.

- \( N \) and \( C \) are the number of nodes and combinations used
  in the computation, respectively.
- \( T_{fn} \) and \( T_{fn}/N \) are the mean time between failures
  (MTBF) on each node and across \( N \) nodes, respectively.
- \( T_{AB} := T_{AB}(N) \) is the total run time of the SGCT
  implementation on \( N \) nodes with AB.

Experimental results presented in Fig. 3a and Fig. 6a allow us to estimate \( T_{AB} \) and \( T_{RP} \), respectively, for different \( N \).
Note that in order to have a reasonable approximation error, it is sensible to choose \( C \) such that at most 1 fault occurs on
average between combinations, that is, \( C \geq T_{AB}/(T_{fn}/N) = N \cdot T_{AB}/T_{fn} \). The only overhead is from the recovery of
processes and reconstruction of communicators using ULFM
when a failure occurs. As the expected number of failures is
equal to the number of combinations, one has the additional
overhead \( C \cdot T_{RP} \). Note that recovery in the SGCT algorithm
only occurs prior to each combination, GENe instances not
affected by the failures continue to run independently up to the
combination at which the status of processes within the
global communicator is checked. Thus, \( C \cdot T_{RP} \) is actually an
upper bound on the overhead for process recovery throughout
the computation. Thus, the expected overhead is bounded
above by

\[
C \cdot T_{RP} = \frac{N \cdot T_{AB}}{T_{fn}} T_{RP}. 
\]

One sees that this is inversely proportional to the MTBF per
node. Note that the occurrence of faults obviously affects the
error of the SGCT. The estimation of this, however, is more
involved.

We will compare the algorithm-based recovery overheads
with the typical overhead of the Checkpoint/Restart applied to
the SGCT computation. Here we define the additional values
of interest.

- \( T_{CR} := T_{CR}(N) \) is the total run time of the SGCT
  computation on \( N \) nodes using Checkpoint/Restart for
recovery from faults.

- \( T_{CR}/(T_{fn}/N) = N \cdot T_{CR}/T_{fn} \) is the expected number
  of failures throughout the computation.

- \( T_{DC} = \sqrt{2T_{WR} \cdot T_{fn}/N} \) is the optimal time between
  checkpoints.

- \( T_{CR}/T_{DC} \) is the total number of checkpoints throughout
  the computation.

- \( T_{R} := T_{R}(N) \) is the total recovery time after a fault
  including restarting MPI and reading a checkpoint on \( N 
  \)
  nodes. This is equivalent to \( T_{RM} + T_{RD} \).

- \( T_{B} = T_{DC}/2 \) is the average backtrace time when a
  fault occurs, that is, the typical time between the last
  checkpoint and a failure for which recomputations must
  be done.

Experimental results summarized in Fig. 3a and Fig. 6a allow us
to estimate \( T_{CR}, T_{WR} \) and \( T_{R} \) for some different values
of \( N \). The total overhead for Checkpoint/Restart consists of
two components. The first is the writing of checkpoints, which
throughout the computation is

\[
\frac{T_{CR} T_{WR}}{T_{OC}} = T_{CR} \frac{\sqrt{N \cdot T_{WR}}}{\sqrt{2T_{fn}}}. 
\]

Additionally, when each failure occurs, MPI must be restarted,
a checkpoint read, and recomputation done up to the point at
which the failure occurred. This overhead is the restart time
plus the typical recomputation time multiplied by the expected
number of faults, that is,

\[
N \frac{T_{CR}}{T_{fn}} (T_{R} + T_{B}) = T_{CR} \left( \frac{N T_{R}}{T_{fn}} + \frac{\sqrt{N \cdot T_{WR}}}{\sqrt{2T_{fn}}} \right). 
\]

Adding the two together the total Checkpoint/Restart overhead is

\[
\frac{N \cdot T_{CR}}{T_{fn}} T_{R} + T_{CR} \frac{\sqrt{2N \cdot T_{WR}}}{\sqrt{T_{fn}}}. 
\]

Note that this overhead obviously extends the run time of
the application thus exposing it to more faults and that the
same applies for the overhead with algorithm-based recovery.
One may, however, divide the application run-time out of both
overheads, and instead compare the overheads relative to the application run-times. We are particularly interested in how the two compare as the time between failures varies. The change in relative overheads with respect to $T_{RP}$ is plotted in Fig. 6b using representative values for the remaining variables obtained from the previous figures. It is observed that the overhead of the algorithm-based approach is significantly less than the equivalent computation done using Checkpoint/Restart (compare CR $N = 112$ and $N = 14$ with AB $N = 128$ and $N = 16$ respectively).

VI. RELATED WORK

A technique for replacing only a single failed process on the communicator and matrix data repair for a QR-Factorization problem is proposed in [21]. Process failure is handled by the ULFM standard, and data repair is accomplished by using a reduction operation on a checksum and remaining data values.

Algorithm-Based Fault Tolerance (ABFT) techniques for creating robust PDE solvers based on the modified sparse grid combination technique are proposed in [22], [9]. The proposed solver can accommodate the loss of single or multiple component grids. Grid losses are tolerated by either deriving new combination coefficients to excise a faulty component grid solution or approximating a faulty component grid solution by projecting the solution from a finer component grid. This work, however, was implemented using simulated, rather than genuine, process failures.

A fault-tolerant implementation of a multi-level Monte Carlo simulation that avoids checkpointing or recomputation of samples was proposed in [23]. It used the ULFM standard to recover the communicator by sacrificing its original size and employing a periodic reduction strategy with all the unaffected samples to generate final result.

Early work in the parallelization of the SGCT for the 3D Navier-Stokes system was reported in [24]. However, the algorithm assumed each node had multiple sub-grids and concentrated on load balancing aspects, whereas our algorithm is designed for the large-scale case.

An SGCT-based, fault-tolerant 2D advection equation solver capable of surviving single or multiple real process failures was proposed in [16]. This paper described in detail how ULFM MPI could be used for process recovery, and our work uses the same method. However, the study was limited to a simple benchmark written for this application for process failure only, while this paper shows how the techniques can be applied to a complex real-world application capable of handling both process and node failure.

The first application of the SGCT to GENE was reported in [13]. Under this scheme, component grid instances of GENE were run, with their respective outputs written to data files that were subsequently combined to compute the SGCT solution. However, this effort has not investigated the fault-tolerant possibilities of the SGCT for this application, nor has it implemented any alternative fault-tolerant techniques.

VII. CONCLUSIONS

In this paper, we have presented an overview of a general parallel SGCT combination algorithm and its associated load balancing strategy. The algorithm can be applied over several of the dimensions of a multi-dimensional field of a PDE time-evolving application. It also easily supports parallelization in the non-SGCT dimensions. Thus, it is capable of supporting extremely large-scale applications. We have shown how it can be integrated into an existing, complex real-world application (GENE) with minimal changes to the source code. For relatively large fields, the overhead of the combination was of the
order of 1s, easily acceptable compared with the current and near future MTBFs.

Using a recent release to the ULFM MPI, our resulting implementation is robust to surviving multiple failures and recoveries, tested up to 2048 cores. We have shown that the 2D, and especially the 3D, SGCT have significant computational efficiencies compared with the traditional full grid simulation while having acceptable losses in accuracy. We did find however that the ‘smoothness’ of the dimensions of the field chosen for the SGCT were important in this respect. In the case of multiple failures, multiple applications of the SGCT can be used to reduce the error.

We have shown that the SGCT in conjunction with ULFM MPI can be used to recover a complex application from both process and node failures. Compared with the built-in checkpoint infrastructure in our application and job restart from the checkpoint, our approach has ≈ 1/4 of the overhead for a one-off failure excluding the overhead of backtrace time. An analysis for a long-running application taking this into account shows that our technique has an overhead between one and two orders of magnitude less. We expect this difference to increase as the relatively recent ULFM MPI matures.

We expect that our ABFT technique will show significant advantages on current and future platforms beyond what we have access to for this paper. This includes both much larger systems where checkpointing overheads become prohibitive, and systems whose components are less reliable than supercomputer nodes, e.g., very cheap processors operated at minimal voltage in order to save power.

Future work includes applying our methodology to other PDE simulations suitable to the sparse grid technique.

ACKNOWLEDGMENTS

The authors thank Christoph Kowitz for providing a distribution of GENE and for advice on the application. This research was supported under the Australian Research Council’s Linkage Projects funding scheme (project number LP110200410). We are grateful to Fujitsu Laboratories of Europe for providing funding as the collaborative partner in this project and the use of the OPL cluster. We thank the National Computational Infrastructure (NCI), supported by the Australian Government, for the use of the Raijin cluster.

REFERENCES