Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 130 – 139

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Fault-Tolerant Grid-Based Solvers: Combining Concepts from Sparse Grids and MapReduce

J. W. Larson[a], M. Hegland[a], B. Harding[a], S. Roberts[a], L. Stals[a], A. P. Rendell[b], P. Strazdins[b], M. M. Ali[b], C. Kowitz[d], R. Nobes[c], J. Southern[c], N. Wilson[c], M. Li[c], Y. Oishi[c]

[a]*Mathematical Sciences Institute, The Australian National University, Canberra, ACT 0200, AUSTRALIA*
[b]*Research School of Computer Science, The Australian National University, Canberra, ACT 0200, AUSTRALIA*
[c]*Fujitsu Laboratories of Europe, Hayes Park Central, Hayes End Road, Hayes, Middlesex UB4 8FE, UK*
[d]*Institut für Informatik, Technische Universität München, Boltzmannstr. 3, 85748 Garching bei München, GERMANY*

Abstract

A key issue confronting petascale and exascale computing is the growth in probability of soft and hard faults with increasing system size. A promising approach to this problem is the use of algorithms that are inherently fault tolerant. We introduce such an algorithm for the solution of partial differential equations, based on the sparse grid approach. Here, the solution of multiple component grids are efficiently combined to achieve a solution on a full grid. The technique also lends itself to a (modified) MapReduce framework on a cluster of processors, with the map stage corresponding to allocating each component grid for solution over a subset of the processors, and the reduce stage corresponding to their combination. We describe how the sparse grid combination method can be modified to robustly solve partial differential equations in the presence of faults. This is based on a modified combination formula that can accommodate the loss of one or two component grids. We also discuss accuracy issues associated with this formula. We give details of a prototype implementation within a MapReduce framework using the dynamic process features and asynchronous message passing facilities of MPI. Results on a two-dimensional advection problem show that the errors after the loss of one or two sub-grids are within a factor of 3 of the sparse grid solution in the presence of no faults. They also indicate that the sparse grid technique with four times the resolution has approximately the same error as a full grid, while requiring (for a sufficiently high resolution) much lower computation and memory requirements. We finally outline a MapReduce variant capable of responding to faults in ways other than re-scheduling of failed tasks. We discuss the likely software requirements for such a flexible MapReduce framework, the requirements it will impose on users' legacy codes, and the system's runtime behavior.

Keywords:
Parallel computing, partial differential equations, fault-tolerance, sparse grids, MapReduce

1. Introduction

The numerical solution of large-scale partial differential equations (PDEs) faces two fundamental scalability challenges. Firstly, as the spatial dimensionality d of the PDE increases, the number of unknowns on a full

*J. Walter Larson Tel.: +61-2-6125-7891 ; fax: +61-2-6125-4984.
E-mail address: jay.larson@anu.edu.au.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.176

isotropic grid is $O(n^d)$, where n is the grid resolution in each dimension. This situation—called the *curse of dimensionality*—renders a high grid resolution on a full grid intractable as d increases. Secondly, even when the number of unknowns is amenable to solution on state-of-the-art supercomputers, the likelihood of encountering faults during a computation increases as the system size scales to petascale and beyond; the probability of errors during the computation becomes proportional to both the number of hardware components used and the duration of the computation. These errors include both *hard faults*, where components fail according to a given probability per time unit, and *soft errors*, which arise from transient faults in the computation [1].

Methods such as checkpoint-restart can be employed to deal with hard errors [2]. While a general technique, checkpointing may be problematic at exascale due to the high energy costs resulting from its inherent memory and I/O intensiveness. Algorithm-based fault-tolerance (ABFT) techniques can provide a solution for both hard and soft faults. While these are special-purpose, depending on particular properties of the computation needing to be solved, they often provide a low-cost solution [3, 4].

MapReduce methods [5] have recently shown strong promise in the parallel solution of many large-scale problems [6]. Provided the data is well-aligned with the constituent Map and Reduce tasks, these can scale very well. Furthermore, the paradigm offers inherent fault-tolerance as the Map or Reduce tasks that fail can be restarted on another node in the system.

In this paper, we present an ABFT solution to PDEs that can both reduce the curse of dimensionality in high-dimensional problems and can be made robust in the presence of both hard and soft faults. It is based on the sparse grid combination technique [7, 8], where the problem is solved on a number of anisotropic component grids that have high resolution in some—but not all—dimensions, which are then combined to form a sparse grid approximation to the full high-dimensional grid. This technique has a complexity of only $O(n \log(n)^{d-1})$ and has been shown to parallelize efficiently [9].

The technique can also be formulated using a MapReduce paradigm, where, in the context of implementation on a multicore cluster, the Map tasks corresponds to the distribution of component grids for solution across subsets of cores in the system, and the Reduce tasks being their subsequent combination over the sparse grid.

While it thus offers the same scope for fault-tolerance as in general MapReduce formulations, the sparse grid combination technique also offers fault-tolerance through the redundancy of the multiple component grids. Thus, if any part or all of the solution on a component grid is missing, instead of restarting the corresponding task, we can instead simply use information from the other grids to either reconstruct the missing component grid, or simply use a modified combination of the remaining component grids to produce the overall solution, at the possible cost of some loss in accuracy. This leads us to a modified MapReduce programming pattern under which the Reduce task is dynamically formulated upon the degree of success (after a given time) of the spawned Map tasks.

The remainder of this paper is organized as follows. Related work is summarized in Section 2. An overview of MapReduce is presented in Section 3. Section 4 describes the sparse grid technique and in particular the combination technique with and without faults. A prototype implementation using the two-dimensional advection equation is described in Section 5, including results of the errors in the sparse grid technique, again with and without faults. Section 6 describes how such an implementation may be generalized in a modified map-reduce context, with conclusions being given in Section 7.

2. Related Work

The development of fault-tolerant techniques (FTT) has been an active research area in High Performance Computing (HPC) for nearly thirty years, following the first ABFT work of Huang and Abraham [3]. FTT can be broadly categorized as either technological strategies or application-specific ABFT. Technological strategies include checkpointing, replication, and fault-tolerant parallel infrastructure such as fault-tolerant extensions to MPI. ABFT includes strategies such as task pools with reassignment of dropped tasks, such as in MapReduce, or a variety of other techniques summarized later in this section. Our work is in the implementation of ABFT at the application level.

Checkpointing entails periodically saving the state of a computation such that the computation can be restarted from that point in the event of a failure. The computation state is usually saved on a parallel file system and is sensitive to parallel I/O performance. Parallel I/O performance enhancements, however, usually cannot compete

with corresponding improvements in processor speeds and memory bandwidth and latency. Thus, the parallel I/O remains a bottleneck that makes checkpointing a time-consuming process. This problem can be partially ameliorated by reducing the volume of state data stored to disk. *Diskless checkpointing* [10] obviates the need for going to disk by saving checkpoint state data in memory on compute nodes, restarting the program from that state in the event of a failure. Specifically, additional nodes are used to store a checksum of the memory checkpoints so that the memory checkpoints of the faulty compute nodes can be reconstructed from this checksum. Although diskless checkpointing offers superior performance comapred to disk-based checkpointing, it requires the use of additional nodes to store only checksums of memory checkpoints and this impost grows in proportion to the size of the global node pool used by the application. Diskless checkpointing's advantages must be considered in light of the communcation and computation costs associated with creating a diskless checkpoint[1]. A detailed survey of the various checkpointing protocols that are currently in use is given in [11].

Replication involves storing the same data on multiple processes in a distributed system [12]. In the event of process failure, these replicated data can be used to reconstruct the work of the failed process. Usually, this can be done by spawning two processes that perform exactly the same computation. If one of these processes fail, the results from the other one can be used to replace failed process. The main benefits of replication of data can be classified as performance enhancement, reliability enhancement, better data locality, shared workload, increased availability, and increased fault tolerance. The constraints are similarly classified as how to keep data consistent, where to place replicas and how to propagate updates, and how these choices impact scalability. The drawbacks to this approach is increased resource requirements and performance degradation stemming from replication communications costs.

Task pools with reassignment is a strategy in which a "master" node coordinates the assignment of tasks executed by a pool of "worker" nodes, reassigning tasks in the event of worker node failure. The best-known example of this strategy is MapReduce and its variants, which will be discussed in Section 3. This approach is robust with respect to worker node failure but remains vulnerable to master node failure.

The prevalance of MPI in HPC applications has driven considerable work in development of fault-tolerant MPI implementations. Although MPI and MPI-2 functions can provide error codes, these standards do not provide strategies for continued execution after encountering such errors. The MPI-3 standards process included a fault-tolerant techniques working group, but there is still no fault-tolerance specification in the MPI-3 standard [13]. Thus, researchers in FTT for MPI-based codes must work with an extension to the MPI standard, such as the semantics described in [14], or with an MPI implementation that provides fault-tolerance through its own checkpointing and/or replication mechanisms, such as MPICH-VCL [15] or openMPI [16].

ABFT techniques at the application level most frequently rely on some underlying checkpointing or replication technology. A notable example is Ltaief et al. [17], who used a fault-tolerant MPI to implement a replication-based robust scheme for heat transfer problems. Algorithms that are naturally fault-tolerant and require only minimal parallel environment infrastructure support are less common. Srinivasan and Chandra [18] describe a parareal—having a parallel domain decomposition in time as well as space—algorithm for molecular dynamics simulation of nanomaterials that is inherently fault-tolerant in its ability to cope with lost portions of a solution by re-computing them from successfully completed portions. The work we describe here pursues ABFT in a similar vein. We propose algorithms in which a set of *component* computations are *combined* to calculate a solution. Each component resides on a distinct pool of processors or *cohort*. If one or more processors in a given cohort fail, the corresponding component calculation is dropped and not used in the combined solution, with the consequence of reduced—but still acceptable—accuracy.

The Open Petascale Libraries project [19] is a global initiative established with the aim of assisting the development of advanced applications software on the emerging generation of supercomputers by the provision of libraries that extract high performance from and can be easily adapted to the underlying computer architecture. The work presented here fits within this framework since it (a) increases application scalability by offering a solution to the curse of dimensionality; and (b) provides an alternative solution to the problem of node failure — which becomes more likely at scale — that does not suffer from the limitations — also more apparent at scale — of checkpointing, replication or the use of task pools with reassignment.

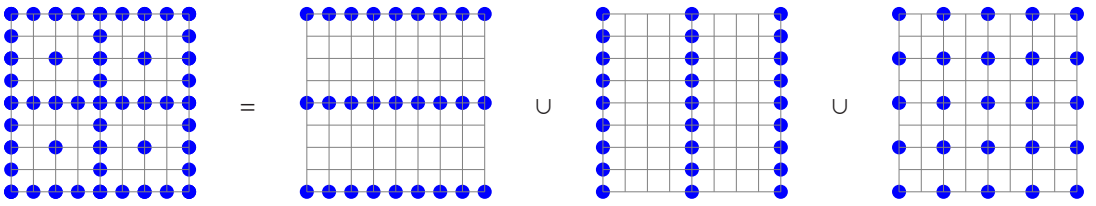

Fig. 1. A sparse grid with its components

3. MapReduce

MapReduce [5, 6] is a functional programming pattern popularized by Google, who have used it to compute page rank and inverted indices of the Internet. In broadest terms, the MapReduce programming model is the composition of one Map() and one Reduce()–sometimes called Fold()–function, with Reduce() operating on the outputs of Map(). The discussion of sparse grid combination method in Section 4 and the 2D advection prototyped described in Section 5 employ the MapReduce pattern in this purely functional sense.

Google's MapReduce programming model defines the Map() and Reduce() functions as follows: the Map() transforms input data into a list of output `<key,value>` pairs; the Reduce() phase transforms all `values` associated with a given `key` into a final result for that `key`. Map outputs are routed to Reduce inputs by an intermediate sorting facility provided by the MapReduce framework. Data input into the framework, intermediate `<key,value>` pairs, and the framework's output are stored on a proprietary distributed file system (DFS). The `<key,value>` pairs are sorted, and grouped by `key` for subsequent routing to Reduce nodes. Fault-tolerance is implemented using replication and task pools with reassignment. The Google MapReduce implementation is proprietary. Yahoo! have developed the open-source Hadoop implementation [20] that imitates the functionality and fault-tolerance of Google MapReduce. Hadoop is widely used in large-scale data analysis in the commercial and research communities. The rapid uptake of MapReduce has spawned other implementations, including one utilizing Python's Pool class [21], and the message-passing-parallel MapReduce-MPI [22]. A review discussion of MapReduce implementations, advantages and disadvantages of the MapReduce programming model, and its variants and extensions is given by [23].

4. Sparse Grids and a Fault-tolerant Combination Technique

Sparse grids [24, 8] are computational grids that contain substantially fewer points than the usual regular isotropic grids. They are particularly suited to higher-dimensional problems as they are less affected by the curse of dimensionality. An example of a sparse grid is seen in Figure 1. The figure also shows that the sparse grid is represented as the union of regular grids. In the following discussion we only consider 2-dimensional grids. Any regular grid is assumed to have a grid spacing of 2^{-i} in the x direction and 2^{-j} in the y direction. Following [25], we call any union of regular grids a sparse grid.

The sparse grid combination technique [7, 8] approximates sparse grid solutions to PDEs by linear combinations of solutions on regular grids. Here we will consider a square domain in which the grid points of the regular grids $G_{i,j}$ are $\{(\frac{x}{2^i}, \frac{y}{2^j}) | x = 0, 1, \ldots, 2^i, y = 0, 1, \ldots, 2^j\}$. For the example shown in Figure 1, five subgrids $G_{i,j}$ with $2^i + 1$ by $2^j + 1$ grid points are required. These include in addition to the three constitutive grids $G_{3,1}$, $G_{1,3}$ and $G_{2,2}$ the two intersection grids $G_{2,1}$ and $G_{1,2}$. The sparse grid itself may be defined using only the first three

$$G_{\mathrm{SG}} = G_{3,1} \cup G_{1,3} \cup G_{2,2}$$

but for the combination technique approximation one requires the solutions $u_{i,j}$ on all the five grids $G_{i,j}$ as the approximation takes the form

$$u_{\mathrm{CT}} = u_{3,1} + u_{1,3} + u_{2,2} - u_{2,1} - u_{1,2}.$$

The combination technique is a very general approach and has been used to obtain sparse grid approximations to problems for which solutions are available on regular grids. An example is the determination of eigenvalues,

see [26]. Here the GENE code [27] was used to obtain approximations of eigenvalues on regular grids. It was demonstrated that combinations of these eigenvalues provided improved approximations to the eigenvalues of the underlying system of PDEs. The combination technique could even adapt the strongly anisotropic grid sizes used in GENE by using the dimension adaptivity explained in [25]. The sparse grid combination technique displays parallelism at two levels: Firstly, all the problems on the regular subgrids can be computed independently in parallel. Secondly, each regular grid solution can be determined in parallel using domain partitioning. A first simple discussion of this is provided in [9].

In the most general case the combination technique takes the form

$$u = \sum_{(i,j)\in I} c_{i,j}\, u_{i,j}, \tag{1}$$

where I is a downset (or lower set), i.e. it satisfies the property that if $(i, j) \in I$, $0 \le i' \le i$ and $0 \le j' \le j$ then $(i', j') \in I$. Clearly, the accuracy of the combination technique approximation depends on the choice of the regular grids $G_{n,m}$ and the combination coefficients. Effective choices of the combination coefficients are discussed in [28, 25, 29]. It turns out that in 2-dimensions good choices of the coefficients are ± 1. For example, in the classical case we have for level l the set $I = \{(i, j)|i, j \ge 0,\ i + j \le l\}$ and the combination coefficients are $c_{i,j} = 1$ if $i + j = l$, $c_{i,j} = -1$ if $i + j = l - 1$, and $c_{i,j} = 0$ otherwise. This gives rise to the combination formula

$$u = \sum_{i+j=l} u_{i,j} - \sum_{i+j=l-1} u_{i,j}\,. \tag{2}$$

This situation is depicted in Figure 2(a) where each square corresponds to a solution $u_{i,j}$ and the non-zero coefficients are 1 or -1 denoted by a plus or minus sign respectively.

The meaning of equation (1) is that any function value $u(x) = \sum_{(i,j)\in I} c_{i,j} u_{i,j}(x)$. This would require accessing all the partial solutions (and there can be thousands in higher dimensions) for every function evaluation. More effectively, one stores the values of u on the sparse grid $G = \bigcup_{i,j} G_{i,j}$ and uses sparse grid interpolation to recover any other values. The determination of the values on the sparse grid is then the reduce step which will be discussed later. The reduce step is closely related to the hierarchisation of sparse grids which is discussed in [30].

The sparse grid combination is mostly used to solve high-dimensional problems. But the pattern of equation (1) make it also attractive for lower-dimensional cases. Indeed, it has been used for preconditioning [31] and here we obtain a fault-tolerant method using the combination technique. Equation (1) is of a MapReduce form, which is a widely applied pattern in parallel processing and the consequences of this is further discussed in Sections 3 and 5. But this nice pattern comes at a price: First, the sparse grid approximation itself provides only an approximation of a regular grid solution at the same level and furthermore the combination technique only is an approximant of the sparse grid approximation. Thus in order to obtain the same accuracy with the sparse grid combination technique as with a regular grid one requires a higher level. This can be seen in Figure 3. This error, however, is not further discussed in the following – more information can be found in the review paper [8] if required.

Below we describe how the combination technique can be modified to implement ABFT.

Given I the problem is to determine a combination as in Equation (1) when one of the solutions $u_{i',j'}$ is missing for some $(i', j') \in I$. We illustrate this in the case of the classical combination technique. We consider two different approaches, one where we modify the set I and the second where we recover $u_{i',j'}$ from other solutions. In the first approach we have two situations in 2D. Either $i' + j' = l$ in which case we define $I' = I\backslash\{(i', j')\}$, or $i' + j' = l - 1$ in which case we may define $I' = I\backslash\{(i' + 1, j')\}$ or $I' = I\backslash\{(i', j' + 1)\}$. Here the resulting I' are downsets and the coefficients $c_{i,j}$ for $(i, j) \in I'$ are assumed to be known [25]. The resulting combination formulas are illustrated in Figure 2 (b) and (c) for a classical combination of level 5 where the failed solution is $u_{1,4}$ and $u_{2,2}$ respectively.

In the second approach we only consider the case $i' + j' = l - 1$. Here we recover $u_{i',j'}$ by sampling either $u_{i'+1,j'}$ or $u_{i',j'+1}$ on the grid $G_{i',j'}$. More details on this approach can be found in [32].

When faults affect multiple solutions, the same approach can be applied as long as there is no overlap in the combination coefficients that need to be changed. In particular if faults only affect two solutions then this is true except where failures affect two solutions $u_{i',j'}$ and $u_{i'-1,j'-1}$ with $i' + j' = l$, in which case many coefficients must be changed. This is also the only case of two failures in which coefficients with $i + j < l - 2$ become non-zero.

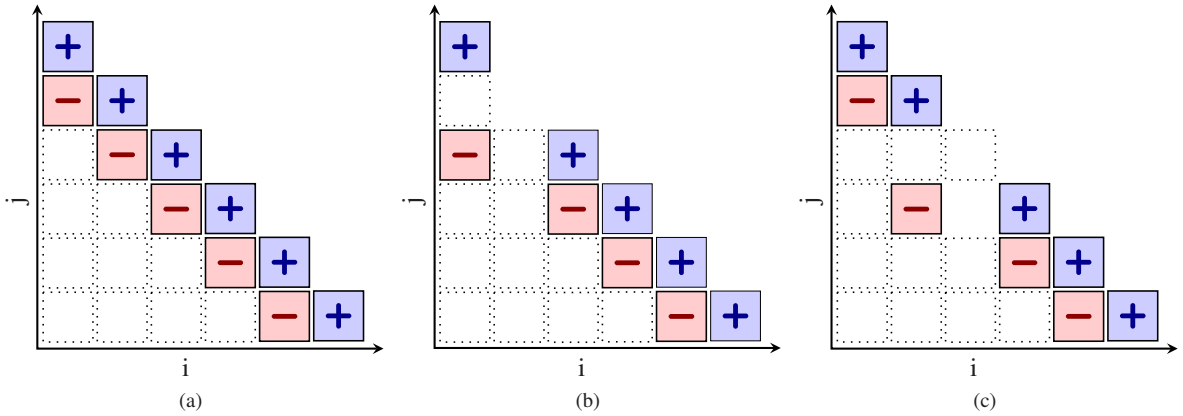

Fig. 2. Sparse grid combination method: (a) classic sparse grid combination for $l = 5$, (b) combination in the event of a lost component solution $u_{1,4}$ and (c) combination in the event of a lost component solution $u_{2,2}$.

This fact means that if there is an extremely low probability of having more than one failure, one may wish to only compute the grids $G_{i,j}$ with $l - 2 \leq i + j \leq l$ instead of the entire downset.

Finally, we mention here the paper [30] where a very efficient new data structure is introduced to store a sparse grid. This data structure is shown to lead to efficient new algorithms for hierarchisation – the determination of the coefficients of the sparse grid hierarchical basis. This representation does allow for very fast evaluation of the sparse grid approximation on a collection of grid points and thus effective post-processing.

5. Prototype Implementation for the 2D Advection Equation

Our test problem in the development of a framework is the scalar advection equation in 2–dimensions which for $u := u(t, x)$ is given by

$$\frac{\partial u}{\partial t} + \nabla \cdot (\underline{a} u) = 0 \,. \tag{3}$$

With $\underline{a} = (1, 1)$ we consider the solution for $x \in [0, 1)^2$ with periodic boundary conditions and smooth initial condition $u(0, x) = \sin(2\pi x_1) \sin(2\pi x_2)$. In our numerical simulation we solve up to $t = 0.5$ for which the exact solution is $u(0.5, x) = u(0, x)$. Our implementation uses PETSc [33] with centred finite difference discretisation of spatial derivatives and Runge–Kutta for the time–stepping. The code has been developed so that it can compute a numerical solution to this problem on any regular grid whose dimensions can be set with command line arguments.

In this section Map() and Reduce() will refer to generic functional programming Map and Reduce operators and not to the corresponding functions used in the Google MapReduce framework. We implement parallel versions of these operators using Python and mpi4py [34].

We require two fundamental data structures in our implementation, one for the collection of component grids and one for the sparse grid. The collection of component grids is represented by a downset I of level pairs (i, j) as explained in Section 4. The $(i, j) \in I$ are also used as keys to access the component solutions $u_{i,j}$. The sparse grid is implemented as a hash table (Python dictionary) with the keys (x, y) and values $u(\frac{x}{2^l}, \frac{y}{2^l})$ where $(\frac{x}{2^l}, \frac{y}{2^l})$ are the sparse grid nodes.

The Map stage computes the solution of the PDE for all of the component grids $G_{i,j}$. Note that these computations are all independent and can therefore be computed simultaneously. They scheduled and managed by several MPI processes. For each grid $G_{i,j}$ MPI_SPAWN is called and a new inter–communicator is generated for communication between the child and parent processes. On completion the child process sends a message to the parent process. This requires only a minor modification of the application code which computes the solution on each component grid.

The Reduce stage computes the sparse grid function u from the componont grid functions $u_{i,j}$ using the formula $u = \sum_{(i,j)\in I} c_{i,j} u_{i,j}$. In the current version this reduction operation is done sequentially.

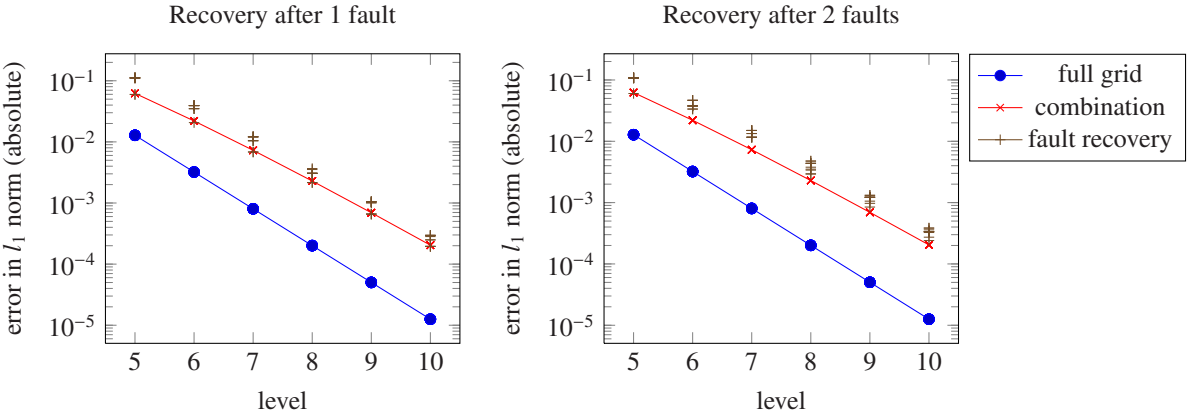

Fig. 3. Comparison of errors after recovery when faults affect different component grids. The first plot is when one grid is affected by faults and the second is when two grids are affected by faults. We recover from faults by deriving new combination coefficients which avoid faulty grids. The combination solutions have been interpolated to the full grid for the calculation of error and comparison with the full grid.

The reduction step is where the algorithm based fault tolerance is implemented. This is different to Google's MapReduce where fault tolerance is implemented at the Map() stage. As discussed in the previous section we have two basic approaches: either the combination coefficients are adapted so that the missing components $u_{i,j}$ have no influence, or the missing $u_{i,j}$ are recovered from the other component solutions.

Each component grid is interpolated onto the sparse grid and stored in an array using the structures and dictionaries previously described. Since interpolating to the sparse grid array as given by the dictionaries provides a consistent layout of the data then the reduction step is as simple as applying the combination formula element-wise to the sparse grid arrays.

We simulate the effect of faults in the Map() stage by either deleting component grids before reaching the Reduce() stage or by randomly removing a component grid from the list before starting the Map() stage. This allows us to test the Reduce() stage's ability to identify missing component grids and then reconfigure the combination. We then compare the result with the exact solution and the approximate solution when no failures are simulated.

In Figure 3 we demonstrate the errors obtained when recovering from failures by obtaining new coefficients from the expansion of the projection operator. The grids with dimensions $2^i \times 2^j$ where $l-2 \leq i+j \leq l$ and $i, j \geq 2$ have been computed for each level l. We simulate one fault for the first plot and two faults for the second. We note that for the second plot we have excluded the cases of faults affecting two component grids $u_{i,j}$ and $u_{i-1,j-1}$ where $i + j = l$. The errors appear to be relatively small and decrease uniformly with increasing level. There are also some cases where the error after a fault occurs is very close to the error when no faults occur. Where faults affect two grids we note that the maximal error is a little larger but still appears to decrease uniformly with increasing level. In all cases, the degree of error with faults is within a factor of 3 of the sparse grid error without faults.

Comparing with the full grid results, we observe that a full grid with level l has approximately the same error of a sparse grid error at level $l + 2$ (i.e. the maximal resolution is four times that of the full grid), while requiring less than $\frac{l+2}{2^{l-3}}$ of the computational work and memory.

6. A Modified MapReduce Programming Pattern for Scalable Fault-Tolerant PDE Solvers

The 2D advection application described in Section 5 employs the MapReduce functional programming pattern, but the interface between the Map and Reduce stages differ from the MapReduce application programming interfaces offered by frameworks such as Google MapReduce, Hadoop, or the MapReduce-MPI library. Below we describe how this prototype could be refactored to use popular MapReduce frameworks and discuss the limitations of these frameworks with respect to our method. We then propose a modified MapReduce programming model and the technical challenges and design questions we face in designing and developing infrastructure to support it.

The sparse grid combination technique described in Section 4 can in principle be implemented within a standard framework such as Hadoop. The Map() function takes initial and boundary conditions and computes a solution on each component grid, emitting <key,value> pairs. The key is the (x, y)-location on the full isotropic grid, which can be used as a sparse grid index. The value is the tuple $[(i, j), u_{ij}(x, y)]$. The Reduce() function processes all data associated with each sparse grid index key (x, y), using the component grid ID (i, j) to determine the coefficient of $u_{ij}(x, y)$ in the combination formula (1) to compute the solution on the sparse grid. Such an implementation would be fault-tolerant because the framework will reschedule any failed Map or Reduce tasks. We are not pursuing this approach using Hadoop because of its reported limited scalability to approximately 4000 nodes [35], reliance on disk I/O and ill-suitedness to the type of problem we are trying to solve. Hadoop's Java-based programming model creates language interoperability issues we wish to avoid imposing on scientific programmers who work primarily in Fortran, C/C++, and Python. Additionally, Hadoop's framework structure does not offer a sufficiently expressive programming model for the algorithmic fault-tolerant schemes we are developing. These issues motivate a reexamination and extension of the MapReduce programming pattern and the development of a software framework capable of supporting this approach. Below we discuss the requirements, design, and runtime scenario for a flexible MapReduce system for grid combination technique solvers.

Our primary aim is to support scientific computation on cluster supercomputers that employ standard programming models such as MPI and OpenMP running under job-queuing systems such as the Portable Batch System (PBS). This differs dramatically from the standard MapReduce application to large-scale data analysis on a dedicated system with a MapReduce-framework-specific queuing system and DFS. The volumes of data we anticipate handling in our sparse grid solver schemes are amenable to storage in memory, obviating the requirement of an application-specific DFS. We wish to avoid the single point of failure inherent in Hadoop's master node. In place of a single master node in MapReduce frameworks we propose a "manager" pool of PEs that creates and destroys "worker" PE pools for execution of Map and Reduce tasks, monitors worker pools to detect faults, responds to faults by modifying workloads or reassigning tasks, and arbitrates direct data transfer of Map outputs to Reduce inputs. We also wish to avoid the global sorting scheme and barrier that exist within standard MapReduce frameworks. In the place of disk storage and a large sorting scheme, we envision a system implementing distributed data description and parallel transfer/transformation mechanisms commonly employed in parallel multiphysics coupling [36]. This will entail storage of distributed state data in memory and an $M \times N$ [37] direct parallel data transfer mechanism between Map and Reduce nodes. The $M \times N$ transfer connections and communications schedules will be computed from user-supplied definitions of component grids and their associated domain decompositions. These parallel data transfers will likely be implemented using a peer communication scheme such as that offered by the Model Coupling Toolkit [38].

Our grid combination method strategy is to use legacy code solvers as the individual Map tasks, associating each task with a component grid. The programming model and framework must be minimally intrusive, requiring only a small number of modifications to the legacy code to render it as a Map. The set of interfaces a legacy Map code must call include a function for mapping component grid locations to sparse grid locations, description of distributed data to be transferred from Map to Reduce, handshaking of $M \times N$ data transfers, and execution of these parallel transfers. In addition to these code modifications the user must supply be a Task Cost Model (TCM) that relates problem size/component grid resolution to wall-clock execution time. This requirement is optional in the sense that not meeting it will limit the framework's ability to perform effective load balancing, estimate task progress, and detect task node failures. A final requirement imposed on the user is a solution quality of service (QoS) model that defines acceptable combination formulae and relates them to error estimates. The framework will supply a system for defining sparse grid combination formulae coefficients and their respective solution error bound estimates. The QoS model determines how the Reduce phase proceeds in the face of lost Map tasks.

A MapReduce framework detects worker PE faults by requiring periodic progress reports from worker nodes as they carry out their tasks. In Hadoop, this is implemented as a periodic "heartbeat," and the master node assumes a worker node has failed if no heartbeat is detected after a predefined waiting period. This approach works well in Hadoop for up to 4000 nodes, but may need to be reexamined for petascale systems. A hierarchical heartbeat monitoring system is one possibility. Another approach is to use the TCM to allow the manager nodes in the framework to predict how long a given Map or Reduce task will take to execute, and to take corrective action at some point after this TCM-predicted execution time interval has been exceeded. Responses to task failures will depend on the type of task lost. The framework will be sufficiently flexible to allow dropping—as

opposed to rescheduling—of failed Map tasks and selecting a best—in terms of lowest error estimate—available alternative combination formula for the Reduce phase. Completion of Reduce tasks will be guaranteed either through replication or re-scheduling.

The likely runtime narrative for the flexible MapReduce framework will be as follows: The framework will be an application run on a cluster through a queuing system. The first prototype will use MPI and be invoked using `mpiexec`. At startup the framework's manager nodes occupies N_{man} PEs. The manager nodes load configuration information defining the Map application(s), component grids, sparse grid, combination formulae with associated error estimates, and—if supplied—TCM data. The manager nodes then spawn Map tasks, either invoking them as functions on a portion of a statically-defined global PE pool, or invoking them with as separate executables using MPI_SPAWN(). The manager pool will then determine the ideal layout of the Reduce task pools and the mapping of computations to them. This information will be combined with the layout of the Map tasks to compute the communications schedule for the $M \times N$ Map-to-Reduce parallel data transfer. The manager pool will initiate a set of Reduce tasks, advising them of their respective $M \times N$ transfer operations. The Map tasks will also be advised of the $M \times N$ transfer operations they will execute once their calculations are completed. The Reduce tasks will post nonblocking receives and await data from the Maps. During execution of the Map tasks the framework will monitor them for failures, either through communication or by comparing the time elapsed with respect to the execution time predicted by the TCM. Notice of Map task failures detected by the manager nodes will be passed on to the Reduce task nodes to avoid waiting on data that will not arrive. This mechanism will also allow the Reduce task nodes to pick the appropriate combination formula. Upon completion of the Reduce phase, its output will be written and all remaining task PE pools will be terminated by the manager pool before it exits.

7. Conclusions and Future Work

Petascale and exascale computer systems will pose unprecedented scalability challenges to scientific software developers. The vast number of hardware components on these systems, combined with finite probability of failure for each component, will make fault-tolerance a core requirement for future HPC applications. Building robust codes in such environments will likely require using multiple fault-tolerance techniques. We believe that ABFT, though application-specific, will be a key technology in meeting these challenges. We have proposed an ABFT technique for constructing robust PDE solvers based on the sparse grid combination technique. We have outlined the theoretical basis of the technique, and demonstrated its potency in terms of reduction of computational complexity and its inherent ABFT properties. We have applied the sparse grid combination method to a benchmark problem—2D advection—in a framework based on the MapReduce functional programming pattern. We have found that the method delivers acceptable errors, even in the presence of simulated faults.

The results presented here are sufficiently promising to merit further research and development of these algorithms and of an ultrascale software framework capable of delivering the fault-tolerant MapReduce variant described in Section 6. Areas of future investigation will include: extension of the sparse grid combination method to other systems and in particular higher-dimensional systems; investigation of the feasibility of these techniques on other spatial grid structures; a detailed requirements and design analysis for a flexible ultrascale MapReduce framework; and implementation of a full working system.

Acknowledgment

This work was supported by the Australian Research Council and Fujitsu Laboratories of Europe through the ARC National Competitive Grants Program (NCGP) Linkage Project LP110200410.

References

[1] F. Cappello, Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities, International Journal of High Performance Computing Applications 23 (3) (2009) 212–226. arXiv:http://hpc.sagepub.com/content/23/3/212.full.pdf+html, doi:10.1177/1094342009106189.

[2] W. Gropp, E. Lusk, Fault tolerance in MPI programs, Special issue of the Journal High Performance Computing Applications (IJHPCA) 18 (2002) 363–372.

[3] K.-H. Huang, J. A. Abraham, Algorithm-based fault tolerance for matrix operations, IEEE Trans. Comput. 33 (6) (1984) 518–528. doi:10.1109/TC.1984.1676475.
[4] G. Bosilca, R. Delmas, J. Dongarra, J. Langou, Algorithm-based fault tolerance applied to high performance computing, J. Parallel Distrib. Comput. 69 (4) (2009) 410–416. doi:10.1016/j.jpdc.2008.12.002.
[5] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, USENIX Association, Berkeley, CA, USA, 2004, pp. 10–10.
[6] J. Dean, S. Ghemawat, MapReduce: a flexible data processing tool, Commun. ACM 53 (1) (2010) 72–77. doi:http://doi.acm.org/10.1145/1629175.1629198.
[7] M. Griebel, M. Schneider, C. Zenger, A combination technique for the solution of sparse grid problems, in: P. de Groen, R. Beauwens (Eds.), Iterative Methods in Linear Algebra, IMACS, Elsevier, North Holland, 1992, pp. 263–281.
[8] H.-J. Bungartz, M. Griebel, Sparse grids, Acta Numerica 13 (2004) 147–269.
[9] J. Garcke, M. Hegland, O. Nielsen, Parallelisation of sparse grids for large scale data analysis, in: P. S. et al. (Ed.), ICCS 2003, Vol. 2659 of LNCS, Springer-Verlag, 2003, pp. 683–692.
[10] J. S. Plank, K. Li, M. A. Puening, Diskless checkpointing, IEEE Transactions on Parallel and Distributed Systems 9 (10) (1998) 972–986. doi:http://doi.ieeecomputersociety.org/10.1109/71.730527.
[11] E. N. M. Elnozahy, L. Alvisi, Y. Wang, D. B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surv. 34 (3) (2002) 375–408. doi:10.1145/568522.568525.
[12] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso, Understanding replication in databases and distributed systems, in: In Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000, 2000, pp. 264–274.
[13] Draft MPI-3 standard, `www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf` (2012).
[14] J. Hursey, R. Graham, Building a fault tolerant mpi application: A ring communication example, in: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, 2011, pp. 1549 –1556. doi:10.1109/IPDPS.2011.308.
[15] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, F. Cappello, Non-blocking coordinated checkpointing for large-scale fault tolerant MPI, in: Proceedings of The IEEE/ACM SC2006 Conference, 2006.
[16] J. Hursey, J. M. Squyres, T. I. Mattox, A. Lumsdaine, The design and implementation of checkpoint/restart process fault tolerance for Open MPI, in: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 2007.
[17] H. Ltaief, E. Gabriel, M. Garbey, Fault tolerant algorithms for heat transfer problems, Journal of Parallel and Distributed Computing 68 (2008) 663—-677.
[18] A. Srinivasan, N. Chandra, Latency tolerance through parallelization of time in scientific applications, Parallel Computing 31 (7) (2005) 777–796.
[19] Open Petascale Libraries, `http://www.openpetascale.org/`.
[20] Hadoop Web Site, `http://hadoop.apache.org/`.
[21] Parallel MapReduce in Python in ten minutes, `http://mikecvet.wordpress.com/2010/07/02/parallel-mapreduce-in-python/`.
[22] MapReduce-MPI web site, `http://mapreduce.sandia.gov/`.
[23] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, B. Moon, Parallel data processing with MapReduce: a survey, SIGMOD Rec. 40 (4) (2012) 11–20. doi:10.1145/2094114.2094118.
[24] C. Zenger, Sparse grids, in: Parallel algorithms for partial differential equations (Kiel, 1990), Vol. 31 of Notes Numer. Fluid Mech., Vieweg, Braunschweig, 1991, pp. 241–251.
[25] M. Hegland, Adaptive sparse grids, in: K. Burrage, R. B. Sidje (Eds.), Proc. of 10th Computational Techniques and Applications Conference CTAC-2001, Vol. 44 of ANZIAM J., 2003, pp. C335–C353.
[26] C. Kowitz, M. Hegland, The sparse grid combination technique for computing eigenvalues in linear gyrokinetics, in: International Conference on Computational Science, ICCS 2013, Procedia Computer Science, Elsevier, 2013, submitted.
[27] GENE web site, `http://www.ipp.mpg.de/fsj/gene/`.
[28] H.-J. Bungartz, M. Griebel, U. Rüde, Extrapolation, combination, and sparse grid techniques for elliptic boundary value problems, Comput. Methods Appl. Mech. Eng. 116 (1994) 243–252.
[29] M. Hegland, J. Garcke, V. Challis, The combination technique and some generalisations, Linear Algebra and its Applications 420 (2–3) (2007) 249–275. doi:10.1016/j.laa.2006.07.014.
[30] G. Buse, D. Pfluger, A. Murarasu, R. Jacob, A non-static data layout enhancing parallelism and vectorization in sparse grid algorithms, in: Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on, 2012, pp. 195 –202. doi:10.1109/ISPDC.2012.34.
[31] M. Griebel, A domain decomposition method using sparse grids, in: A. Quarteroni (Ed.), Domain decomposition methods in science and engineering, the 6th International Confernce on Domain Decomposition, 15.-19. Juni 1992, Como, Italy, Contemporary Mathematics, Providence: American Mathematical Society, 1994, pp. 255–261.
[32] B. Harding, M. Hegland, A robust combination technique, in: Computational Techniques and Applications Conference, CTAC 2012, ANZIAM Journal, Cambridge University Press, 2012, submitted.
[33] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc Web page, http://www.mcs.anl.gov/petsc (2013).
[34] L. Dalcin, mpi4py Web page, http://mpi4py.googlecode.com (2013).
[35] K. V. Shvachko, Apache Hadoop: The scalability update, USENIX ;login: 36 (3) (2011) 7–13.
[36] J. W. Larson, Ten organising principles for coupling in multiphysics and multiscale models, ANZIAM Journal 48 (2009) C1090–C1111.
[37] F. Bertrand, R. Bramley, D. E. Bernholdt, J. A. Kohl, A. Sussman, J. W. Larson, K. Damevski, Data redistribution and remote method invocation for coupled components, J. Parallel Distrib. Comput. 66 (7) (2006) 931–946.
[38] J. Larson, R. Jacob, E. Ong, The Model Coupling Toolkit: A new Fortran90 toolkit for building multi-physics parallel coupled models, Int. J. High Perf. Comp. App. 19 (3) (2005) 277–292. doi:10.1177/1094342005056115.