A Case Study of Application Structure Aware Resilience Through Differentiated State Saving and Recovery

Anshu Dubey[1(✉)], Hajime Fujita[2], Zachary Rubenstein[2], Brian Van Straalen[1], and Andrew A. Chien[2,3]

[1] Computational Research Division Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
adubey@lbl.gov
[2] Computer Science Department, University of Chicago, Chicago, IL 60637, USA
[3] Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, Lemont, IL, USA

Abstract. Resilience is a growing concern for large-scale simulations. As failures become more frequent, alternatives to global checkpointing that limit the extent of needed recovery become more desirable. Additionally, platforms will differ in both error rates and types, therefore, a flexible and customizable recovery strategy will be extremely helpful to the applications running on these platforms. Applications often have structures that provide logical confinement spaces that can be exploited for this purpose. We investigate a customizable recovery strategy using Chombo, a structured adaptive mesh refinement (SAMR) library, as a case study. We exploit the inherent granularities and hierarchy in SAMR to limit the impact of faults for localized recovery, and identify tunable parameters for customizing the strategy depending upon the application and platform behavior. We use Global View Resilience (GVR) library, which provides global versioning arrays for application-controlled state saving as our resiliency interface.

1 Introduction

In order to effectively utilize future large-scale, high performance computers, applications will face several challenges, including more frequent failures that manifest themselves in various ways. The usual mode of checkpoint-restart is already reaching the limit of its usefulness in large-scale simulations where a non-trivial fraction of execution time is taken up by the checkpointing process. The restarts tend to be even slower than the checkpoints and even a node failure every few hours can prove to be significantly detrimental to the applications' runtime and computational efficiency. The checkpoints and restarts are slow because they use global snapshots and parallel file system for read and write. As the number of nodes involved in a calculation increases, the mean time between node failures proportionately decreases. Node failures usually cause an abort followed by job termination in the batch queue.

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 619–630, 2015.
DOI: 10.1007/978-3-319-27308-2_50

To deal with frequent failures, several applications have developed the ability to avoid "abort" when there is a failure, thereby avoiding the job termination. The application is able to restart itself from a previously saved checkpoint in the same job slot and continue execution. However, there is still the cost of reading from the file system and restarting, and the cost of lost calculations since the last checkpoint. There are many efforts to provide more efficient ways of checkpointing such as: non-blocking multilevel checkpointing [12], replication-based checkpointing [15], or localized checkpointing/restart [13]. Although these, and other new technologies that will eliminate the involvement of the file system in checkpointing, are expected to reduce the costs of state saving substantially, these technologies do not adequately address the variability of error rates in different application instances. For that we need a flexible, local and customizable recovery strategy.

One option for local recovery is to carry redundant information on surrounding nodes from which a consistent global state can be reconstructed in case of a node failure. This approach was explored in an oct-tree based SAMR code, FLASH [4,5], in [6]. Another option is to exploit the structure within the application for local state saving recovery, for example [13]. Some methods combine more than one approach to achieve fault tolerance, for example [14] makes use of adaptive fault-tolerance through both checkpointing and redundancy. In this paper we take a multi-pronged approach to fault tolerance. Similar to containment domains we use the application structure to confine the extent of recovery [2]. But unlike containment domains we tailor our recovery mechanisms to specific error modes. Additionally, we make use of tuning parameters based on cost-benefit analysis to explore the available trade-offs.

We use Chombo [3], a general purpose SAMR library, augmented with the Global View Resilience (GVR) library, [7] as a case study for our approach. GVR leaves the definition of the consistent state to be saved up to the application. This allows applications to decompose their state into smaller, more localized snapshots from which recovery can be effected with minimal impact or interaction with the overall global state. Additionally, the versioning capability of GVR further reduces the cost of saving individual local snapshots as long their definition does not undergo any change. Also, the co-existence of several versions of the snapshot (the extent of versioning is also under application control) gives further flexibility in devising the recovery strategy.

The Chombo-GVR interface can recover from both resource failure and transient data corruption. Recovery from resource failure itself can be either global or local depending on circumstances. The transient data corruption errors within a node can be handled locally if they are detected immediately and their extent is known to be confined to the node. Depending upon the difficulty and/or delay in transient error detection (which is out of scope of this paper) a non-local recovery may be necessary. In future we may also include forward error correction that could use the corresponding data from coarser parts of the mesh for reconstruction similar to [6].

The focus and main contribution of this paper is devising a recovery strategy that exploits the application's structure and granularities, and provides tunable parameters for customization. We take the approach of identifying the logical confinement regions of the state spatially and temporally, and examining the error modes that can be mapped to these regions. We then devise a recovery strategy for each error mode and map it to the corresponding granularity in the application. In the final step we model the overheads for cost-benefit analysis. Though we use SAMR as our case study, our approach should be equally useful to other applications which have nested granularities or hierarchies. The paper is organized as follows, we first give a brief description of the two libraries, Chombo and GVR, in Sect. 2, followed by a discussion of the local saving and reconstruction strategy in Sect. 3. Section 4 describes preliminary cost measurements of the overheads introduced into the application by the resiliency strategy, and how they can be used as tuning parameters for specific computing platforms and/or specific instances of application use. Finally, conclusions and future work are discussed in Sect. 5.

2 The Libraries

Chombo defines patches of uniform resolution (finer) that are embedded within other patches of lower resolution (coarser) as shown in Fig. 1. Patches can be subdivided into boxes, all boxes with the same resolution constitute a level, where individual boxes may be distributed arbitrarily in physical space as long as they are fully enclosed within a region of next level of coarse resolution. Therefore a level can be viewed either as a logical entity (same resolution) (see Fig. 2) or a physical entity (union of all boxes at the same resolution), and Chombo's data structures allow either view. The solution advances with the same time-step everywhere on a given level, though it differs from other levels if subcycling is being used. With subcycling, for a refinement ratio $\mathbf{R}$ the finer level takes $\mathbf{R}$ steps for every step taken by the embedding coarse level. Most of bookkeeping in Chombo is managed on a level-by-level basis, with some cross-level management. The level meta-information includes knowledge of all boxes with their integer index space within the global mesh. This information can be harnessed to determine the adjacency of boxes for the purpose of filling the halo of ghost cells as needed. The information is also used to distribute the load among processors. The cross-level management is used for purposes such as filling the ghost cells for boxes that exist at fine-coarse boundaries, to reconcile various physical quantities such as fluxes as the fine-coarse boundaries, and synchronization of time-advancement when there is sub-cycling.

The depth of AMR hierarchy depends upon the scientific domain and the specific application. The count and shape of the patches and their boxes is not static, it changes as necessary when the solution evolves. The finer patches follow more structure in the solution space, with the finest ones existing where there is maximum structure and therefore smallest length scales in the physical domain. With subcycling the finer levels do considerably more work than the

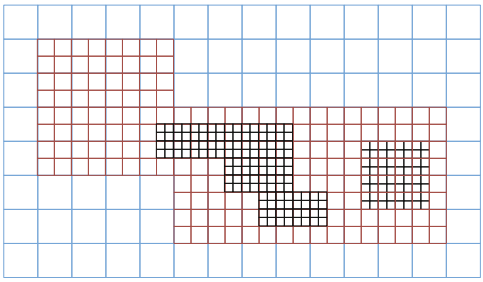

Fig. 1. SAMR mesh showing three levels of resolution.

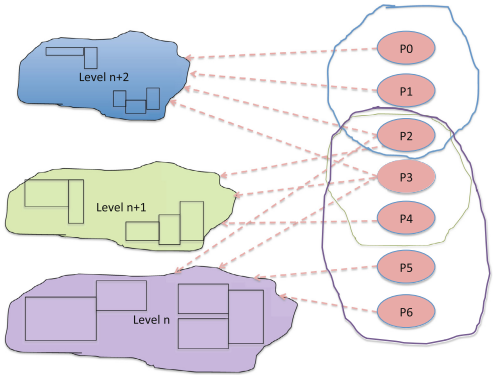

Fig. 2. A logical view of multiple levels of resolution in an AMR mesh.

coarser levels, which is why each level does its own load distribution. This way all compute resources get a mix of levels and therefore roughly equal amount of work.

GVR is a lightweight library which enables applications to run reliably on unreliable computers. It provides two main features: multi-version, multi-stream distributed arrays and a unified error handling interface. GVR provides PGAS-style distributed arrays (similar to Global Arrays [11]), but extends them with primitives to create persistent *versions* [16]. Multiple versions enable applications to perform more powerful recovery from complex errors such as *latent errors*, which cannot be detected immediately (see [8]). Different arrays can choose optimal versioning cadences depending on the array. GVR also provides a unified error handling interface for various error sources, through which application can receive error events and handle the errors. This allows application programmers to reuse an error handler for different error causes, thus reducing the cost for writing error handlers.

As mentioned in Sect. 1, GVR allows the application to define what constitutes a consistent recoverable state, and what goes into each array. In addition, GVR places no restriction on how many arrays can be defined by an application, and being PGAS-style distributed arrays, provides random access anywhere

within an array. The GVR interface works by allocating arrays as requested by the application, and then letting the application store data through "put" functions. When it is safe to save the state of an array, the application can create a version through the"version_inc" function. Created versions are accessible to the application by specifying the appropriate version number. The "put" functions allow random access within the array. Similar to "put", the "get" function is used to fetch arbitrary information from a specified version of an array. There are also no restrictions on the amount of information that can be fetched in one instance of using "get". Therefore the granularity of information exchanged with an array is also under the control of the application. These flexible accessibility features of GVR, and its low overheads, make it particularly attractive for exploring containment and differentiation based recovery strategies in applications where the structure of the application can be exploited for this purpose.

GVR takes two measures to protect the contents of the array. The first measure is preserving multiple versions. GVR's API defines that old versions are read-only, which makes it easier for the library to store them in a different location or to apply coding, compression, encryption, etc. to the data. Then as a second measure, GVR stores an old version of an array in a different process' memory or a secondary storage such as node-local SSD or parallel shared file system, in order to protect the array data from resource failures such as node/process crash. When storing old versions in a secondary storage, GVR utilizes Scalable Checkpoint Restart [9] to exploit multi-level storage hierarchy and its data protection schemes for node-local storage.

3 Resilience Strategy

Our resiliency strategy is based on five simple ideas.

- Check if there is any hierarchy to be exploited in the application's structure.
- Identify granularities within the application that can be used to confine the impact of the fault and recovery from it.
- Consider the types of faults that the strategy is expected to handle, and map recovery from each fault type to the appropriate granularity of the application.
- At each granularity determine the minimum state to be saved to effect a full recovery.
- Identify tuning parameters to enable customization to specific instances of application execution.

The default checkpointing in Chombo, like many other production-grade high-performance computing (HPC) applications, saves a global snapshot of the state into a parallel file using HDF5 [10] library. Such global snapshots are saved at regular intervals, and therefore an attempt is made to minimize the amount of saved information that can be used to reconstruct the complete AMR hierarchy without any loss upon restart. For this purpose Chombo needs to save a very small amount of global state meta-information and also a moderate amount of level-specific meta-information in addition to the physical data residing on each

box. Because the snapshot is global, it can only be done when all the levels are synchronized. By definition, that is the point at which the coarsest level completes its timestep.

In order to formulate a differentiated recovery strategy for AMR we consider its inherent structures. An obvious coarse granularity in AMR is a "level" which is an almost self contained unit of computation. The meta-information of a level includes complete knowledge of all existing boxes and their mapping to the processes, the evolution time and the timestep (dt), which is uniform for a level. Because the boxes in AMR are dynamically created and destroyed, a level can also do its own regridding and distribution of boxes among processors. A level interacts with the level below and the level above at the fine-coarse boundaries. The second, finer granularity exists at the "box" level. A box with its surrounding halo of ghost cells is a complete computational unit for the operators being applied to the field variables. If an error occurs in a box and can be detected before it spreads out to other boxes, it should be possible to completely confine the recovery to the affected box.

3.1 Saving Chombo State with GVR

For saving the state using GVR we view each level in the AMR hierarchy as a loosely independent component. Therefore, we construct one array for each level which includes the meta information as well as the physical data on all the boxes at that level. Figure 3 shows an example of mapping one AMR level to one array. For each box in the level, we include its index-space on the discretized mesh and the offset within the global array of the physical data belonging to the box. In order to complete a global snapshot we only need to add one more array for the global meta-information such as the number of field variables, the depth of mesh hierarchy, the refinement ratio between levels and the array version numbers for each level.

However, AMR presents an additional challenge to the GVR model. In a normal mode GVR operates by allocating resources for an array, which remain fixed throughout its existence. Versioning is used to preserve the array state as needed. This assumption is not valid for AMR where the resources used by a level change whenever there is a regridding of the mesh at that level. This limitation can be overcome in one of two ways, each with its own advantage and disadvantage. One option is to free the existing array for the level and reallocate a new one every time there is regridding. This option minimizes the space used by the array but may not be very efficient in time because there will be cost of freeing and reallocating the resources. This is more important when the cost of allocation is higher than that of incrementing the version. The other option is to allocate more space for the level's array than needed and free and reallocate only when the required space exceeds the current allocation. This option may be more efficient time-wise, but is not as efficient in space usage. We have chosen the second option with extra allocation being a tunable parameter, where setting the extra allocation to zero optimizes the space usage, and can give better performance time-wise if the size of the mesh data for the level does

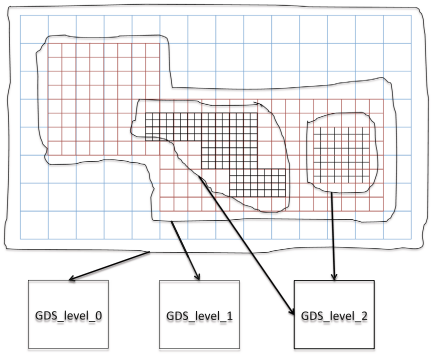

Fig. 3. Mapping AMR levels to GVR arrays

not increase with every regrid event. Such a scenario can arise, for example, where the higher refinement is following a shock. Here the part of the physical domain which is highly refined changes with time, but the amount of highly refined domain may not change very much.

Populating the Arrays. The initial state saving requires making an estimate of space to be allocated for the global meta-data and the level arrays. We populate them at the beginning of the simulation as soon as the mesh is initialized. The cadence of state-saving at each level is a tunable parameter. When it is time to save the state, the physical data of the boxes and meta-information such as "current_time" that change at every timestep are updated in the array and the version of the array is increased. A regrid event has to be treated differently. It may be necessary to reallocate an array for a level if its size has grown more than the allocation for the current array. In addition to all quantities updated at a normal timestep state save, with regrid the index-space for the boxes and their offsets also change, so they have to be updated. When no regridding is involved one can either complete a timestep computation for a level and put all the boxes data into the array at once, or one can put individual box's data as soon as it is computed within the loop. This is another tuning parameter which can be used only when state is being saved at every timestep at every level.

3.2 Failure Scenarios and Recovery Modes

We target the following failure and recovery scenarios in designing our strategy.

Permanent Resource Failure: The most common manifestation of resource degradation is node failure where recovery implies a restart on fewer nodes. Detection and notification to the application is designed be to under the control of GVR. The recovery is either global with a full restart at all levels, or it can be done at the granularity of the affected levels if situation allows. The computation will be rolled back at minimum to the beginning of the last saved timestep of the

coarsest affected level. In the worst case it will have to roll back to the beginning of the last coarsest timestep saved. In the present version of Chombo resource failure recovery ends up defaulting to the global mode because all levels in the AMR hierarchy distribute their work on all processors. The instances where some levels don't have any boxes on a processor are rare. Deep AMR hierarchies with highly localized refinement patterns may benefit from non-global recovery, but those are also rare among current suite of applications using Chombo.

Temporary Resource Failure: We assume that a temporary resource failure implies that none of the data on that resource is reliable, however there is no need to reconstitute and restart. In this situation recovery at the granularity of affected levels by fetching the data from the corresponding GVR array will suffice if the snapshots were being saved at every timestep of every level. If the cadence of saving was different, consistent recovery may have to rollback further. The detection and notification to the application is again up to GVR.

Data Corruption: Data corruption is detected by the application and is a research area in its own right. In this work we are not focussing on fault detection and injection methods, except one simple detection method discussed below (also see [1] for a similar method). For many AMR applications corruption in a box can be detected by checking for $|x_n - x_{n-1}| < \epsilon$. Here x_n and x_{n-1} are the newly computed and the previous timestep's values respectively at a point in space. And ϵ is largest valid change in value at any point between two consecutive timesteps in a given operator as determined by the application expert. If data corruption is detected, we recompute x_n. There can be two possible outcomes of recomputing: the new value is the same, or it is different. If it is different from the original calculation and falls within the valid range we can either accept the value or recompute it one more time to verify correctness if more confidence is desired. However, if several errors are detected in the same box the computation for the whole box should roll back to the beginning of the timestep. The granularity of recovery is confined to a box if the last version increment was at the beginning to the current timestep. If that is not true, one has to determine the closest coarse level "m" that had a version increment at the beginning of its timestep. All finer levels including the current level and "m" have to roll back to the beginning of m's timestep. If "m = 0" it effectively becomes a global restart. If the new value is different and still outside the valid range, or if it is identical to the previously calculated one, then a more systemic problem is indicated, and the recovery may need to be abandoned. The more appropriate thing to do would be to trigger diagnostics to determine the overall state of the simulation data to see if an abort is necessary.

Meta-data Corruption: Meta-data corruption is more difficult to detect but easier to recover from. The reason why detection is difficult is because the data elements have no inherent correlation with each other. Neither is there any evolution in the values of most data elements. In current version of Chombo the meta-data is replicated on all processors, so it can simply be fetched from one of the neighbors. And if the saving is being done every timestep then it can also

be fetched from GVR array. The granularity of recovery for this type of error is also at the box level. For all data corruption recoveries the random access into the GVR array is the crucial feature.

4 Tuning for a Specific Platform

Computing platforms vary in their rates and extent of failure. This will be particularly true of the future large scale platforms. Also, depending upon the specific problem being solved, the same code may run at different machine scales at different times. For some runs a local cluster is sufficient while for others a large fraction of a leadership-class machine may be necessary. Therefore, one strategy with fixed parameters is not likely to be an efficient resiliency solution everywhere. The best way to make a strategy flexible is to turn the parameters into tuning knobs wherever possible. In the current version of our strategy the tuning parameters can be: (1) the timing of various snapshots at each level, (2) whether to allocate larger array or reallocate for every regrid, (3) save as soon as every box is done or after all boxes are done, and (4) whether to trigger diagnostics upon detecting unrecoverable data error or abort.

We illustrate the cost-benefit analysis with the example of frequency of snapshots as the tuning parameter. In a state-save-recovery scenario the cost of recovery is $T_{save} + T_{lost} + T_{reconfig}$, where T_{save} is the cost of saving the snapshot, T_{lost} is the cost of lost computation which has to be redone and $T_{reconfig}$ is the cost of fetching data and reconfiguring as needed. For any level T_{save} for one snapshot is a fixed cost in between regridding steps. It includes the cost of putting necessary meta-data and the physical data from all blocks into the GVR array and incrementing the version. At regridding time it is the cost of possibly reallocating an array, putting all the meta-data and physical data into the array and incrementing its version. $T_{reconfig}$ can be as little as the cost of reading back a box's data from the last GVR array version, or it could be as large as the complete global restart. T_{lost} depends upon where the error occurred and how far back the application has to roll back.

To get an estimate of the costs involved we ran an experiment with a gas-dynamics problem where a shock hits a ramp, using built-in timers in Chombo for measurements. The quantities we measured for the AMR part of the code are the overall runtime and t_{levn}, the time to compute one timestep at level n, and $t_{reconfig}$ time for reconfiguring a level for a restart. We also measured the time to write a complete checkpoint file to the filesystem (t_{file}). For GVR we measured T_{alloc}, the time to allocate a GVR array; T_{box} time to put/get (they are very similar) one box worth of data; T_{level}, time for putting away or getting one level worth of data. The experiment was run on Edison, the Cray machine at NERSC using 128, 256, 512 and 1024 cores. For the experimental setup we used 3 levels of refinement (in all a hierarchy of 4 levels), with the problem size weak-scaling as the number of cores is increased. The parallelization model is pure MPI for the AMR, and distributed arrays for GVR. The quantities displayed in the tables are taken from measurements on rank 0 because there is very little variance in

Table 1. Measured AMR quantities in the application.

Procs	Run	t_{lev0}	t_{lev1}	t_{lev2}	t_{lev3}	$t_{reconfig}$	t_{file}
128	1581	0.82	2.65	2.91	3.13	0.63	60.33
256	1639	1.72	2.78	3.01	3.15	0.75	61.67
512	1923	1.82	2.86	3.03	3.16	1.58	125.67
1024	1841	2.19	2.83	3.01	3.18	2.9	52.67

Table 2. Measured GVR related quantities for saving state.

Procs	T_{alloc}	T_{level}	T_{box}	T_{verInc}
128	0.22	1.75	0.0048	1.02
256	0.04	1.82	0.0055	1.07
512	0.61	1.37	0.0081	0.41
1024	2.36	3.25	0.0092	1.23

timing between cores since the application operates in bulk-synchronous mode. This set of measurements are only meant to highlight the use of tuning knobs. Experimentation in many more computing environments under different fault conditions will be necessary to formulate a full cost model for AMR resiliency, and will be part of our future work.

Now let us consider a scenario where there was data corruption error in the final timestep of the finest level running on 512 cores. We can compute the cost of recovery under two contrasting snap-shot saving regimes. One where only global snapshots are being taken, and another one where every level takes its own snapshot at each one of its timesteps. Recollect that the global snapshots can only be taken when all levels are synchronized, which is the end of the coarsest time step. Here in the first scenario $T_{save} = 0.61 + 4 \times 1.37 = 6.09$ from Table 2. Assuming that we are doing subcycling $T_{lost} = 1.82 + 2 * 2.86 + 4 * 3.03 + 8 * 3.16 = 44.94$ from Table 1 and $T_{reconfig} = 1.58$ second, for the overall recovery cost of 52.61 seconds. For the second scenario the dominant cost is the saving cost because only the corrupted box will need to be read back from GVR and there is no need for reconfiguration. Again taking into account subcycling, $T_{save} = 0.61 + 1.37 + 2 * 1.37 + 4 * 1.37 + 8 * 1.37 = 20.55$, $T_{lost} = 3.16$ and $T_{reconfig} = 0.0081$, giving the cost of recovery at 23.72. The above analysis shows that different fault scenarios should consider different snap-shot saving regimes. When data corruption errors are infrequent the fixed cost of saving every timestep at every level will be an overkill. Whereas when the data corruption errors are frequent, taking the global snapshots only strategy could cause the application to repeat computations often, thereby costing much more than the fixed cost of T_{save} at higher cadence.

5 Future Work

We have presented a methodology for spatial and temporal decomposition of a complex but highly structured application in order to devise a differentiated resiliency strategy. This kind of approach is particularly important for the class of problems that do not have the option of devising fault-resistant computations. These applications have to rely on rollback-recovery, so minimizing its cost is very important to them. An important part of cost minimization is evaluation of the trade-offs between overheads and lost work with different error rates in different platforms and application instances. To facilitate this evaluation we have shown how tuning knobs can be incorporated in the strategy. One aspect of transient error recovery mode was not explored in this work: forward error correction through reconstructing from lower fidelity coarse grid data. Additionally, the cost model is preliminary, it does not take into account the possible delays, and therefore lost work, in transient error detection. Both these aspects of resiliency tend to be specific to an application instance and also domain dependent, and are not understood very well. Further out we will extend the forward-error correction work from [6] to cover more application domains that use AMR. As error detection in AMR matures, we will incorporate the error detection related costs in our model.

Acknowledgments. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy and completed in part with resources provided by NERSC, a DOE user facility supported by the Office of Science.

References

1. Berrocal, E., Bautista-Gomez, L., Di, S., Lan, Z., Cappello, F.: Lightweight silent data corruption detection based on runtime data analysis for HPC applications. Technical report (2014)
2. Chung, J., Lee, I., Sullivan, M., Ryoo, J.H., Kim, D.W., Yoon, D.H., Kaplan, L., Erez, M.: Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In: The Proceedings of SC12 (2012)
3. Colella, P., Graves, D., Keen, N., Ligocki, T., Martin, D., McCorquodale, P., Modiano, D., Schwartz, P., Sternberg, T., Van Straalen, B.: Chombo software package for AMR applications design document. Technical report, LBNL, Applied Numerical Algorithms Group, Computational Research Division (2009)
4. Dubey, A., Antypas, K., Ganapathy, M., Reid, L., Riley, K., Sheeler, D., Siegel, A., Weide, K.: Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. Parallel Comput. **35**(10–11), 512–522 (2009)
5. Dubey, A., Reid, L., Fisher, R.: Introduction to FLASH 3.0, with application to supersonic turbulence. In: Physica Scripta T132, : Topical Issue on Turbulent Mixing and Beyond, Results of a Conference at ICTP. Trieste, Italy, August (2008)
6. Dubey, A., Mohapatra, P., Weide, K.: Fault tolerance using lower fidelity data in adaptive mesh applications. In: Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale, pp. 3–10. ACM (2013). http://doi.acm.org/10.1145/2465813.2465817

7. Fujita, H., Dun, N., Rubenstein, Z.A., Chien, A.A.: Log-structured global array for efficient multi-version snapshots. In: IEEE CCGrid 2015 (2015)
8. Lu, G., Zheng, Z., Chien, A.A.: When is multi-version checkpointing needed? In: Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale, FTXS 2013. ACM (2013)
9. Moody, A., Bronevetsky, G., Mohror, K., De Supinski, B.R.: Design, modeling,and evaluation of a scalable multi-level checkpointing system. In: SC 2010 (2010)
10. NCSA: Heirarchical Data Format 5 (2008). http://hdf.ncsa.uiuc.edu/HDF5/
11. Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., Apr, E.: Advances, applications and performance of the global arrays shared memory programming toolkit. IJHPCA **20**(2), 203–231 (2006)
12. Sato, K., Mohror, K., Moody, A., Gamblin, T., de Supinski, B., Maruyama, N., Matsuoka, S.: Design and modeling of a non-blocking checkpointing system. In: SC 2012 (2012)
13. Shet, A.G., Elwasif, W.R., Foley, S.S., Park, B.H., Bernholdt, D.E., Bramley, R.: Strategies for fault tolerance in multicomponent applications. Procedia Comput. Sci. **4**, 2287–2296 (2011)
14. Shi, X., Pazat, J., Rodriguez, E., Jin, H., Jiang, H.: Adapting grid applications to safety using fault-tolerant methods: design, implementation and evaluations. Future Gener. Comput. Syst. **26**(2), 236–244 (2010)
15. Walters, J., Chaudhary, V.: Replication-based fault tolerance for MPI applications. IEEE Trans. Parallel Distrib. Syst. **20**(7), 997–1010 (2009)
16. Zheng, Z., Chien, A.A., Teranishi, K.: Fault tolerance in an inner-outer solver: a gvr-enabled case study. In: 11th International Meeting High Performance Computing for Computational Science, VECPAR 2014 (2014)