Improving Resilience of Scientific Software through a Domain-Specific Approach

I.Z. Reguly[a,*], G.R. Mudalige[b], M.B. Giles[c], S. Maheswaran[d]

[a]*Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Budapest, Hungary*
[b]*Department of Computer Science, University of Warwick, UK*
[c]*Maths Institute, University of Oxford, UK*
[d]*AWE Plc, Aldermaston, UK*

Abstract

In this paper we present research on improving the resilience of the execution of scientific software, an increasingly important concern in High Performance Computing (HPC). We build on an existing high-level abstraction framework, the Oxford Parallel library for Structured meshes (OPS), developed for the solution of multi-block structured mesh-based applications, and implement an algorithm in the library to carry out checkpointing automatically, without the intervention of the user. The target applications are a hydrodynamics benchmark application from the Mantevo Suite, CloverLeaf 3D, the sparse linear solver proxy application TeaLeaf, and the OpenSBLI compressible Navier-Stokes direct numerical simulation (DNS) solver.

We present (1) the basic algorithm that OPS relies on to determine the optimal checkpoint in terms of size and location, (2) improvements that supply additional information to improve the decision, (3) techniques that reduce the cost of writing the checkpoints to non-volatile storage, (4) a performance analysis of the developed techniques on a single workstation and on several

[*]© British Crown Owned Copyright 2018/AWE. Published with permission of the Controller of Her Britannic Majesty's Stationery Office. "This document is of United Kingdom origin and contains proprietary information which is the property of the Secretary of State for Defence. It is furnished in confidence and may not be copied, used or disclosed in whole or in part without prior written consent of Defence Intellectual Property Rights DGDCDIPR-PL - Ministry of Defence, Abbey Wood, Bristol, BS34 8JH, England."

[*]Corresponding author

Email address: `reguly.istvan@itk.ppke.hu` (I.Z. Reguly)

supercomputers, including ORNL's Titan.

Our results demonstrate the utility of the high-level abstractions approach in automating the checkpointing process and show that performance is comparable to, or better than the reference in all cases.

Keywords: Domain Specific Language, High Performance Computing, Checkpointing, Resilience, Parallel I/O

1. Introduction

Ever since the end of Dennard scaling [1], the principal source of performance improvement has been from a continuous increase in parallelism. This, combined with the slow-down of the process shrinkage and the push toward exascale, has resulted in the ever increasing scale of High Performance Computing (HPC) systems. However, scale is a major threat to reliability [2, 3, 4]; even if the Mean Time Between Failures (MTBF) for a single machine is on the order of years, for a system with tens of thousands of nodes (such as TaihuLight [5] or the planned Aurora system [6]), MTBF could drop below a day, well within the runtime of a large-scale scientific simulation.

One of the key causes for interruptions is hardware failure - most commonly components fail, bringing down entire nodes. Cosmic radiation can also cause errors in memory and execution units, though memory is commonly protected with ECC technology [7]. The second key factor causing errors is software: as we are using more and more complex software, developed in a loosely-coupled way, it is increasingly likely that bugs will cause interruptions.

Two of the biggest US peta-scale systems, Blue Waters and Titan have shown that interruptions and outages are a daily occurrence: Blue Waters was reported [8] to have a failure every 4.2 hours - some of which wouldn't interrupt running jobs - a node failure every 6.7 hours, and a whole system failure every 160 hours. A separate report on ORNL's Titan [9] cites a failure every 28 hours.

Perhaps the most common way to address resiliency in scientific software is checkpointing; periodically saving the state of the simulation, and in the event of a failure restoring the previous state. The application of these checkpointing methods to existing software can be tedious and/or expensive in terms of coding effort and the efficiency of the checkpoint creation process itself - depending on how high- or low-level the chosen method is.

There are two key classes of checkpointing approaches; system-level, and application-level. For system-level methods, most commonly the operating system is extended to periodically serialise the entire state of the running process (data as well as stack), which makes it entirely transparent to the application, but has the drawback of having to save a lot of data, and having to restart with the same number of processes. For application level approaches, currently it is the responsibility of application programmer to determine what needs to be saved to enable restoring at the checkpoint: indeed, this can be much smaller than the total amount of memory used, however in complex software it can be difficult to determine the minimal state space, and disruptive to implement such application-level checkpointing methods. It is also a challenge to restore the function calls stack.

We carry out this research in the context of domain specific languages which have shown excellent results in generating highly optimized implementations from high-level abstractions, thereby reducing the development effort from domain scientists [10, 11, 12]. In this work we report on research using the OPS [13, 14, 15, 16] framework, an embedded domain specific language (EDSL), for implementing checkpointing.

We demonstrate how these techniques allow to create an application-level checkpointing mechanism that is almost completely transparent to the user but also deliver near-optimal performance in terms of the impact of checkpointing on the runtime of the simulation. Specifically, we make the following contributions:

1. We present the basic concepts and algorithms behind the automated checkpointing and recovery in OPS.
2. We introduce techniques that allow further improvements and more control over the checkpointing process.

3. We integrate methods to store checkpoints and minimise the impact of the process on the "useful" computations by making it non-blocking.
4. We analyse the performance of various algorithms and methods of storing the checkpoints on a single workstation, a small-scale Intel cluster, a Cray XC30 (ARCHER) system and on Titan, a Cray XK-7 system.
5. We discuss how some of these techniques could be transferred to existing software that does not use OPS.

The rest of this paper is organised as follows: Section 2 presents the related work, Section 3 briefly discusses the OPS framework, Section 4 presents the algorithmic ideas behind the checkpointing methods in OPS, Section 5 describes the implementation techniques, Section 6 discusses file I/O options, Section 7 presents the performance results and their analysis, Section 8 discusses how to use some of these ideas without OPS, and finally Section 9 draws conclusions.

2. Related Work

There is a wide range of options for improving the resiliency of software running on large machines [2, 4, 17], but the most commonly used method is the periodic checkpointing of the application state to files; in the case of a failure the application is restarted from the last checkpoint. Significant research has been carried out in predicting failures in large-scale systems [18, 19], to reduce the overheads. To determine the optimal checkpointing interval, Young [20] developed the first model ($\sqrt{2 * MTBF * T_{checkpoint}}$). For current large-scale systems, this would mean checkpointing on the order of every hour, and as we show later in this paper, this takes considerable time, significantly increasing time-to-solution. It is therefore crucial to minimise the time taken to create a checkpoint.

Low-level, or system-level approaches to checkpointing use operating system extensions, compiler analysis [21, 22, 23], data compression and aggregation [24] to automate the creation of process checkpoints. However these approaches still face the challenge that they do not understand the semantics of data without input from the application programmer. While BLCR [25] and similar methods are attractive because they require no (or very little) changes in the application codes, because they operate on a kernel/OS level, they need to save the entire state of the application, and the same number of processes are needed to restart the application. Other works aim to improve the scheduling of jobs, and set their checkpointing intervals to maximise efficiency [26]. There are large-scale efforts to provide an OS, and system-level approaches, such as XtreemOS and its Grid Checkpointing Service [27].

There is also a considerable amount of work on application-level checkpointing approaches - these are perhaps the most developed and the most commonly used in production environments [28, 30] . GVR [29] uses versioned distributed arrays [31, 32], and is integrated into several large applications such as Chombo - it still requires significant changes to the user code (thousands of lines for Chombo), and the programmer has to determine what data to save. Additionally, extra effort has to be spent to save and restore the call stack of the application.

There is a large amount of work looking to improve the speed of saving data using multi-level checkpointing: data is saved at multiple levels of storage technology in the system. These include in-memory [33], on node-local non-volatile memory (such as SSDs), to protect against the failure of a single process, then on NVRAMs of other nodes to protect against the failure of a whole node, and finally on the parallel file system, to protect against whole-system outage. Technologies such as FTI [34] and SCR [35] have demonstrated the utility of the multi-level approach, and Charm++ has also integrated this [36].

We expand on the algorithm briefly presented in its key points in our previous work [13], which at the time was not detailed, implemented, or evaluated. Our work contributes to the state of the art by introducing an application-level checkpointing approach that requires significantly fewer changes to existing code than other approaches. Indeed, existing applications using OPS only need a single additional API call to enable the creation of checkpoints as well as the automated recovery to a given checkpoint, including the restoration of the call stack. Furthermore, based on feedback from OPS, a user can improve on this in various ways through additional API calls that reduce the checkpoint size and/or makes the restore process faster. We complement this by integrating techniques into the OPS library akin to SCR and FTI that mitigate the overhead of checkpointing.

3. The OPS Embedded DSL

The Oxford Parallel library for Structured grid computations (OPS) is an Embedded Domain Specific language embedded in C/C++ and Fortran together with supporting libraries and code generators, targeting the development of computations on multi-block structured meshes. The abstraction consists of four principal components:

1. Blocks: a collection of structured grid blocks. These have a dimensionality but no size.
2. Datasets: data defined on blocks, with explicit size.
3. Halos: description of the interface between datasets defined on different blocks.
4. Computations: description of an elemental operation applied to grid points, accessing datasets on a given block.

Given blocks, datasets and halos, an unstructured collection of structured meshes can be fully described. The principal assumption of the OPS abstraction is that the order in which elemental operations are applied to individual grid points during a computation may not change the results, within machine precision (OPS does not enforce bitwise reproducibility). This is the key assumption that enables OPS to parallelise execution using a variety of programming techniques.

From a programming perspective, OPS looks like a traditional software library, with a number of Application Programming Interface (API) calls that facilitate the definition of blocks, datasets and halos, as well as the definition of computations - the details of the API are described in [13, 16]. From the user point of view, using OPS is like programming a traditional single-threaded sequential application, which makes development and testing intuitive - data and computations are defined at a high level, making the resulting code easy to read and maintain.

```
for( int j = 12; j < 50; j++ ) {
  for( int i = 12; i < 50; i++) {
    a[j][i] = b[j][i] + b[j+1][i]+b[j][i+1];
  }
}
```

Listing 1: A classical 2D stencil computation

```
void calc(double *a, const double *b) {
  a[OPS_ACC0(0,0)] = b[OPS_ACC1(0,0)] + b[OPS_ACC1(0,1)]
                   + b[OPS_ACC1(1,0)];
}
...
int range[4] = {12,50,12,50};
ops_par_loop(calc, block, 2, range,
  ops_arg_dat(a,S2D_0,"double",OPS_WRITE),
  ops_arg_dat(b,S2D_1,"double",OPS_READ));
```

Listing 2: A parallel loop defined using the OPS API [13]

Take for example a classic nested loop performing a stencil operation as shown in Listing 1. The description of this operation using the OPS API is shown in Listing 2; it defines an iteration over the grid points specified by `range`, executing the user kernel `calc` on each, passing pointers to datasets `a` and `b`, `a` is written using a one-point stencil and `b` is read, using a three point stencil - these stencils are described by the data structures `S2D_0` and `S2D_1` respectively, which are defined by the user using a `ops_decl_stencil`. The `OPS_ACC` macros are used to compute the index offsets required to access the different stencil points, these are set up by OPS automatically.

An application implemented once using the above API can be immediately compiled using a common C++ compiler (such as GNU g++ or Intel icpc), and tested for accuracy and correctness - this is facilitated by a header file that provides a single-threaded implementation of the parallel loops and the halo exchanges. Code generation is then used to create specialised parallel implementations of the computational loops for different parallel programming

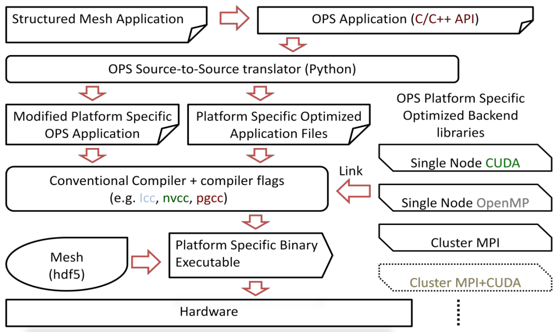

Structured Mesh Application
OPS Application (C/C++ API)
OPS Source-to-Source translator (Python)
Modified Platform Specific OPS Application
Platform Specific Optimized Application Files
OPS Platform Specific Optimized Backend libraries
Single Node CUDA
Single Node OpenMP
Cluster MPI
Cluster MPI+CUDA
Link
Conventional Compiler + compiler flags (e.g. Icc, nvcc, pgcc)
Mesh (hdf5)
Platform Specific Binary Executable
Hardware

Figure 1: OPS code generation and build process

models and hardware, such as OpenMP, CUDA, OpenACC and others. The structure of the OPS library is shown in Figure 1.

The high-level application code is built to rely entirely on the OPS API to carry out computations and to access data; after an initial setup phase where data is passed to OPS using either existing pointers or HDF5 [37] files, OPS takes ownership of all data, and it may only be accessed via API calls. This enables OPS to make transformations to data structures that facilitate efficient parallel execution.

This abstraction and API can be viewed as an instantiation of the AEcute (Access-Execute descriptor) programming model [38] that separates the abstract definition of a computation from how it is executed and how it accesses data; this in turn gives OPS the opportunity to apply powerful optimisations and re-organise execution.

4. Checkpointing in OPS

Building on the abstraction described above, the main contribution of our paper is to detail an automated checkpointing method in OPS, that does not require alterations to existing user code, except for a single API call after set-up.

OPS takes ownership of all data, and that data “leaving” the realm of OPS only happens through API calls (such as reductions, the results of which might be used to alter control flow). This makes it possible for OPS to keep track of what, when and how data is modified, and therefore to reason about the state space. Building on a transactional point of view [39], the fundamental observation behind our checkpointing strategy is that if a dataset is overwritten immediately after the checkpoint, then that dataset does not need to be included in the checkpoint. The question therefore becomes: when to create a checkpoint, and out of all datasets defined, which ones to save.

The second key requirement comes from being able to *restore* the state of the application; not only the values of data arrays, but also call stack and any user-defined state that represents where during the execution the application was at the time of the checkpoint (e.g. time iteration index). While in most application-level checkpointing approaches this requires custom code, this can be entirely automated in OPS by re-playing the execution of the application up to the checkpoint, without actually performing any computations or communications. This can be done by saving the results of, for example, reductions that return data to the user, which then may be used to determine the high-level control flow of the application - during the recovery process, these values are returned, ensuring the same control flow.

The execution of an application from an OPS point of view essentially comes down to a sequence of parallel loop calls, each of which read certain datasets and write others. However, any given loop usually only accesses a small subset of all datasets, therefore reasoning about the state space at any particular parallel loop, given the data it accesses, is not sufficient; this leads to the introduction of “checkpointing regions”: the beginning of the region is the location of the checkpoint in the classical sense, but the actual process spans several subsequent parallel loops.

In practise, the only modification to the user code is the addition of either a runtime argument or a call to an OPS API during initialisation that specifies the checkpointing frequency. During execution, OPS will save the value of global reductions, and when a timer triggers checkpointing, it will automatically find the next entry into a “checkpointing region” and execute the algorithm below, saving data to a HDF5 file. The pseudocode for this process is given in Algorithm 1, and shown as a diagram in Figure 2: `ops_par_loop` API calls call `process_loop` before executing, and API calls that query data (such as getting the result of a reduction) call `process_query` before returning their results.

A high-level description of the algorithm, referencing lines in Algorithm 1 is as follows:

1. Line 1: If a dataset was never modified (as might be the case with e.g. mesh coordinates), then it is not saved at all.
2. Line 32: The results of global reductions in loops are saved for every occurrence of the loop because data returned after a loop is out of the hands of OPS, and may be used for control decisions.
3. Lines 10-14: When checkpoint creation is triggered, then enter a "checkpointing region" upon reaching the first parallel loop, and before executing that loop:
   (a) Lines 15-17: Include datasets accessed by the loop that are not write-only.
   (b) Lines 18-20: Do not include datasets that are write-only in the loop from the checkpoint.
4. When already in a "checkpointing region" (previous point), start executing subsequent loops to determine whether datasets that were not yet saved nor dropped (i.e. are flagged) would have to be saved:
   (a) Lines 15-20: If a flagged dataset is encountered, save it if it's not write-only, otherwise do not include it and remove the flag.
   (b) Lines 21-26: If a flagged dataset is not encountered within a reasonable timeframe allocated for the "checkpointing region", then save it.

In the event of a failure, the application needs to be restarted, and if a checkpoint file is found then "restore mode" is enabled, during which calls to `ops_par_loop` do not carry out any computations, and data query type API calls return the saved values. Once the location of the last checkpoint is reached, the state space is restored from the HDF5 file, "restore mode" ends, and execution returns to normal. The pseudocode for restoring is shown in Algorithm 2.

Algorithm 1 The checkpointing algorithm

```
1: Initially: $ever\_written[0..datasets] = 0$, $seen[0..datasets] = 0$, $in\_region = 0$, $OPS\_red =$
   $empty$, $loop\_index = 0$.
2: function PROCESS_LOOP(index, arguments)
3:     $loop\_index + +$
4:     for all args written do
5:         $ever\_written[dataset] = 1$
6:     end for
7:     if $restore\_mode$ then PROCESS_LOOP_RESTORE(index, arguments)
8:         return
9:     end if
10:    if checkpoint timeout then
11:        if $in\_region == 0$ then
12:            Open checkpoint file, save $loop\_index$ and contents of $OPS\_red$
13:            $in\_region = 1$
14:        end if
15:        for all args read, $seen[dataset] == 0, ever\_written[dataset] == 1$ do
16:            Save dataset to checkpoint, $seen[dataset] = 1$
17:        end for
18:        for all args written, $seen[dataset] == 0$ do
19:            $seen[dataset] = 1$
20:        end for
21:        if checkpoint region timeout or $seen[0..datasets] == 1$ then
22:            for all datasets $seen[dataset] == 0$ and $ever\_written[dataset] == 1$ do
23:                Save dataset to checkpoint
24:            end for
25:            Close checkpoint, $seen[0..datasets] = 0$, $in\_region = 0$
26:        end if
27:    end if
28: end function
29: function PROCESS_QUERY(data returned)
30:    if $restore\_mode$ then PROCESS_QUERY_RESTORE(index, arguments)
31:    else
32:        Save data returned to $OPS\_red$
33:    end if
34: end function
```

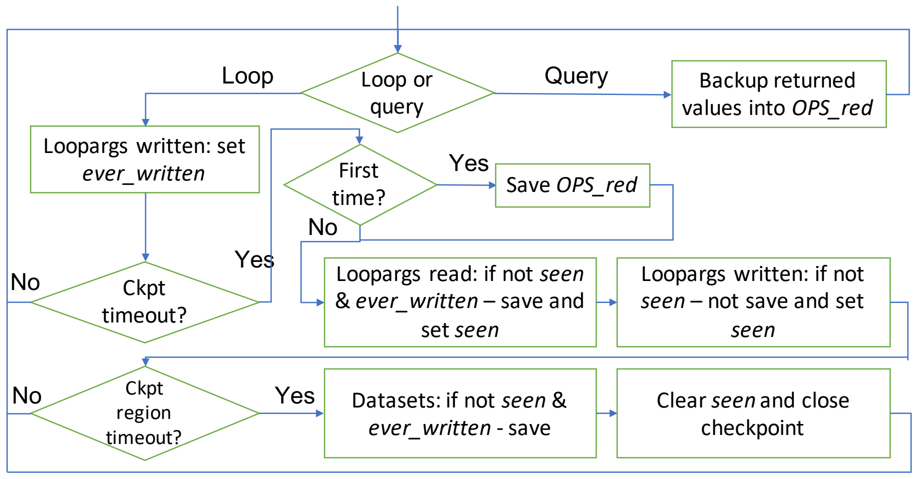

Loop
Loop or query
Query
Backup returned values into *OPS_red*
Loopargs written: set *ever_written*
First time?
Yes
Save *OPS_red*
No
Yes
No
Ckpt timeout?
Loopargs read: if not *seen* & *ever_written* – save and set *seen*
Loopargs written: if not *seen* – not save and set *seen*
No
Ckpt region timeout?
Yes
Datasets: if not *seen* & *ever_written* - save
Clear *seen* and close checkpoint

Figure 2: Diagram for checkpoint creation. Entry at the top, for any API call. Loop: `ops_par_loops`, Query: any OPS API that returns values to userspace, typically reductions. *seen*: flags indicating whether a dataset was encountered in the checkpointing region. *ever_written*: flags that indicate whether the dataset was written during the execution of the application. *OPS_red*: memory allocated for storing reduction data.

One of the key challenges is deciding where exactly to enter the "checkpointing region" so that the state space that has to be saved is minimal; entering it at the first loop that has a write-only dataset may only be locally optimal.

As discussed, it is easy to find a locally optimal checkpoint location, however in order to globally minimise the amount of data that needs to be saved, it is necessary to find a regularly occurring point during execution where entering checkpointing mode results in the least amount of data saved.

5. Implementation

The implementation of checkpointing in OPS closely follows the algorithmic description, with several additions that can help improve performance and the size of the checkpoint given further information from the user.

The key OPS API call to enable checkpointing is `ops_checkpointing_init( file_path, interval, options )`. The argument `file_path` can point either

Algorithm 2 THE RESTORE FROM CHECKPOINT ALGORITHM

```
1: Initially: $restore\_mode = 1$, $loop\_index$ (renamed $restore\_index$) and $OPS\_red$ read from
   checkpoint file
2: function PROCESS_LOOP_RESTORE(index, arguments)
3:     if $loop\_index == restore\_index$ then
4:         restore all datasets in checkpoint
5:         $restore\_mode = 0$
6:     end if
7:     skip computation of this loop
8: end function
9: function PROCESS_QUERY_RESTORE(data returned)
10:    Return data from $OPS\_red$
11: end function
```

to a parallel file system or to node-local storage. The time period between checkpoints can be defined with `interval`, and the timer is subsequently managed by OPS. Options are described later in this section. OPS does not suggest a checkpointing time interval, as that is a highly machine specific parameter.

By placing this single API call in the code, and setting the runtime flag, checkpointing will be fully automated by OPS.

To coordinate processes in order to make sure entering the "checkpointing region" is a collective operation, we can piggyback on global reductions issued by the user, and if any process timed out, we begin checkpointing.

5.1. Initialisation phase

Most complex scientific simulations start with an initialisation phase, where several datasets are populated that are never modified later, for example auxiliary arrays like coordinates. This is indeed the case for CloverLeaf, TeaLeaf and OpenSBLI. We therefore introduce a simple extension to the above checkpointing model: the initialisation phase. This phase is completely ignored by the checkpointing algorithm; in backup mode datasets written in this phase are considered "never modified" and dropped from any future checkpoint (the *ever_written* flag array is reset at the end of the initialisation phase). In restore mode, this initialisation is re-run, in order to re-populate those datasets, and only afterwards does execution skip to the last checkpoint. OPS adds an API call, `ops_checkpointing_initphase_done`, that the user can place in the code to mark the end of the initialisation phase.

5.2. Checkpointing location

During execution, before the checkpointing process can begin, OPS needs to find a recurring point during execution where checkpointing will be initiated, and the amount of state to be saved is minimised. By default, finding this location is entirely up to the OPS runtime, but with additional APIs, the user can specify this location as well.

If the user does not specify the location for checkpoints, OPS will use a simple strategy that calculates the amount of data that would be saved if the checkpointing region were to be entered at any given loop. This is done by building a table with statistics for each loop, along with the frequency of the loop occurring and the variance in the amount of data to be saved between occurrences. Statistics are gathered up to the very first time the timeout triggers a checkpoint, at which point the loop that occurs sufficiently frequently, and would save the least amount of data, is selected, and checkpointing will start the next time that parallel loop is encountered. The details of this algorithm are given in `ops_checkpointing_strategy.c` in [16]. This decision can be reported by setting the `-OPS_DIAGS=3` runtime flag and integrated into the user code, by using one of the API calls that specify the checkpoint location.

It is possible for the user to explicitly define the location of the checkpoint: this is particularly easy to do given the reports from the automated checkpointing location finding mechanism in OPS. This method still pushes the responsibility to determine which datasets to save onto OPS. The user then has an option to include some user-space data into the checkpoint (such as current timestep) - this can be used in restore mode to fast-forward to the checkpoint (“FastFW” optimisation), and avoids having to save the results of all reductions. Then, it is also possible to explicitly specify the list of datasets to be included in the checkpoint (“Datlist” optimisation). All of these options help to give the run-time more information and ultimately to reduce the size of the checkpoint. Finally, it is also possible to manually trigger the creating of the checkpoint, instead of relying on the built-in timers.

6. Implementation of file I/O

The checkpointing functionality of OPS currently relies on the HDF5 library to read and write checkpoint files. HDF5 supports MPI I/O to write a single checkpoint file onto the parallel file system, but it is also possible for each process to create its own file: we implement both options. As we will see later, the MPI I/O version has some performance issues, but allows to re-start with a different number of processes, as re-partitioning the data from a single source is simple. Creating files for each process has the drawback of having to re-start (in the event of a failure) with the same number of processes, but it has performance advantages.

While MPI I/O requires writing to the parallel file system directly, the per-process checkpointing method enables a multi-level checkpointing approach [35]. At the first level, OPS supports writing checkpoints to node-local non-volatile memory, such as SSDs - this protects against the failure of a single process. At the second level, OPS supports the replication of checkpoints on different nodes (by MPI processes sending their checkpoint data to a neighbouring process): this is possible both with in-memory checkpointing and writing the checkpoints to files - this protects against the failure of a complete node. The third level of checkpointing is the MPI I/O approach itself, this protects against a full system outage.

By default, when a checkpoint is triggered, OPS opens a new HDF5 file (either collectively or per-process), and saves all required state (current loop, reduction data, user-space payload), and writes the datasets to the file when the decision is made to save them. However, this is a blocking operation, which may be expensive for larger problem sizes or when using a parallel file system. To avoid this problem, OPS implements two further strategies: in-memory checkpointing and thread-offload checkpointing.

In-memory checkpointing replicates data in the memory space of the process itself, which is the cheapest way of saving data, but of course it is saved into volatile memory. When a process/node fails and the scheduler terminates the job, the processes receive `SIGINT`, which is caught by OPS, and only then is data written to files. In order to avoid any loss of data, it is important that in-memory checkpointing is combined with the replication of checkpoint records, a second-level checkpointing method, automatically supported by OPS. While the amount of memory available in a compute node is often a concern on clusters, because of the ability of OPS to determine which datasets can be discarded from the checkpoint, the actual amount of memory required to hold the checkpoint in memory is only a fraction of what is required by all the datasets combined - typically 10-20% in our benchmarks. In-memory checkpointing can be enabled, by specifying the `OPS_CHECKPOINT_INMEMORY` argument to the executable.

Thread-offload checkpointing makes the writing of datasets to disk an asynchronous operation. OPS creates a separate thread that is only responsible for writing the files; the main thread doing the computations just has to make a copy of the data to be saved in memory and hand it off to the background thread, then it can continue on with the execution of the simulation. Thread-offload checkpointing can be enabled by specifying the `OPS_CHECKPOINT_THREAD` argument to the executable. Currently, this option is not compatible with the MPI I/O approach of writing to a single file, due to issues with the thread safety of MPI distributions.

OPS currently requires the re-launch of the application in the case of failure, however, given a resilient MPI distribution that can substitute reserve processes for failed ones, it would be possible for OPS to roll-back intact processes and fast-forward the substitute processes without having to re-launch.

7. Performance Analysis

In this section, we analyse the performance of the checkpointing implementations in OPS on three applications; CloverLeaf 3D [40], OpenSBLI [41] and TeaLeaf [42]. In our analysis, we are mainly interested in the overhead of creating checkpoints, and restoring from them, and also their relative cost compared to the cost of time iterations. Therefore our test runs are fairly short - just long enough to create one checkpoint, several seconds into the execution, once the regular time stepping has begun. The results are averaged across several runs. Error bars show the standard deviation in measurements, which are symmetric, but for readability, we only show the error bar on the positive side.

We first evaluate them on a single node with the different improvements described above, then moving on to direct comparison with a reference implementation of CloverLeaf 3D which uses TyphonIO [43] (referred to as "Ref Checkpoint" in figures). TyphonIO uses collective HDF5 operations to write a single file, which was originally used for visualisation, and saves datasets in single precision. OpenSBLI has no equivalent "reference" implementation, therefore it is evaluated on its own.

The CloverLeaf mini-app involves the solution of the compressible Euler equations, which form a system of four partial differential equations. The equations are statements of the conservation of energy, density and momentum and are solved using a finite volume method on a structured staggered grid. The cell centres hold internal energy and density while nodes hold velocities. The solution involves an explicit Lagrangian step using a predictor/corrector method to update the hydrodynamics, followed by an advective remap that uses a second order Van Leer up-winding scheme. The advective remap step returns the grid to its original position. The original application [40] is written in Fortran and operates on a 3D structured mesh. It is of fixed size in both x and y dimensions.

The application consists of 141 parallel loops, and 45 datasets, out of which 30 are full cardinality (15 are 1D datasets for data such as x coordinates). The overall structure of the main hydro loop is shown in Figure 3.

```
Hydro Loop:
    Timestep
        Ideal Gas
        Viscosity
        Calc dt
    PdV
    Accelerate
    PdV
    Flux calc
    Advection
        Advec cell #1
        Advec mom X-Y-Z
        Advec cell #2
        Advec mom X-Y-Z
        Advec cell #3
        Advec mom X-Y-Z
    Reset Field
    Field Summary
```

Figure 3: Overall structure of the CloverLeaf code

OpenSBLI [41] is a large-scale academic research code, the successor of SBLI [44], which accounts for a significant portion of compute time on several UK national supercomputers. The code is being developed at the University of Southampton, and is used for the solution of compressible Navier-Stokes equations with an application to shock-boundary layer interactions. In this paper, we evaluate a 3D Taylor-Green vortex testcase, which consists of 87 nested loops over the computational grid, and 65 datasets (all full cardinality).

TeaLeaf [42] is a mini-app designed to be representative of matrix-free sparse linear solvers. It solves the heat conduction problem on a sparse, structured mesh and use a five point stencil and cell-centred temperatures to calculate the conduction coefficient. It supports a number of sparse solvers, including Conjugate Gradient, Chebyshev, or Chebyshev polynomially preconditioned CG (PPCG). The application has 31 datasets, out of which 21 are full cardinality, and there are 49 parallel loops distributed across 12 source files.

Considering the behaviour of the checkpointing algorithm for the three applications is very similar, in the interest of brevity, we only show OpenSBLI results in the single node tests, on Archer and on Titan, and TeaLeaf results in the single node tests and on Archer.

7.1. Single node

As CloverLeaf 3D is a representative mini-application, the number of loops is small enough that it is possible to determine the absolutely minimal amount of data that has to be saved at a globally optimal checkpoint location - this is the traditional application-level checkpointing methodology. The optimal location is right before calling `Reset Field`, and the list of datasets that need to be saved is `density1,energy1,xvel1,yvel1,zvel1`, if datasets representing the mesh are not saved. For a 192^3 problem, using double precision, this is $5{*}8{*}196^3$ bytes (accounting for block halo with a depth of 2 on all sides), or 301MB. The full state space, including all the datasets (45) is 1624MB, not counting the MPI halo regions when using MPI, which can be substantial. Thus in the best case we need to save only 18% of all the data used by the simulation.

Table 1: CloverLeaf 3D single node performance (*single precision)

	Time (sec)	Checkpointing time (sec)	Restore time (sec)	OpenMP size (MB)	40 MPI size (MB)
OPS Plain	33.35	-	-	-	-
Ref Plain	37.29	-	-	-	-
Ref Checkpoint	39.98	n/a	n/a	193.77*	193.77*
No info	33.92	0.42	0.086	737.76	938.23
Initphase	33.82	0.33	0.049	488.99	647.57
+FastFW	33.77	0.22	0.048	488.99	647.57
Datlist	33.73	0.13	0.018	312.57	364.86
Datlist+FastFW	33.54	0.13	0.016	312.57	364.86
Triggered	33.72	0.13	0.016	312.57	364.86
Thread offload	33.71	0.027	0.016	312.57	364.86
In-memory	33.60	0.021	0.016	312.57	364.86
MPI I/O	38.78	5.07	0.588	312.57	312.57

Tests were run on an Intel Xeon E5-2650 v3 (Haswell) machine. The machine has 10 cores per socket, running at 2.3 GHz, with HyperThreading on, and there is 64 GB of DDR4 RAM, running CentOS 7.2. The checkpoints are written to a local HDD (I/O speed is 1.6 GB/s as measured with `dd`). Runs utilise all logical cores (40 OpenMP threads or 40 MPI processes), with thread/process binding enabled. All codes were compiled with the Intel Compilers, version 16.0, and use Intel MPI, and HDF5 1.8.4.

CloverLeaf was configured to run for 87 time iterations and a 192^3 mesh. The full runtime is around 33 seconds, thus we set the checkpointing interval to 20 seconds (arbitrary choice, that is well into the execution of the application). Table 1 shows the results - standard deviations are not indicated as they were all less than 1% of the mean. The first two rows show the performance of CloverLeaf 3D without checkpointing using either OPS or the reference implementation [40]. The third row shows performance using TyphonIO in the reference implementation; clearly checkpointing adds a considerable overhead, 2.6 seconds - the size of the resulting file is only 193MB, because it is storing data in single precision. We should note that the reference solution's output is aimed at visualisation, rather than checkpointing, hence the lower precision, but it is storing more datasets (such as coordinates) that do not need to be included in an OPS checkpoint.

CloverLeaf's baseline OPS version performs slightly better than the baseline reference version - this is because some of the code generated loop structures optimise better than the one in the original. The checkpointing variant also performs slightly better: "No info" (OPS) versus "Ref TIO" (reference). The algorithm which tries to locate the optimal checkpointing location does find the global optimum (right before `reset_field`), however without any further information it has to save all the data describing the mesh, as well as boundary regions of datasets that are not written to. Once the initialisation phase option is enabled ("Initphase"), the only full datasets that OPS saves are the five listed above, plus the block halos of the rest - at such a small mesh size and 40 MPI processes those add up to almost double the bare minimum.

Given the feedback from OPS regarding the location of the checkpoint, one can enable the "fast-forward (FastFW)" and "Datlist" optimisations by placing an additional OPS API call to the given location. Fast-forward reduced the amount of reduction data to be saved, improving checkpointing time. When we explicitly list the datasets to be saved ("Datlist"), the halos of other datasets are not saved anymore, and we get very close to the optimum, the only additional data being saved are the MPI halos of the five datasets. As expected, the more information OPS is given, the less data it has to save and the time spent in checkpointing also decreases. Similarly, the cost of restoring to a checkpoint is very small, about an order of magnitude less than creating a checkpoint. However it is clear that at this point, checkpointing onto a workstation's disk is inexpensive - less than the cost of a single iteration of the simulation. In line with this, the thread-offload and the in-memory techniques do not give meaningful improvements over the blocking write to disc.

When MPI I/O is enabled, only one checkpoint file is written for all the processes. This however, involves expensive collective MPI communications, that slow the process down. Performance figures shown in Table 1 show results with the Initphase, FastFw and Datlist optimisations enabled - they are still an order of magnitude slower than the per-process checkpointing approaches.

OpenSBLI is being tested with a 192^3 problem, for 20 time iterations (note that runtimes are also around 20 seconds). Given a 2-wide halo around the boundaries, the total state space is $196^3 * 64 * 8$ bytes, or 3.59 GB. Manual code analysis shows that the minimum state is right before saving the old results during the time-stepping, and includes 5 datasets: `rho, rhoE, rhou0, rhou1, rhou2`. Thus, in the best case we need to save only 7.7% (276 MB) of all the data used by the simulation. The automatic checkpoint location finding algorithm in OPS does find this location, though it determines that the halo regions of other datasets need to be saved as well - the amount of data saved is 473 MB, or 13% of the total state space.

Results are shown in Table 2 - which are consistent with the results seen on CloverLeaf. In OpenSBLI, there is no advantage of using the *Initphase* optimisation. Specifying the list of datasets to be saved, more than halves checkpoint size and time overhead, because halo regions of other datasets are not saved. On a single workstation, the overhead of creating a checkpoint is 0.02-0.16 time iterations, and restoring from a checkpoint is 0.01-0.13 time iterations.

Table 2: OpenSBLI Single node performance

	Time (sec)	Checkpointing time (sec)	Restore time (sec)	OpenMP size (MB)	40 MPI size (MB)
OPS Plain	20.97	-	-	-	-
No info	21.53	0.16	0.13	473.68	780.38
Initphase	21.53	0.16	0.13	473.68	780.38
+FastFW	21.39	0.10	0.076	473.68	780.38
Datlist	21.31	0.022	0.045	287.23	336.74
Datlist+FastFW	21.31	0.022	0.013	287.23	336.74
Triggered	21.30	0.019	0.011	287.23	336.74
Thread offload	21.32	0.026	0.011	287.23	336.74
In-memory	21.31	0.022	0.011	287.23	336.74
MPI I/O	27.1	5.57	0.77	287.23	287.23

TeaLeaf is being tested with a 2000^2 problem, for 10 time iterations. The total state space is 628 MB. Manual code analysis shows that the minimum state is right before starting the linear solver during the time-stepping, and includes only one dataset: `energy1`. Thus, in the best case we need to save only 5.1% (32 MB) of all the data used by the simulation. The automatic checkpoint location finding algorithm in OPS does find this location, though it determines that a large number of datasets that describe the mesh, as well as the halo regions of other datasets need to be saved as well - the amount of data saved is 185 MB, or 29% of the total state space.

Results are shown in Table 3 - which are consistent with the results seen on CloverLeaf and OpenSBLI. In TeaLeaf, there is a significant advantage of using the *Initphase* optimisation, which brings the checkpoint sizes down to 32 MB. Specifying the list of datasets to be saved only improves checkpoint sizes significantly over MPI. On a single workstation, the overhead of creating a checkpoint is 0.02-0.16 time iterations, and restoring from a checkpoint is 0.01-0.12 time iterations.

Table 3: TeaLeaf Single node performance

	Time (sec)	Checkpointing time (sec)	Restore time (sec)	OpenMP size (MB)	40 MPI size (MB)
OPS Plain	78.27	-	-	-	-
No info	79.26	0.99	0.02	185	414
Initphase	78.55	0.27	0.01	32.6	40.8
+FastFW	78.5	0.23	0.01	32.4	33.7
Datlist	78.5	0.23	0.01	32.2	32.7
Datlist+FastFW	78.39	0.12	0.01	32.1	32.7
Triggered	78.4	0.13	0.01	32.1	32.7
Thread offload	78.4	0.13	0.01	32.1	32.7
In-memory	78.3	0.03	0.01	32.1	32.7
MPI I/O	80.4	2.13	0.32	32.1	32.1

7.2. Scaling on Arcus-b

Arcus-b is a small compute cluster at the University of Oxford, with Intel E5-2640 v3 (Haswell) CPUs (2.6 GHz, 8 cores per socket, HyperThreading disabled). The nodes have 64 GBs of RAM, and run CentOS 6. The system has node-local hard disks as well as a large shared Panasas parallel file system, therefore it is a good candidate for the comparison of multi-level checkpointing strategies. Codes were compiled with the Intel Compilers (16.0), Intel MPI, and use HDF5 1.8.20. Runs were done with 16 processes per node, process binding enabled. We modified the reference implementation and use manually triggered checkpoints in OPS to run for a total of 87 time iterations and create a single checkpoint at step 45.

First, we evaluate strong scaling with a 384^3 mesh; the results are shown in Figure 4(a). Strong scaling without any checkpointing follows the expected exponentially decreasing runtime: the scaling efficiency going from 16 cores to 256 cores is 82%, the reference implementation and OPS are within 10% of each other.

Enabling global checkpointing with MPI I/O and parallel HDF5, shows that there is a significant overhead even at the lowest core count: 5.46 iterations

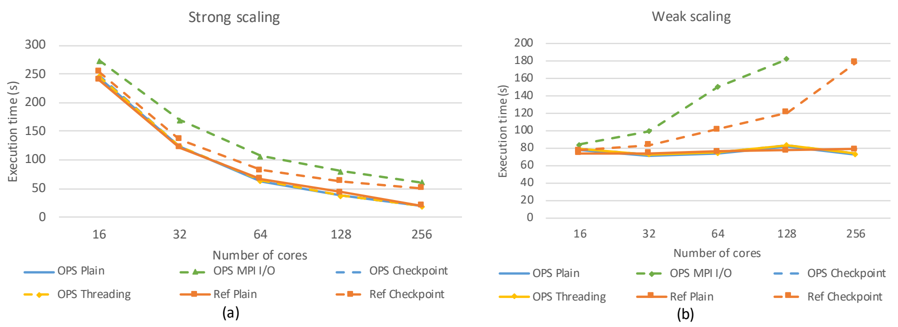


Figure 4: CloverLeaf 3D performance on Arcus-b, strong and weak scaling

or 15 seconds for the reference and 6.8 iterations or 19 seconds for the OPS implementation. This overhead persists when scaling to larger node counts, and proportionally becomes much larger as one would expect when using a parallel file system with a fixed amount of bandwidth: 130 iterations or 30 seconds for the reference and 207 iterations or 43 seconds for the OPS implementation. The overhead significantly increases in absolute terms as well, more than doubling when going from 16 cores to 256; this is due to the parallel file system handling a large number of processes accessing the same file poorly. Clearly, the cost of checkpointing to this parallel file system is considerable, ranging between the cost of 5 time iterations to 136 time iterations.

When enabling the node-local checkpointing feature in OPS, where one file is created for each process onto the hard disk of the node, the overhead dramatically drops: with 16 cores 0.7 iterations or 2.2 seconds, with 256 cores 0.5 iterations or 0.12 seconds, a much more acceptable overhead. We should note that with second-level checkpointing over MPI, the amount of data saved is actually twice as much, because processes mirror their checkpoints with the help of processes on other nodes.

Weak scaling is also evaluated on arcus-b with a 256^3 mesh per node, the results are shown in Figure 4(b). Once again performance with checkpointing disabled follows the expected curve: weak scaling efficiency is 95% going from 16 cores to 256. Enabling global checkpointing with parallel HDF5 and MPI I/O results in steeply increasing runtimes; indeed, this is what one would expect

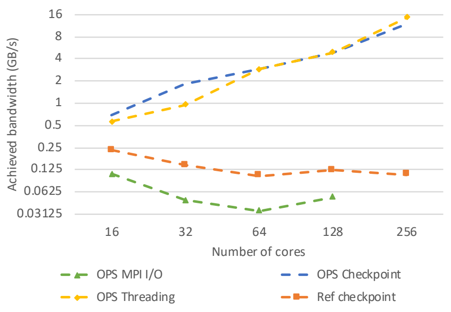


Figure 5: CloverLeaf 3D achieved bandwidth on Arcus-b, weak scaling

from a limited bandwidth parallel file system. The overhead at 16 cores is 3.4 iterations or 2.9 seconds with the reference and 6.8 iterations or 6 seconds with the OPS implementation, and scaling to 256 cores the overhead becomes 114 iterations or 104 seconds with the reference and 114 iterations or 108 seconds with the OPS implementation at 128 cores. Enabling the node-local checkpointing in OPS effectively enables the scaling of I/O bandwidth with the number of nodes, at 16 cores the overhead is 1.1 iterations or just below 1 seconds, and at 256 cores is 1.3 iterations or 0.92 seconds: showing that weak scaling of checkpointing can be achieved with near perfect efficiency. Finally, we show achieved bandwidth in Figure 5 for weak scaling: clearly the performance of the parallel file system is low and stagnating as we use more processes, whereas the node-local checkpoints scale almost perfectly.

7.3. Scaling on ARCHER

ARCHER, a Cray XC30 system, uses a Lustre parallel file system, which we use to perform our tests. It has a nominal bandwidth of 30 GB/s. Nodes have dual-socket E5-2697 v2 (Ivy Bridge) processors, 2.7 GHz, 12 cores each (HyperThreading disabled), and 64 GB of RAM. We use the Cray compilers (8.2.1), MPI (cray-mpich 6.1.1) and HDF5 libraries. Runs use 24 MPI processes per node, with process binding enabled.

Considering that many people are using the supercomputer at the same time, there is considerable noise in the performance of the file system - therefore we ran

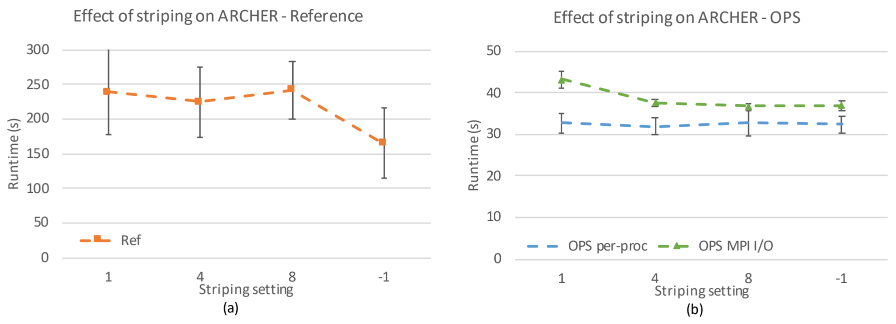


Figure 6: Checkpointing performance with different striping settings on 192 cores of ARCHER, running CloverLeaf 3D

all tests ten times and averaged the results, indicating the standard deviation as well - no such noise is observed when the checkpointing is not used. In order to try and provide a fair comparison of the reference implementation and OPS, we interleave the execution of the five tests (run the ref version, then OPS, then ref again, then OPS again, etc.), hoping that the load on the file system will not vary significantly between two runs.

The machine's Lustre parallel file system permits the user to set the *striping* - changing how many Object Storage Targets (OSTs) any given file is spread over. This is closely related to the performance of writing files to the parallel storage, as particularly for large files, better bandwidth can be achieved at higher stripe counts. There are a total of 50 OSTs in the system, and the user can set the striping setting for any given file or directory to 1, 4 (default), 8, or -1 (which corresponds to the maximum number of OSTs). According to the best practices guide, we have evaluated performance for both MPI I/O checkpoint creation and the per-process checkpointing method in OPS at different striping settings. For 192 processes and a 384^3 mesh, results are shown in Figure 6; performance at higher striping counts is better, but does not increase significantly when going from the default of 4 to 8 or -1, except for the reference version, where there is a significant improvement at -1. All further results are obtained at a -1 setting for both OPS and the reference version.

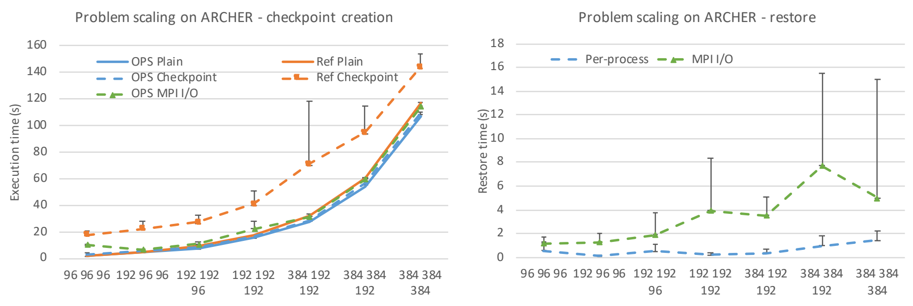


Figure 7: CloverLeaf 3D performance on 2 nodes of ARCHER, with different problem sizes. Error bars show the standard deviation in measurements.

First, we evaluated performance on 2 nodes of the system, a total of 48 cores, on different problem sizes. Results are shown in Figure 7(a), which again demonstrate that the baseline versions are close in performance, with OPS being slightly faster (5-10%) on this system. Enabling checkpointing adds a considerable overhead for the reference implementation, which at this scale is between 17-35 seconds, or 26 iterations at the largest problem and increases to 650 iterations at the smallest. Measurements show that there is a large cost of opening/closing a single file concurrently with TyphonIO's MPI I/O, but the bandwidth of the I/O, once the file is open, is reasonable.

Even though ARCHER, as most Cray systems, has no node-local non-volatile storage that is accessible to the users, we evaluate the per-process checkpointing in OPS. The blocking per-process checkpointing method also adds an overhead, 0.8 seconds (36 iterations) for the smallest 96^3 problem, up to 2.4 seconds (1.9 iterations) for the largest 384^3 problem - at this point faster than the reference solution even with checkpointing turned off. Using MPI I/O for checkpointing in OPS does introduce an overhead compared to the per-process checkpointing method, but much less compared to the TyphonIO version; around 4 time iterations, for larger problem sizes.

We also evaluate the overhead of restoring from a checkpoint, shown in Figure 7(b), which includes everything starting from the launch to resuming normal

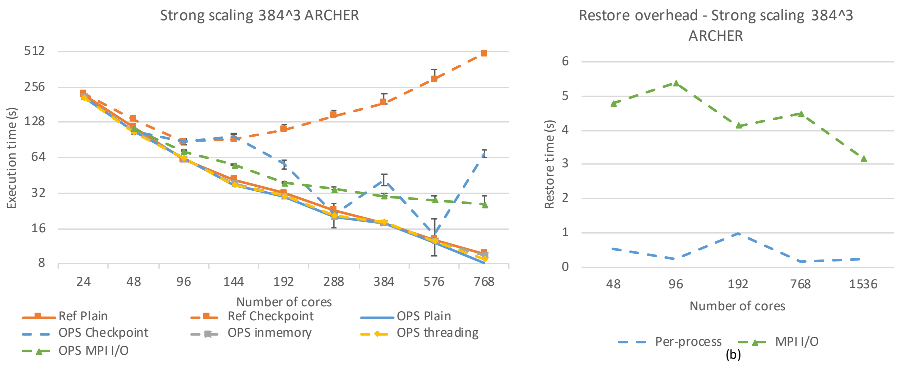


Figure 8: Strong scaling performance with the reference and OPS CloverLeaf 3D implementations

execution. Restoring from a global checkpoint using MPI I/O has significant overheads, just like when writing the checkpoint, and restoring from the per-process checkpoints is once again significantly faster; scaling from 0.048 seconds up to 0.89 seconds, almost an order of magnitude less than the cost of creating the checkpoint.

Figure 8(a) shows the strong scaling performance on the 384^3 problem, with the non-checkpointing version scaling with 95% efficiency. Once checkpointing is enabled, it is clear that execution time increases dramatically, with decreasing performance beyond 96 processes for the reference version. Between 96 and 288 processes, the standard deviation in fairly low, and becomes significant beyond that point - further scaling was not attempted at this time due to the cost of test runs. The cost of checkpointing increases exponentially with the number of nodes for the reference version beyond 96 cores.

Figure 8(a) also shows performance with the various implementations in OPS, using the manual triggering method, creating a checkpoint at iteration 45. Runtimes without checkpointing match the runtimes of the reference version within 5%. The per-process method of checkpointing introduces very little overhead most of the time - however 1 or 2 times out of 10 test-runs execution time jumps up; this is because one of the processes busy-waits to either open or close its checkpoint file - this is an issue with the parallel file system not

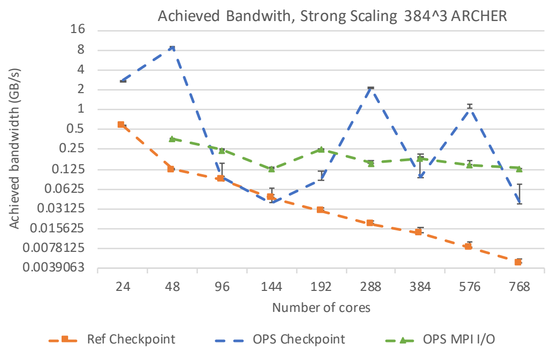


Figure 9: Achieved bandwidth when strong scaling CloverLeaf 3D on ARCHER

handling these requests efficiently. As it can be observed on the figure, this did not occur for the 288 and 576 core runs, where the overhead was negligible. Since no failures actually occurred during these tests, in-memory checkpointing did not write any files to disk, its overhead was also negligible. What is notable however is that when using the thread-offload method to write files to disk this issue of opening or closing files did not occur even once, despite running several tests a further 10 times.

The overhead of restoring to a checkpoint, shown in Figure 8(b), is once again significantly lower than that of creating the checkpoint; between 0.2 and 1 seconds, scaling flatly when using the per-process checkpointing method, and between 3 and 5.5 seconds with MPI I/O with a decreasing trend as the number of processes increases. The time at which the tests were run did introduce some variance in how much time writing files took, but not how frequently the file opening/closing issue occurred. The reason for these issues is unknown at this time.

Figure 9 shows the achieved bandwidth when writing checkpoints on ARCHER; as expected from the timings, the reference version achieves a very low bandwidth, the OPS MPI I/O version slightly higher, and the OPS per-process version the highest, but with considerable fluctuations.

We have also evaluated weak scaling performance on ARCHER with OPS, the performance results are shown in Figure 10. To begin with, it should be

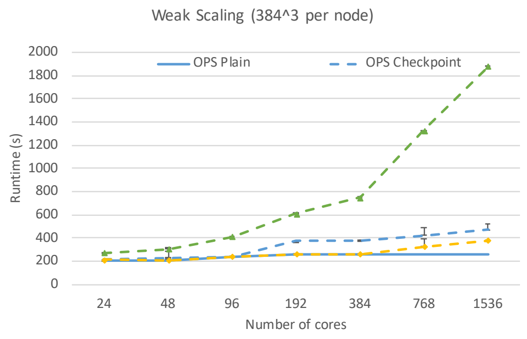
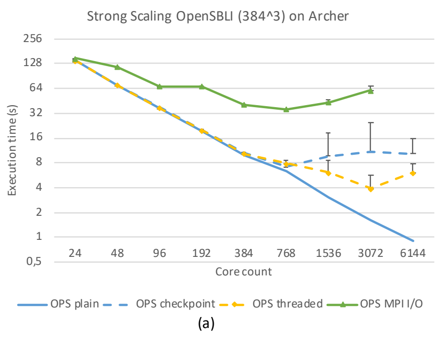


Figure 10: Weak scaling performance with various OPS CloverLeaf 3D implementations on ARCHER

noted that going from 48 to 96 cores and from 96 cores to 192 cores, there is a drop in weak scaling efficiency, even without checkpointing - this is due to an increased cost of communication - which does not increase further going up to 1536 cores however. The overhead when using the blocking version of checkpointing is fairly small up to 96 cores, but increases significantly beyond that point. This is where the thread-offload version improves performance - by hiding the cost of writing to files; up to 384 cores it hides this cost efficiently, but beyond that point the creation of the checkpoint takes a longer time that the time interval between checkpoints - in this setup this is half of the total runtime. When enabling MPI I/O, the cost of checkpointing increases dramatically.

In summary, there seems to be an inherent bottleneck in the way TyphonIO interacts with the ARCHER parallel file system and MPI I/O (based on HDF5 installed in the system). This can be observed as a fairly constant overhead when varying the problem size on the same amount of cores, and this overhead seems to exponentially increase with the increasing amount of cores used. OPS does not demonstrate such an increasing overhead, however when scaling it is impacted by an issue with the filesystem managing file opening and closing operations. With a reasonably large time period between checkpoints, this overhead can be hidden using a thread-offload strategy for writing files.

We have also evaluated the scaling of checkpointing on OpenSBLI; strong



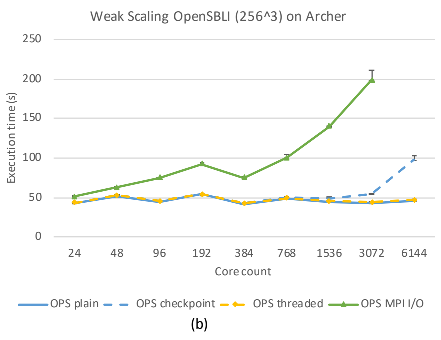


Figure 11: OpenSBLI scaling performance on Archer

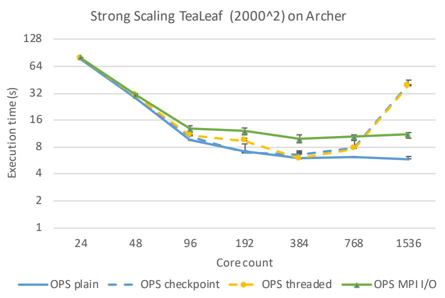


Figure 12: Strong scaling performance with various OPS TeaLeaf implementations on ARCHER

scaling is shown in Figure 11(a), similarly to CloverLeaf, the per-process check-
685 pointing approaches scale with fairly low overhead (1-3 time iteration) up to a point, but beyond 768 cores they start slowing down as well - at 6144 cores, the overhead is 200 and 115 time iterations for the plain and threaded versions. Using MPI I/O to write checkpoints scales significantly worse, even at low node counts it presents a significant overhead; 30 iterations at 48 cores, going up
690 to 740 iterations at 1536 cores. Figure 11(b) shows results for weak scaling at 256^3 problem; once again the per-process approaches scale well, with only a 0.8 iteration overhead with the threaded version even at 6144 cores, whereas the MPI I/O costs go up dramatically.

Additionally, we evaluated strong scaling of checkpointing on TeaLeaf as
695 well - results are shown in Figure 12. Behaviour is qualitatively similar to that

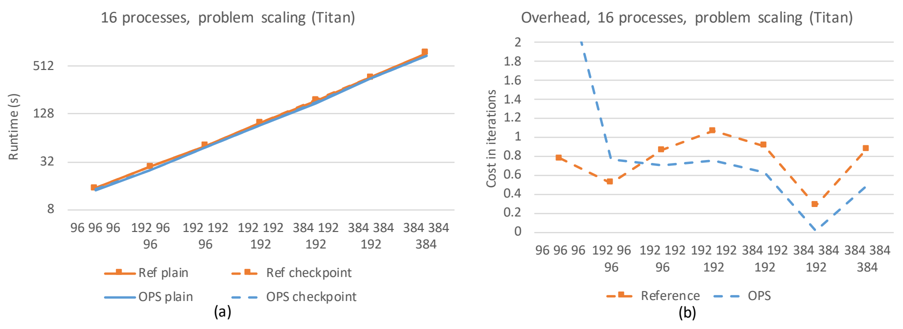


Figure 13: CloverLeaf 3D performance on 1 node of Titan

of CloverLeaf and OpenSBLI - the key difference being that since the size of the checkpoint is significantly smaller (32 MB vs 280-310 MB). Thus at larger scales opening and writing many small files is more costly than collective I/O. Considering that TeaLeaf is a linear solver, we could not do a weak scaling study in the traditional sense, because convergence is affected by problem size.

7.4. Scaling on Titan

Tests very similar to the ones described previously were carried out on ORNL's Titan supercomputer as well, which also uses a Lustre storage [45]. The machine consists of nodes with 16-core Opteron 6274 CPUs, and 32 GB of RAM. Codes were compiled with the Cray compilers (8.2.2), cray-mpich (6.3.0), and were run with 16 MPI processes per node.

The noise in measurements on this system has been insignificant (below 3% of the mean), therefore we do not show the standard deviations. To begin with, we run scaling tests on a single node, using 16 processes - Figure 13 shows the runtime results; unlike on ARCHER there is very little overhead from checkpointing; runtime scales almost perfectly, with OPS being 5-10% faster overall. The overhead of writing a checkpoint is on the order of one or less time iterations, with OPS being 30-50% faster on the larger meshes, but considerably slower on the smallest mesh.

Moving on to large-scale tests, we study CloverLeaf's strong scaling with a 384^3 problem, and weak scaling with 192^3 per node. Figure 14(a) shows strong

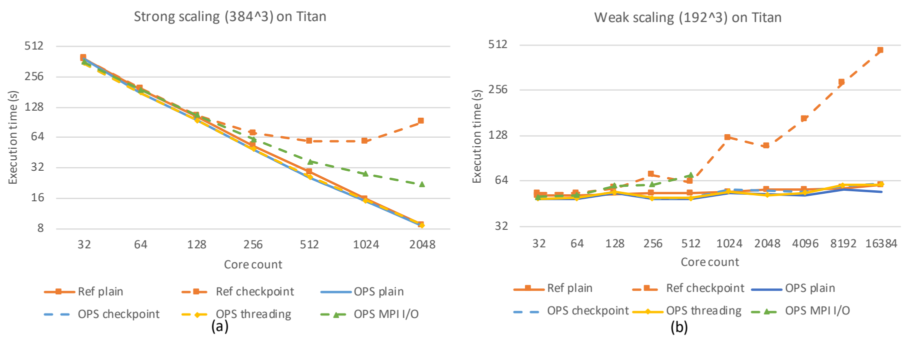


Figure 14: CloverLeaf 3D strong and weak scaling performance on Titan

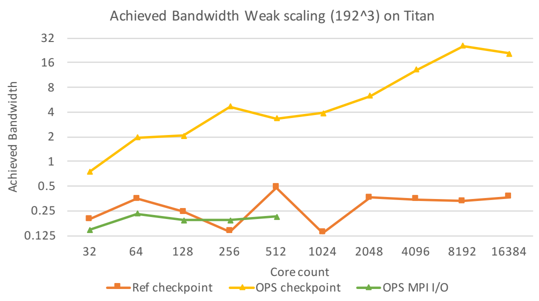


Figure 15: Achieved bandwidth when weak scaling CloverLeaf 3D on Titan

scaling performance with up to 2048 cores. On Titan, the reference implementation with its MPI I/O performs worse than the similar implementation in OPS, but the general trend is the same; at increasing core counts, the overhead of so many processes accessing the same file is significant. At only 32 cores, the overhead of creating a checkpoint with MPI I/O is only 2.3 iterations or 10 seconds, but at 2048 cores it goes to 829 iterations or 82 seconds for the reference implementation and 1.8 iterations or 8 seconds to 133 iterations or 13 seconds for the OPS implementation.

Enabling the per-process checkpointing once again proves beneficial, even though Titan does not have node-local storage either, with an overhead of 0.2 iterations or 1 seconds at 32 cores to 0.7 iterations or 0.07 seconds at 2048 cores. The thread-offload implementation of checkpointing does not give significant improvement compared to the blocking version.

Weak scaling results on CloverLeaf, with 192^3 per node are shown in Figure

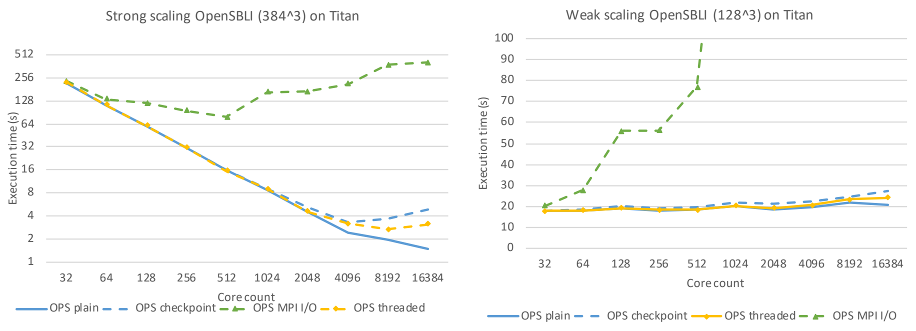


Figure 16: Strong and weak scaling performance with OpenSBLI on Titan

14(b). Once again, the overhead of MPI I/O checkpointing is exponentially increasing with increasing problem size: 2.5 iterations or 1.5 seconds at 32 cores, up to 591 iterations or 408 seconds at 16384 cores for the reference implementations - similar figures apply for the OPS implementation as well - to conserve compute time on Titan, we did not run tests beyond 512 cores for OPS MPI I/O. Switching OPS over to using per-process checkpoints, the overhead becomes very small; 0.7 iterations or 0.4 seconds at 32 cores, up to 11 iterations or 7.2 seconds.

We also evaluate scaling performance of OpenSBLI on Titan. For strong scaling, we use a 384^3 mesh - the results are shown in Figure 16(a). Behaviour is similar to CloverLeaf; MPI I/O scales the worst, flattening and slowing further down above 128 cores. Per-process checkpointing has an overhead of 1.35 time iterations or 0.6 seconds at 1024 cores, which is reduced to 0.54 iterations or 0.24 seconds with the threaded optimisation. By 16384 cores, the time it takes to checkpoint dominates runtime.

Weak scaling a 128^3 mesh on Titan gives results shown in Figure 16(b) - the per-process checkpointing approaches have acceptable overhead; at 16384 cores, the blocking version takes 7 seconds to create a checkpoint (7.15 time iterations), which is reduced to 4.1 seconds with the threaded optimisation (4.2 time iterations). Similarly to CloverLeaf, MPI I/O slows down dramatically.

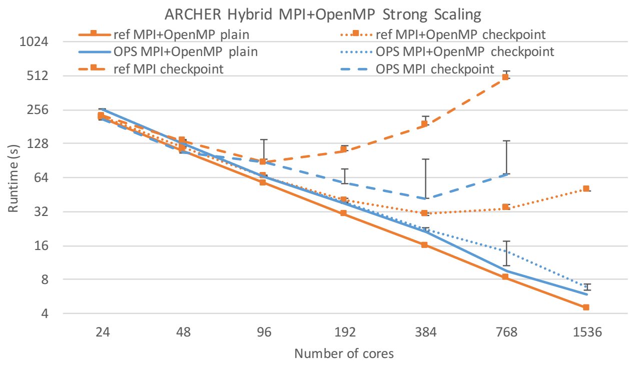


Figure 17: Strong scaling performance of CloverLeaf 3D on Archer with MPI+OpenMP

7.5. Performance with MPI+OpenMP

In previous analysis we have inferred that one of the bottlenecks for checkpointing onto a parallel file system (both for MPI I/O and per-process checkpoints), especially on the ARCHER machine, is the opening and closing of files. To confirm this, we have evaluated performance with a hybrid MPI+OpenMP setup, expecting that because of the reduction in the number of processes, this overhead will decrease. We use the same software setup as described in Section 7.3 (CCE has support for OpenMP 4.5), but instead of 24 MPI processes per node, we launch 2 MPI processes per node (one per socket), and 12 OpenMP threads each.

As Figure 17 shows, when strong scaling, the reference version, with checkpointing enabled scales much better compared to the plain MPI version, although it does exhibit the same behaviour, only at larger process counts. Similarly, the overheads experienced with the per-process checkpointing of OPS are greatly reduced, and the variance is significantly less as well.

8. Without the OPS abstraction

The algorithms and results in this paper so far apply to applications already using the OPS library, which of course limits their scope. While there are good reasons for porting to OPS, revolving around portability, productivity, and performance (as discussed in previous papers), in this section we describe how our work can be generalised to codes that do not use the OPS abstraction.

Some modifications are definitely required to enable or mirror some of the techniques described above. While it is trivial for OPS to skip computations and proceed to the checkpoint when recovering, this would probably require too many modifications in other codes, and a fast-forward approach is more feasible; one has to determine the set of state variables, outside of data arrays, such as iteration count, simulated time, etc. and simply save/restore them. The true challenge therefore lies with determining what datasets need to be included in a checkpoint and which ones may be discarded. Fortunately, there is a cyclical pattern of execution in most scientific simulations, and most temporary datasets are not used across iterations, therefore in many cases the potential locations for the best checkpoint are relatively few.

The key information needed to decide whether a dataset needs to be saved is: (1) whether any part of it was read, and (2) whether if was fully written to (i.e. all the data overwritten). Given this, it is possible to apply the algorithm described in Section 4. If these operations are indicated for each and every loop where a given dataset is accessed, and for every dataset that might potentially need to be saved, then the algorithm can be used directly. If some datasets are excluded, then those will not be saved, unless through a separate method they are marked to be always saved, but excluded from the decision algorithm.

In many cases, it is not feasible to annotate every computational loop. It is however also possible to annotate computational regions - indicating how datasets are being accessed in that region, and making this region an atomic unit of work from the perspective of the checkpointing algorithm - just as in case of OPS a parallel loop is an atomic unit of work. Furthermore, it is also possible to relax the condition of annotating every computational region, and marking the beginning and the end of an encompassing region, within which every computational region is annotated; this will restrict the decision algorithm to reason about the state of datasets within this larger region - assuming that all datasets not accessed or not written in this larger region will need to be saved.

The implementation side is more isolated from the algorithms - HDF5 provides a simple interface to write hierarchical datasets, either on a per-process bases as OPS currently does, or collectively using MPI I/O. Given all the information required about datasets to write them to disk, it is also trivial to make a copy of them in memory to support the thread-offload mechanism and enable non-blocking checkpoint writes. Similarly, in-memory checkpointing can be supported, through the same mechanism. Libraries such as SCR and FTI already support some of these.

Integration of such an approach into widely used programming approaches - such as OpenMP poses several challenges, primarily because of the lack of data ownership, as well as side effects of function calls. However, compilers can determine the type of data access (such as read/write) and thus can help with the annotation of code regions.

9. Conclusions and Future Work

In this paper, we have discussed how, through a high level abstractions approach, it is possible to fully automate near-optimal checkpointing, both in terms of performance and checkpoint size. This is carried out by keeping track of how datasets are accessed, and an algorithm that performs analysis of the state space across a number of computational loops in order to determine which datasets can be discarded from the checkpoint. We have presented the OPS API that enables checkpointing and allows the user to supply further information to aid the automated decision and APIs that allow the user to explicitly indicate the location of the checkpoint as well as what datasets have to be saved.

We have developed a number of implementations that carry out multi-level checkpointing: using MPI I/O onto the parallel file system, per-process checkpoints - saving one checkpoint file for each process, onto node-local storage (optionally replicating data on multiple nodes) or the parallel file system, and in-memory checkpointing. Given the high level algorithm, parts of the implementation can in the future be outsourced to libraries such as SCR [35].

We have evaluated our algorithms on three applications using OPS; CloverLeaf 3D, TeaLeaf, and OpenSBLI. CloverLeaf was also compared to a reference implementation that uses TyphonIO. We have shown that on a single workstation, the overhead of creating a checkpoint is small - on the order of a single iteration of the hydro loop or less. Performance on ARCUS-b, a small Intel cluster, showed the benefits and scalability of node-local checkpointing, and the poor scalability of MPI I/O-based checkpointing approaches. Checkpointing performance on ARCHER showed a significant overhead, especially in the case of the reference implementation; the cost of checkpointing increased exponentially when strong scaling (instead of decreasing exponentially). OPS scaled better, but the poor performance of the parallel file system still affected both strong and weak scaling. We have shown that using the thread-offload method, we can hide much of the overhead of writing files to disk. Finally, scaling on Titan has shown that even with a fast parallel file system, per-process checkpointing can be beneficial, because it avoids the overhead involved in writing to a single file with MPI I/O.

Acknowledgements

The OPS project is funded by the UK Engineering and Physical Sciences Research Council projects EP/K038494/1, EP/K038486/1, EP/K038451/1 and EP/K038567/1 on "Future-proof massively-parallel execution of multi-block applications" and EP/J010553/1 "Software for Emerging Architectures" (ASEArch) project. Cloverleaf development is supported by the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements), E9R00160, and CDK0724, and also the Royal Society through their Industry Fellowship Scheme (IF090020/AM). István Reguly was supported by the János Bólyai Research Scholarship of the Hungarian Academy of Sciences. Project no. PD 124905 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_17 funding scheme. The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work. http://dx.doi.org/10.5281/zenodo.22558. An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research used the resources of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).

References

[1] R. Dennard, F. Gaensslen, H.-N. Yu, V. Leo Rideovt, E. Bassous, A. R. Leblanc, Design of ion-implanted MOSFET's with very small physical dimensions, Solid-State Circuits Society Newsletter, IEEE 12 (1) (2007) 38–50. doi:10.1109/N-SSC.2007.4785543.

[2] J. Dongarra, Fault-tolerance techniques for high-performance computing, in: Y. R. T Herault (Ed.), Fault-Tolerance Techniques for High-Performance Computing, Springer, 2015, pp. 1–66.

[3] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, M. Snir, Toward exascale resilience, Int. J. High Perform. Comput. Appl. 23 (4) (2009) 374–388. doi:10.1177/1094342009347767.
URL http://dx.doi.org/10.1177/1094342009347767

[4] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, M. Snir, Toward exascale resilience: 2014 update, Supercomput. Front. Innov.: Int. J. 1 (1) (2014) 5–28. doi:10.14529/jsfi140101.
URL http://dx.doi.org/10.14529/jsfi140101

[5] Top500, The TaihuLight Supercomputer,

https://www.top500.org/resources/top-systems/sunway-taihulight-national-supercom puting-center-i/ (2018).

[6] The Next Platform, Argonne Hints at Future Architecture of Aurora Exascale System, https://www.nextplatform.com/2018/03/19/argonne-hints-at-future-architecture-of- aurora-exascale-system/ (2018).

[7] A. A. Hwang, I. A. Stefanovici, B. Schroeder, Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design, SIGPLAN Not. 47 (4) (2012) 111–122. `doi:10.1145/2248487.2150989`.
URL `http://doi.acm.org/10.1145/2248487.2150989`

[8] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, W. Kramer, Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters, in: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2014, pp. 610–621. `doi:10.1109/DSN.2014.62`.

[9] E. Meneses, X. Ni, T. Jones, D. Maxwell, Analyzing the interplay of failures and workload on a leadership-class supercomputer, in: CUG 2015, 2015, pp. 1–10.

[10] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, P. Hanrahan, Liszt: a domain specific language for building portable mesh-based PDE solvers, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 9:1–9:12.

[11] F. P. Russell, P. H. J. Kelly, Optimized code generation for finite element local assembly using symbolic manipulation, ACM Trans. Math. Softw. 39 (4) (2013) 26:1–26:29. `doi:10.1145/2491491.2491496`.
URL `http://doi.acm.org/10.1145/2491491.2491496`

[12] M. B. Giles, G. R. Mudalige, Z. Sharif, G. R. Markall, P. H. J. Kelly, Performance analysis and optimization of the OP2 framework on many-core architectures, The Computer Journal 55 (2) (2012) 168–180.

[13] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, S. McIntosh-Smith, The OPS domain specific abstraction for multi-block structured grid computations, in: Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 58–67. doi:10.1109/WOLFHPC.2014.7.
URL http://dx.doi.org/10.1109/WOLFHPC.2014.7

[14] I. Z. Reguly, G. R. Mudalige, M. B. Giles, Design and development of domain specific active libraries with proxy applications, in: Cluster Computing (CLUSTER), 2015 IEEE International Conference on, 2015, pp. 738–745. doi:10.1109/CLUSTER.2015.128.

[15] G. Mudalige, I. Reguly, M. Giles, A. Mallinson, W. Gaudin, J. Herdman, Performance analysis of a high-level abstractions-based hydrocode on future computing systems, in: S. A. Jarvis, S. A. Wright, S. D. Hammond (Eds.), High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation, Vol. 8966 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 85–104. doi:10.1007/978-3-319-17248-4_5.
URL http://dx.doi.org/10.1007/978-3-319-17248-4_5

[16] OPS Library, https://github.com/OP-DSL/OPS (2014).

[17] H. S. Paul, A. Gupta, A. Sharma, Finding a suitable checkpoint and recovery protocol for a distributed application, Journal of Parallel and Distributed Computing 66 (5) (2006) 732 – 749. doi:http://dx.doi.org/10.1016/j.jpdc.2005.12.008.
URL http://www.sciencedirect.com/science/article/pii/S0743731505002662

[18] G. Aupy, Y. Robert, F. Vivien, D. Zaidouni, Checkpointing algorithms and fault prediction, Journal of Parallel and Distributed Computing 74 (2) (2014) 2048 – 2064. doi:http://dx.doi.org/10.1016/j.jpdc.2013.10.010.
URL http://www.sciencedirect.com/science/article/pii/S0743731513002219

[19] A. Gainaru, F. Cappello, M. Snir, W. Kramer, Failure prediction for HPC systems and applications: Current situation and open issues, International Journal of High Performance Computing ApplicationsarXiv:http://hpc.sagepub.com/content/early/2013/07/02/1094342013488258.full.pdf+html, doi:10.1177/1094342013488258.
URL http://hpc.sagepub.com/content/early/2013/07/02/1094342013488258.abstract

[20] J. W. Young, A first order approximation to the optimum checkpoint interval, Commun. ACM 17 (9) (1974) 530–531. doi:10.1145/361147.361115.
URL http://doi.acm.org/10.1145/361147.361115

[21] C. C. J. Li, W. K. Fuchs, CATCH-compiler-assisted techniques for checkpointing, in: [1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium, 1990, pp. 74–81. doi:10.1109/FTCS.1990.89337.

[22] G. Kingsley, M. Beck, J. S. Plank, Compiler-assisted checkpoint optimization using suif, in: First SUIF compiler workshop, Citeseer, 1995, pp. 1–16.

[23] J. S. Plank, M. Beck, G. Kingsley, Compiler-assisted memory exclusion for fast checkpointing, IEEE TECHNICAL COMMITTEE ON OPERATING SYSTEMS AND APPLICATION ENVIRONMENTS 7 (1995) 62–67.

[24] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. De Supinski, R. Eigenmann, McrEngine: a scalable checkpointing system using data-aware aggregation and compression, in: High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for, IEEE, 2012, pp. 1–11.

[25] P. H. Hargrove, J. C. Duell, Berkeley lab checkpoint/restart (BLCR) for Linux clusters, Journal of Physics: Conference Series 46 (1) (2006) 494.
URL `http://stacks.iop.org/1742-6596/46/i=1/a=067`

[26] M. S. Bouguerra, D. Kondo, D. Trystram, On the scheduling of checkpoints in desktop grids, in: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2011, pp. 305–313. `doi:10.1109/CCGrid.2011.63`.

[27] J. Mehnert-Spahn, T. Ropars, M. Schoettner, C. Morin, The architecture of the XtreemOS grid checkpointing service, in: H. Sips, D. Epema, H.-X. Lin (Eds.), Euro-Par 2009 Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 429–441.

[28] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, M. Schulz, Application-level checkpointing for shared memory programs, SIGOPS Oper. Syst. Rev. 38 (5) (2004) 235–247. `doi:10.1145/1037949.1024421`.
URL `http://doi.acm.org/10.1145/1037949.1024421`

[29] A. Chien, P. Balaji, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, J. Hammond, I. Laguna, D. Richards, A. Dubey, B. van Straalen, M. Hoemmen, M. Heroux, K. Teranishi, A. Siegel, Exploring versioned distributed arrays for resilience in scientific applications: global view resilience, The International Journal of High Performance Computing Applications 31 (6) (2017) 564–590. `arXiv:https://doi.org/10.1177/1094342016664796`, `doi:10.1177/1094342016664796`.
URL `https://doi.org/10.1177/1094342016664796`

[30] T. Herault, Y. Robert, Fault-tolerance techniques for high-performance computing, Springer, 2016.

[31] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Aprà, Advances, applications and performance of the global arrays shared memory programming toolkit, Int. J. High Perform. Comput. Appl. 20 (2) (2006)

203–231. doi:10.1177/1094342006064503.
URL http://dx.doi.org/10.1177/1094342006064503

[32] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2) (2008) 4:1–4:26. doi:10.1145/1365815.1365816.
URL http://doi.acm.org/10.1145/1365815.1365816

[33] G. Zheng, X. Ni, L. V. Kalé, A scalable double in-memory checkpoint and restart scheme towards exascale, in: IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012), 2012, pp. 1–6. doi:10.1109/DSNW.2012.6264677.

[34] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, S. Matsuoka, FTI: High performance fault tolerance interface for hybrid systems, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 32:1–32:32. doi:10.1145/2063384.2063427.
URL http://doi.acm.org/10.1145/2063384.2063427

[35] A. Moody, G. Bronevetsky, K. Mohror, B. R. d. Supinski, Design, modeling, and evaluation of a scalable multi-level checkpointing system, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–11. doi:10.1109/SC.2010.18.
URL http://dx.doi.org/10.1109/SC.2010.18

[36] G. Zheng, C. Huang, L. V. Kalé, Performance evaluation of automatic checkpoint-based fault tolerance for AMPI and Charm++, SIGOPS Oper. Syst. Rev. 40 (2) (2006) 90–99. doi:10.1145/1131322.1131340.
URL http://doi.acm.org/10.1145/1131322.1131340

[37] The HDF Group, Hierarchical Data Format, version 5, http://www.hdfgroup.org/HDF5/ (1997-NNNN).

[38] L. W. Howes, A. Lokhmotov, A. F. Donaldson, P. H. J. Kelly, Deriving efficient data movement from decoupled access/execute specifications, in: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC '09, Springer-Verlag, 2009, pp. 168–182.

[39] T. Haerder, A. Reuter, Principles of transaction-oriented database recovery, ACM Computing Surveys (CSUR) 15 (4) (1983) 287–317.

[40] AWE cloverleaf, `https://github.com/UK-MAC` (2014).

[41] C. T. Jacobs, S. P. Jammy, N. D. Sandham, OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures, Journal of Computational Science 18 (2017) 12 – 23. `doi:https://doi.org/10.1016/j.jocs.2016.11.001`.
URL `http://www.sciencedirect.com/science/article/pii/S187775031630299X`

[42] M. Martineau, S. McIntosh-Smith, W. Gaudin, Assessing the performance portability of modern parallel programming models using TeaLeaf, Concurrency and Computation: Practice and Experience 29 (15) (2017) e4117, e4117 cpe.4117. `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4117`, `doi:10.1002/cpe.4117`.
URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4117`

[43] AWE TyphonIO, `https://github.com/UK-MAC/typhonio` (2014).

[44] E. Touber, N. D. Sandham, Large-eddy simulation of low-frequency unsteadiness in a turbulent shock-induced separation bubble, Theoretical and Computational Fluid Dynamics 23 (2) (2009) 79–107. `doi:10.1007/s00162-009-0103-z`.
URL `https://doi.org/10.1007/s00162-009-0103-z`

[45] S. Oral, D. A. Dillow, D. Fuller, J. Hill, D. Leverman, S. S. Vazhkudai, F. Wang, Y. Kim, J. Rogers, J. Simmons, et al., OLCF's 1 TB/s, next-

generation lustre file system, in: Proceedings of Cray User Group Conference (CUG 2013), 2013, pp. 1–12.