Detecting scale-induced overflow bugs in production HPC codes

Published version is available at:
https://link.springer.com/chapter/10.1007/978-3-031-23220-6_3

Detecting scale-induced overflow bugs in production HPC codes

Justs Zarins[1 [0000−0002−0633−1404]], Michèle Weiland[1 [0000−0003−4713−3073]], Paul Bartholomew[1 [0000−0001−7413−670X]], Leigh Lapworth[2 [0000−0002−5705−8102]], Mark Parsons[1 [0000−0003−4097−7468]]

[1] EPCC, University of Edinburgh, UK
[2] Rolls-Royce plc, Derby, UK
j.zarins@epcc.ed.ac.uk

Abstract. Scaling bugs – errors that only manifest at large scale simulations, in terms of number of parallel workers or input size – are critical to detect early in the testing of HPC codes. If missed, these bugs can cause applications to either crash at runtime during production runs or, even worse, silently continue and corrupt results. This results in wasting vast amounts of resources and the crash might not provide any useful debugging information. Laguna et al presented a method for solving this in [13] using an approach where scale variables are traced throughout an application statically and potentially overflowing instructions are detected, with further refinements done by running a few small scale experiments. However, their algorithm is not able to trace multiple code patterns found in production HPC applications, for example code modularity, and has not been applied to Fortran applications. We present an extension to their algorithm which addresses these issues thus enabling us to find scaling bugs in complex real applications where they could not be found before. The key features that enable this are backward/forward tracing and optimistic GEP comparison.

Keywords: Scaling bugs · Correctness · LLVM

1 Introduction

Verifying the correctness of supercomputing applications is a significant challenge, not least due to the ever increasing scale that these applications run at. The majority of testing is done with either serial or small scale parallel runs, with the largest scales reserved for production jobs. This approach of testing at small scale cannot catch "scaling bugs" – errors that occur as the number of processes used by a program, or the size of the simulation, increases. As a result, an application that is deemed to be correct may fail only on a large scale production run, at which point the cost of the failure is huge and debugging information likely unavailable. It is therefore desirable to be able to anticipate and report such issues based on small scale testing only.

A promising approach was presented by Laguna and Schulz [13] to predict integer overflow bugs and pinpoint their location in an application's code. In

their method, an application is analysed at the LLVM bitcode level, marking scale dependent variables (such as the number of MPI ranks or the size of the input) and identifying integer arithmetic instructions that are influenced directly or indirectly by the scale variables. The marked instructions are narrowed down to the most likely to cause overflow bugs by running an instrumented version of the application at small scales and logging the relationship between the resulting values of the instructions and the scale of each run. Their method proved to be successful in finding many scaling bugs in multiple C/C++ test applications and benchmarks, as well as the widely used MPICH library [10].

We adopted their method to analyse the scaling behaviour of supercomputing applications of interest to our research. However we encountered a number of significant limitations in the power of the method, most importantly the ability to verify Fortran applications (which continue to represent a very large fraction of applications run on supercomputers). Additionally, the complexity of production applications, which caused no instructions to be traced at all (see Sec. 3), also needed to be addressed in order for the approach to be viable for our purposes. Real world supercomputing applications are often structured in a modular way, which precludes straight-forward tracing between scale variables and the affected instructions, but this can be disentangled with "optimistic" analysis. In this paper we present *OFT*[3], the Overflow Tool, which includes an extension to the tracing algorithm first presented in [13] to handle modular code, with a view to support Fortran applications in particular, and significantly expand the ability to find scaling bugs in production codes.

Our specific contributions presented in this paper are:

- Enable the tracing of integer overflow bugs in Fortran-based applications;
- Present an extension to the tracing algorithm that uses a *backwards/forwards* approach to support tracing of modular code;
- Enable tracing of complex data structures, such as allocatable arrays in Fortran and heap allocated structures in C.

2 Tracing algorithm extension

OFT is implemented as an LLVM Module pass and analyses an application in two steps, a static and a dynamic one. The static step detects scale variables that are marked either by MPI communicator size functions or by the user in a whole-application bitcode. These variables are traced to find all 32-bit integer arithmetic operation instructions that are influenced by the scale variables. The user is presented with a list of these instructions, including their location in the source code. Additionally, a modified version of the bitcode is generated, where all scale affected instructions are instrumented to record their maximum value encountered at runtime.

[3] The code is available at https://github.com/asimovpp/oft

Listing 1.1: Limited backward tracing used in [13]. A scale variable is marked on line 5, and influences a potentially overflowable instruction on line 8.

```
1 %rank_299 = alloca i32, align 4
2 ...
3 %2 = bitcast i32* %rank_299 to i8*
4 %3 = bitcast void (..)* @oft_mark_ to void (i8*, ..)*
5 call void (i8*, ..) %3(i8* %2)
6 ...
7 %15 = load i32, i32* %rank_299, align 4
8 %16 = mul nsw i32 %15, 3
```

The user may run the instrumented application to perform the dynamic part of the analysis to reduce the number of false positives. This can be done by running the instrumented application at a few small scales and passing the output to our analysis tool. The tool fits a linear or polynomial function to the max-value versus scale-size relationship of each recorded instruction. The fitted lines can then be extrapolated to scales relevant to each application and it can be seen whether any instruction will overflow. The amount of overhead introduced by instrumentation depends on the number of instructions instrumented, but this can be significantly reduced by focusing on instructions most likely to overflow [13]. Note that multiple test cases may be required in order to exercise every part of a codebase.

Listing 1.2: Common pattern that requires advanced tracing.

```
1  struct my_mpi { int rank; int size; };
2
3  void set_scale_var(struct my_mpi *sv) {
4      MPI_Comm_size(MPI_COMM_WORLD, &(sv->size))
5      MPI_Comm_rank(MPI_COMM_WORLD, &(sv->rank))
6  }
7
8  int main() {
9      struct my_mpi *sv = malloc(sizeof(struct my_mpi));
10     set_scale_var(sv);
11     return sv->rank * 3 + sv->size * 7;
12 }
```

In the method presented in [13] scale variables can only be traced if the emitted instructions are close in scope, for example the scale variable has to be declared, set and then used in subsequent instructions (see Listing 1.1). However, more complicated patterns are often used in real applications, for example to organise the information pertaining to the configuration of a simulation (see Listing 1.2). In such a scenario there may be an initialisation function that sets the MPI environment and other scale variables in a container data structure, which is accessed later in the application's code. As a result, there will be a local pointer in an initialisation function which is set as the scale variable; its tracing

is contained within the initialisation function and the rest of the application cannot be reached when tracing. Examples such as the one in Listing 1.3 were not traceable prior to the extensions we introduced as part of OFT.

Listing 1.3: An example of extended tracing enabled by the method presented in this paper. A scale variable is marked on line 10, and influences a potentially overflowable instruction on line 20.

```
1  %.Z0632_306 = alloca i32*, align 8
2  ...
3   %16 = load i32*, i32** %.Z0632_306, align 8
4   %17 = bitcast i32* %16 to i8*
5   %18 = getelementptr i8, i8* %17, i64 4
6   %19 = load i64, i64* %z_b_3_302, align 8
7   %20 = mul nsw i64 %19, -4
8   %21 = getelementptr i8, i8* %18, i64 %20
9   %22 = bitcast void (..)* @oft_mark_ to void (i8*, ..)*
10  call void (i8*, ..) %22(i8* %21)
11 ...
12  %48 = load i32*, i32** %.Z0632_306, align 8
13  %49 = bitcast i32* %48 to i8*
14  %50 = getelementptr i8, i8* %49, i64 4
15  %51 = load i64, i64* %z_b_3_302, align 8
16  %52 = mul nsw i64 %51, -4
17  %53 = getelementptr i8, i8* %50, i64 %52
18  %54 = bitcast i8* %53 to i32*
19  %55 = load i32, i32* %54, align 4
20  %56 = mul nsw i32 %55, 3
```

To address this scenario, it is necessary to trace the scale variable back to its "root", i.e. the first instruction defining the variable: we call this part of the analysis "backward tracing". If GetElementPointer (GEP) instructions[4] are involved, additional steps are required to find instructions performing equivalent memory accesses to the one of the original scale variable. This can result in multiple instructions from which to start tracing the rest of the application.

There are three stages in the backward/forward tracing method:

1. Trace the scale variable *backwards* to its root and record a track.
2. Analyse the track, resolving the root and GEPs in the track, to identify accesses equivalent to the one made by the originally marked scale variable.
3. Trace the accesses *forwards* to the rest of the application.

Stage 1 Firstly, marked scale variables are traced backwards as far as possible in order to find the originally allocated variable. This tracing is done by following bitcast, load, store and GEP operators, as well as function arguments.

[4] GEP instructions calculate addresses of sub-elements of data structures based on a base pointer and one or more indices into the data structure [2]

All encountered instructions form a list and are stored as a *trace*. Traced backward, one scale variable may result in multiple traces due to the possibility of a function being called from multiple locations in the analysed application.

Stage 2 Secondly, the traces produced in stage 1 are analysed to find all instructions that access the marked scale variable either directly or via a GEP instruction. The trace is iterated through and the GEP instructions are added to a list. The iteration stops when an stack allocation or global variable, or a call instruction to a predefined function (e.g. malloc), are encountered. The terminating instruction is recorded as the "root" of the trace. The "transitional" instructions like bitcasts and loads/stores are skipped while traversing the trace.

Next, the reduced traces are used to generate a list of scale instructions that indirectly connect to the originally traced scale variable.

1. If a root has been recorded without any GEPs, the root is returned.
2. If a root and a single GEP have been recorded, find and return all equivalent GEPs to the one that was recorded.
3. If a root and multiple GEPs have been recorded, find equivalent GEPs for each recorded GEP in reverse order (resetting the root at each level to the intermediate GEP), and return the last level of equivalent GEPs.

In order to connect a scale variable set via a GEP to further uses of that scale variable, we must find equivalent GEPs to the source GEP. We do this by first finding all GEPs that use the same base pointer as the source GEP by following the define-use relationship chain of the base pointer, including through store and call instructions. The search stops on each branch when the first GEP operation is encountered on that branch, or if there are no further instructions to follow.

Each discovered GEP is compared to the source GEP by comparing the indices of both instructions (the base pointers have already been established to be equal due being results of tracing). Supported types of indices are simple integers, results of load instructions and results of arithmetic operations. For indices that are the results of other instructions, the comparison is applied recursively, thus supporting a sequence of arithmetic operations which computes an index. Nested GEPs with complex index calculations can be generated by Fortran applications that use allocatable arrays which are supported by array descriptors.

Listing 1.3 shows an example where GEP comparison is required, resulting from a Fortran array descriptor. A scale variable is marked on line 10 and can be traced backwards through two GEP instructions (including index calculations) to an `alloca` instruction root. The same variable is accessed (and used thereafter) on line 17, which can be seen by comparing lines 3–8 with lines 12–17.

The GEPs found in this way are not guaranteed to resolve to the same memory accesses in runtime because we do not consider all possible memory interactions that could have happened between the accesses. Doing a definitive trace statically is expensive and impossible for most complex applications. Hence we call these GEPs equivalent, not equal. However, scale variables are normally

set at the beginning of an application run and remain unchanged and are not overwritten. Therefore, performing "optimistic" GEP comparisons will produce the correct results in this use case.

Stage 3 Finally, the list of scale instructions are passed on to forward tracing where their influence on the rest of the application is established, as in [13]. Scale instructions are iteratively traced via define-use relationships, including store and load pairs, up to function calls and those calls expanded until no more changes occur.

2.1 Fortran support

In principle, performing analysis at bitcode level should allow Fortran support automatically. However, in practice, there are key differences between how Fortran bitcode and, for example, C bitcode is generated, which prevent tracing. In Fortran bitcode function calls are performed with an intermediate bitcast operation, which obscures the function name and thus the starting points of tracing cannot be identified. Also, global variables stored in modules are accessed like structure elements, not directly. These issues can be overcome, but they require special cases in the analysis code. However, dynamic arrays in Fortran are significantly more challenging to handle, due to the reliance on array descriptors, and require multi-step processing, such as backwards/forwards tracing described in this paper.

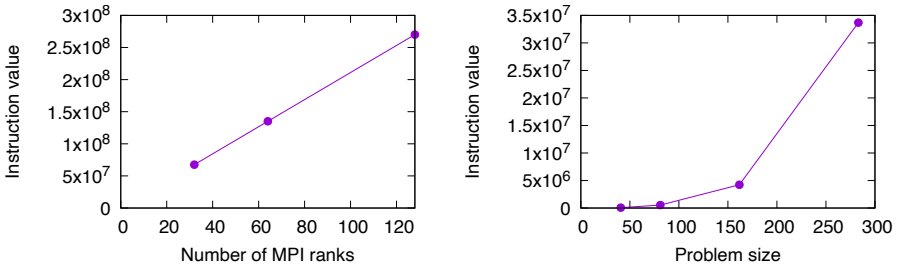
3 Evaluation

To evaluate OFT, we compare its results in two configurations: one that replicates the functionality of the original tool in [13], and the second where we extend its functionality with backward/forward tracing.

The evaluated applications were compiled using LLVM 12.0.0 clang and classic flang [1] without optimisations (`-O0`). We chose to use `-O0` to retain a close link between generated bitcode and the source code, which facilitates identification of scale bugs in the analysed application. Whole-application bitcode was generated at link time using the LLVM gold linker plugin by adding `-flto` to compilation steps and `-fuse-ld=gold -Wl,-plugin-opt=emit-llvm` to the linking step. OFT analyses the bitcode and detects functions that set MPI rank and size automatically for tracing. User defined scale variables are detected if they are passed to a function called `oft_mark_(variable)`; the function does not perform any actions, OFT merely registers arguments passed to it for tracing. Note that constant variables defined using macros (e.g. `#define` in C) cannot be marked in this way. Also, dynamically linked libraries, where the bitcode has not been generated and linked during compilation, are not traced.

Xcompact3d Xcompact3d [5, 14, 15] is an open source [4] solver for the incompressible and low Mach number Navier-Stokes equations focused on Large Eddy and Direct Numerical Simulations (LES and DES) of turbulent flows. It is based on a fractional step method for time advancement with a direct Poisson solver based on FFTs for the enforcement of the velocity divergence constraint [5, 15]. Combined with 6^{th} order accurate compact finite difference schemes [17] for derivative approximations a quasi-spectral accuracy is achieved. Both the FFTs for the Poisson solver and the compact finite difference schemes map naturally to a 2D pencil-based parallel decomposition as provided by the 2DECOMP&FFT library [18]. As the FFTs and compact finite differences operate in a single pencil at a time the data must be transposed between different decompositions for each spatial dimension, these are implemented using `MPI_ALLTOALL(V)` and the code makes multiple MPI calls per time step.

Implemented in approximately 50k lines of Fortran the resulting whole-application bitcode for the analysis is about 1M lines long. The program accepts as input numerous parameters related to the problem to be studied, in terms of scale-dependence these include the mesh size $n_x \times n_y \times n_z$ and the $p_{row} \times p_{col} = np$ pencil decomposition where np is the number of MPI ranks.



(a) MPI size scaling **(b)** Mesh size scaling

Fig. 1: Potentially overflowable instruction values at small scale in Xcompact3d. When extrapolated, (a) will overflow when MPI ranks exceed around 1,000 and (b) when the problem size exceeds 1,291. The problem size is defined as $\sqrt[3]{nx \times ny \times nz}$.

It was found that without backward/forward tracing no tracing was possible at all for Xcompact3d, resulting in no marked overflowable instructions. The reason is that in Xcompact3D the scale variables are stored in a module separate from where they are initialised. With our new backward/forward tracing enabled, OFT reported around 1,000 potentially overflowable instructions. Among these, the value of two calculations grew rapidly in the dynamic part of the analysis, as shown in Figure 1. These corresponded to two code locations, one initialising random number generator seeds based on the MPI rank and the other printing the total size of the problem. Based on the rate of growth of the values observed

in small scale runs, the value in Fig. 1a extrapolated linearly would overflow at around 1,000 MPI ranks and the value in Fig. 1b extrapolated exponentially would overflow with a $1,291^3$ problem size. These values are representative of real-world code use: production runs of 1,000 MPI ranks or more are very common, with scaling demonstrated well beyond this [5, 15], and problems in excess of $1,000^3$ mesh nodes have been used to assess performance at extreme scales [8]. While the two bugs that we identified are not critical to the application's successful execution *per se* (it will run and produce correct results), detecting them demonstrates the necessity for our tracing extensions for production HPC applications. The random seed-related overflow bug in particular has potentially serious implications for reproducibility and validation of results in the future. It is worth noting that it only took a few hours to analyse Xcompact3d, with most of that time spent compiling code and running small scale experiments, thus this is an efficent method for assessing application correctness.

OPlus parallel library The Oxford Parallel Library for Unstructured Solvers (OPlus) [6] aims to ease the development of parallel solvers for unstructured grids written in Fortran by insulating the programmer from the burden of writing parallel code. To do so OPlus introduces parallel abstractions such as `op_par_loop` which allow the application to be written as though serial and executed in parallel by the OPlus framework. By abstracting the parallel execution a program written in the OPlus framework may be executed in a distributed or shared memory context without changes to the source code [6]. This not only allows application experts to focus on their problem domain but also opens the possibility for significant impact through optimisations to the OPlus framework benefiting the whole ecosystem of OPlus programs. The library has been applied to real world problems, for example Crumpton et al [7] show results obtained using a multigrid solver parallelised with OPlus to perform aerodynamic calculations of an aircraft and it has been used in developing industrial CFD codes [16]. The success of this approach has led to the development of a follow on OP-DSL effort [3] which will be the focus of future work. Also studied as part of this work is the ParMETIS [11] parallel graph partitioning library that can be used by OPlus to decompose and reorder the input mesh for parallel processing. It is written in C and parallelised using MPI.

Our analysis shows that without backward/forward tracing some scale instructions are marked, but many more are identified with our extensions to tracing (see Table 1). This shows better code coverage and increases the confidence in the program's correctness after analysis.

We used a 3D Poisson solver which uses multigrid with Jacobi smoothing as a test case to drive the OPlus library for the dynamic part of the analysis. No MPI size related issues were found in OPlus, even when projecting out to 500,000 MPI ranks. However, problem scaling (which is defined by the size of the finest grid) revealed potential overflow bugs related to checking buffer sizes (shown with coloured lines in Fig. 2) which may be triggered with problem size of 1024 and greater. Multiple other instructions (shown in grey in Fig. 2) would overflow

if extrapolated exponentially further to 2048. Unfortunately, the test program does not exercise ParMETIS code so we plan to investigate this in future work, in addition to the instructions that may overflow at very large scales.

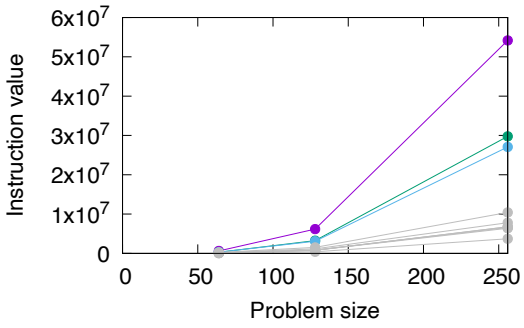


Fig. 2: Potentially overflowable instruction values at small scale in OPlus. When extrapolated, the coloured instruction lines will overflow when the problem size exceeds 1,024 and the gray instruction lines when the problem size exceeds 2,048. The problem size is defined by the size of the finest grid.

Table 1: OFT static analysis result summary. *Lines of Bit Code are given for a test application which includes both OPlus and ParMETIS.

Application	LoBC	Instructions marked (old)	Instructions marked (new)
Xcompact3d	1,167,461	0	1,058
OPlus	154,109*	121	1,156
ParMETIS	154,109*	48	130

4 Related work

Verification and validation are two intensely studied areas in software development. In the world of scientific and large-scale parallel computing, they have added layers of complexity: will the answer remain correct for an arbitrary level of parallelism, will the application complete successfully when it is scaled up, and are the scale and level of parallelism themselves potential sources of incorrectness? The "Report of the HPC Correctness" summit [9] provides a comprehensive overview of these different topics.

Laguna et al [12] discuss the general challenges when it comes to trying to address errors at scale. Aside from it being technically challenging, with poor support from debugging tools at very large scales, it is also expensive and time

consuming. The conclusion is that ideally it should be possible to predict errors that occur when running at scale based on much smaller job sizes. In [13], the paper that describes the method on which OFT is based, the authors take this learning and apply it to scaling bugs using tracing and dynamic analysis.

Another approach for detecting scaling bugs is presented in [19]. The authors favour a statistical approach, building and applying models of known bug free behaviour at small scale. When an application is run at larger-scale, the behaviour should remain the same; if an error is detected, the correlation will break. A follow-on paper [20] extends the method by adding automatic pinpointing of errors to a region of the code, which was previously not possible. Although both approaches can make use of small-scale runs, and are implementation language agnostic, they rely on data from larger scales in order to detect and locate issues with the application.

5 Conclusion

We have presented an an extension to the scale variable tracing algorithm presented in [13]. The backward/forward tracing algorithm enables us to detect more integer oveflow bugs than was possible before as well as to analyse modular Fortran-based applications. For the first time, we were able to evaluate the correctness of Xcompact3d application, finding two scaling bugs, and greatly expanded the analysis coverage for the OPlus and ParMETIS libraries. We plan to expand the capabilities of OFT to detect other kinds of scaling issues, for example memory usage scaling.

6 Acknowledgements

This research forms part of the Rolls-Royce led EPSRC Prosperity Partnership (Grant Ref: EP/S005072/1) entitled “Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Virtual Systems – AsiMoV”. The permission of Rolls-Royce to publish this paper is gratefully acknowledged.

References

1. Classic Flang. https://github.com/flang-compiler/flang, accessed: 2022-02-27
2. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html #getelementptr-instruction, accessed: 25/02/2022
3. Oxford Parallel Domain Specific Languages. https://op-dsl.github.io/, accessed: 2022-02-26
4. Xcompact3d github repository. https://github.com/xcompact3d/Incompact3d, accessed: 2022-02-25
5. Bartholomew, P., Deskos, G., Frantz, R.A., Schuch, F.N., Lamballais, E., Laizet, S.: Xcompact3D: An open-source framework for solving turbulence problems on a Cartesian mesh. SoftwareX **12**, 100550 (2020)

6. Burgess, D.A., Crumpton, P.I., Giles, M.B.: A parallel framework for unstructured grid solvers. Tech. Rep. NA-95/20, Oxford University Numerical Computing Laboratory, Numerical Analysis Group (1994), https://people.maths.ox.ac.uk/ gilesm/files/NA-95-20.pdf
7. Crumpton, P.I., Giles, M.B.: Multigrid aircraft computations using the OPlus parallel library. In: Parallel Computational Fluid Dynamics 1995, pp. 339–346. Elsevier (1996)
8. Giannenas, A.E., Laizet, S.: A simple and scalable immersed boundary method for high-fidelity simulations of fixed and moving objects on a cartesian mesh. Applied Mathematical Modelling **99**, 606–627 (2021)
9. Gopalakrishnan, G., Hovland, P.D., Iancu, C., Krishnamoorthy, S., Laguna, I., Lethin, R.A., Sen, K., Siegel, S.F., Solar-Lezama, A.: Report of the HPC Correctness Summit (2017)
10. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the mpi message passing interface standard. Parallel computing **22**(6), 789–828 (1996)
11. Karypis, G.: METIS and ParMETIS. In: Padua, D.A. (ed.) Encyclopedia of Parallel Computing, pp. 1117–1124. Springer (2011)
12. Laguna, I., Ahn, D.H., de Supinski, B.R., Gamblin, T., Lee, G.L., Schulz, M., Bagchi, S., Kulkarni, M., Zhou, B., Chen, Z., Qin, F.: Debugging high-performance computing applications at massive scales. Commun. ACM **58**(9), 72–81 (aug 2015). https://doi.org/10.1145/2667219, https://doi.org/10.1145/2667219
13. Laguna, I., Schulz, M.: Pinpointing scale-dependent integer overflow bugs in large-scale parallel applications. In: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 216–227. IEEE (2016)
14. Laizet, S., Lamballais, E.: High-order compact schemes for incompressible flows: A simple and efficient method with quasi-spectral accuracy. Journal of Computational Physics **228**(16), 5989–6015 (2009)
15. Laizet, S., Li, N.: Incompact3d: A powerful tool to tackle turbulence problems with up to O(105) computational cores. International Journal for Numerical Methods in Fluids **67**(11), 1735–1757 (2011)
16. Lapworth, L.: Hydra-CFD: a framework for collaborative CFD development. In: International conference on scientific and engineering computation (IC-SEC). vol. 30 (2004)
17. Lele, S.K.: Compact finite difference schemes with spectral-like resolution. Journal of computational physics **103**(1), 16–42 (1992)
18. Li, N., Laizet, S.: 2DECOMP&FFT–A highly scalable 2D decomposition library and FFT interface. In: Cray User Group 2010 conference. Edinburgh, UK (2010)
19. Zhou, B., Kulkarni, M., Bagchi, S.: Vrisha: Using scaling properties of parallel programs for bug detection and localization. pp. 85–96 (2011). https://doi.org/10.1145/1996130.1996143
20. Zhou, B., Too, J., Kulkarni, M., Bagchi, S.: WuKong: automatically detecting and localizing bugs that manifest at large system scales. pp. 131–142 (2013). https://doi.org/10.1145/2462902.2462907