2016 15th International Symposium on Parallel and Distributed Computing

Massive Parallel Fluid Flow Simulations using Hierarchical Data Format Version 5 (HDF5)

Christoph Ertl
Computation in Engineering
Technical University of Munich
Germany
Email: christoph.ertl@tum.de

Jérôme Frisch
Institute of Energy Efficiency and
Sustainable Building E3D
RWTH Aachen University
Germany
Email: frisch@e3d.rwth-aachen.de

Ralf-Peter Mundani
Computation in Engineering
Technical University of Munich
Germany
Email: mundani@tum.de

***Abstract*—More and more massive parallel codes running on several hundreds of thousands of cores enter the computational science and engineering domain, allowing high-fidelity computations up to trillions of unknowns for very detailed analyses of the underlying problems. During such runs, typically gigabytes of data are being produced, hindering both efficient storage and (interactive) data exploration. Here, advanced approaches based on inherently distributed data formats such as HDF5 become necessary in order to avoid long latencies when storing the data and to support fast (random) access when retrieving the data for visual processing. Avoiding file locking and using collective buffering, we achieved write bandwiths to a single file close to the theoretical peak on a modern supercomputing cluster. The structure of our output file supports a very fast interactive visualisation and introduces additional steering functionality.**

I. Introduction

According to the U.S. Presidents Council of Advisors on Science and Technology *'high-performance computing must now assume a broader meaning, encompassing not only flops, but also the ability, for example, to efficiently manipulate vast and rapidly increasing quantities of both numerical and non-numerical data'* [1]. Latest advances in hardware led to high-performance computing (HPC) systems consisting of hundreds of thousands to millions of cores that exhibit petaflop performance for high-fidelity applications. Designing massive parallel codes that can utilise such systems is a challenging task itself, being able to handle the inherent huge data advent – easily exceeding tons of gigabytes per computational step – is yet another one. This raises the necessity for sophisticated concepts to store and interact with the computed data—even in real-time.

Within our researches, we have developed a computational fluid dynamics (CFD) code for multi-scale, multi-physics problems arising in the field of computational engineering with special emphasis on thermal comfort assessment. This code has been successfully deployed up to a total of 140,000 processes on two of Germany's three top-ranked supercomputers, namely SuperMUC (based on System x® iDataPlex® dx360 M4 compute nodes) installed at Leibniz Supercomputing Centre and JuQueen (Blue Gene/Q system) installed at Jülich Supercomputing Centre. Core part of our code is a hierarchical data structure consisting of logical and computational grids following a space-tree based [2] spatial partitioning. In conjunction with a so-called neighbourhood server, a topological repository storing which computational grid resides on which process, this structure fosters distributed computing and supports efficient numerical solvers as well as dynamic load balancing strategies.

Furthermore, the structure also supports in-situ monitoring of the computed data, i. e. users have the possibility to retrieve computation results already during runtime for visual exploration. A sliding window called approach allows to interactively select any region of the computational domain, whereupon the size of the window defines the corresponding level-of-detail, thus keeping the total amount of data to be transmitted between the CFD code and the user constant in order not to exceed given bandwidth limitations [3]. Hence, a window covering the entire domain allows for a qualitative evaluation of the global flow field while smaller windows – cf. a zooming into the data – reveal more and more details for quantitative assessments. Even for huge domains with trillions (i. e. 10^{12}) of unknowns, the sliding window approach is advantageous as only small parts of the data need to be processed for visual display. For further research regarding actual HPC in-situ visualisations the reader is advised to Rivi et al. [4].

Nevertheless, at certain time steps the data will be written to a storage system either for checkpointing (fault tolerance) purposes or an offline post-processing of the computation results. During this time, all processes cannot continue with their computations and have to wait until the data dump has been finished, thus slowing down the entire execution. Finally, the user is left with tons of sequentially ordered data that – due to its mere size – forbid any efficient access or treatment within subordinated post-processing tools. In order to tackle this problem, we have implemented an I/O kernel based on HDF5 that supports parallel I/O functionality and in addition allows to utilise our sliding window approach even on offline data. Hence, users can switch between online (i. e. present) and offline (i. e. past) data for visual exploration—practically they can reverse in time. Such a time reversal further provides the possibility to modify a scenario at any point in time and re-compute it with altered settings in case of undesired results

978-1-5090-4152-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ISPDC.2016.13

or effects, thus opening the door for computational steering or interactive computing applications.

The remainder of this paper is organised as follows. In the next section, the basic mathematical concepts as well as our data structure are introduced, including some runtime and speed-up measurements done on SuperMUC and JuQueen. The third section will address the implementation of our I/O kernel using HDF5, whereas the achieved results will be presented in section four. Finally, section five will close with a short summary and outlook.

II. Fluid Flow Simulations

In this section, we will present the foundations of our CFD code *mpfluid* including all mathematical concepts. Main contribution is a hierarchical data structure inherently supporting distributed computing and allowing for in situ data exploration already during runtime.

A. Mathematical Modelling

The implementation of our CFD kernel is based on the Navier–Stokes equations which can be derived from the conservation of mass, momentum, and energy principles [5] [6].

The governing equations for incompressible Newtonian fluid flows consist of the continuity equation and the momentum equations. The continuity equation can be written in differential form as

$$\nabla \cdot \vec{u} = 0\,, \tag{1}$$

where $\vec{u}$ describes the velocity vector of the flow field. For a divergence free vector field, the continuity equation has to be satisfied at every time step in the entire domain.

The momentum equations for every direction $i \in \{1, 2, 3\}$ can be written in differential form as

$$\frac{\partial \rho_\infty u_i}{\partial t} + \nabla \cdot (\rho_\infty u_i \vec{u}) = \nabla \cdot (\mu \nabla u_i) - \nabla \cdot (p\vec{e}_i) + b_i\,, \tag{2}$$

where t represents the time, ρ_∞ the density of the fluid (assumed constant over the entire domain), u_i the velocity in direction i, μ the dynamic viscosity, p the pressure, b_i some interior body forces in direction i, and $\vec{e}_i$ the unit vector in direction i.

In order to include thermal effects such as buoyancy, the last part b_i on the right-hand side in (2) must be replaced by $\rho_\infty \cdot \beta \cdot (T - T_\infty) g_i$ to couple effects of the temperature field to the momentum equations, commonly known as the Boussinesq approximation, see Lienhard and Lienhard [7] for instance. Here, β describes the thermal expansion coefficient of the fluid, T the temperature, T_∞ the temperature of the undisturbed fluid at rest, and g_i the gravitational force in direction i. Finally, for modelling the thermal heat transport, the energy equation (adapted from a generic convection-diffusion equation to a heat transfer problem) can be written in differential form as

$$\frac{\partial T}{\partial t} + \nabla \cdot (T\vec{u}) - \nabla \cdot (\alpha \nabla T) - \frac{q_{int}}{\rho_\infty \cdot c_p} = 0\,, \tag{3}$$

where $\alpha = k/(\rho_\infty \cdot c_p)$ represents the heat diffusion coefficient, k the heat conduction coefficient, c_p the specific heat capacity at constant pressure, and q_{int} the internal heat generation.

As pressure p is solely contained in its gradient form ∇p in (2), some pressure correction methods as proposed by Harlow and Welch [8] or Chorin [9] have to be applied. In our approach, we use the so-called fractional step or projection method introduced by Chorin that iterates between the velocity and pressure fields, where the latter one acts as correction to the velocity field in every time step to fulfil (1). Choosing an explicit Euler time discretisation for the temporal derivatives $\partial/\partial t$, we are eventually left with a Poisson equation for the pressure that has to be solved in every time step and also determines the computationally most complex part. For the spatial discretisation we use a finite volume method that – due to the block substructuring of the domain into regular Cartesian grids – locally degenerates into finite differences and, thus, favours fast computations based on standard stencil operators.

In the next part, we will now introduce our data structure together with a multigrid-like solver – directly derived from the data structure's exchange routines – for the solution of the pressure Poisson equation.

B. Data Structure

The data structure follows the general idea of space-trees (with quadtrees as 2D and octrees as 3D representatives) for a spatial partitioning. Starting from a single root, each cell is further subdivided by $r_x \times r_y \times r_z$ cells until a predefined depth d_{max} has been reached. This hierarchy of cells defines the logical part of the structure – also called logical grid or l-grid – and plays a vital role when retrieving any hierarchic grid information. For computation purposes, now every cell of this logical grid links to a data grid of size $s_x \times s_y \times s_z$ that stores all necessary variables such as velocities, pressure, or temperature values. Furthermore, each data grid – also referred to as d-grid – is surrounded by a halo (currently of size one) for the proper data exchange between d-grid boundaries. This completes the data structure which is composed of block-structured, nonoverlapping, orthogonal, regual, hierarchical grids.

To keep track about the distribution of d-grids to (MPI) processes, a dedicated process called neighbourhood server was implemented. This server stores the entire logical structure, the l-grid, in order to answer topological queries, while all computational processes solely store the d-grids assigned to them. The assignment per se follows a space-filling Lebesgue curve that has proven to preserve neighbouring relations, thus reducing the necessary communication overhead. For a ghost layer update, any computational process queries the neighbourhood server by its own d-grid IDs in order to obtain all neighbouring d-grid IDs and their physical residence (i. e. MPI ranks). Afterwards, the process can launch an inter-grid data transfer (communication phase) which is strictly separated from the computation phase.

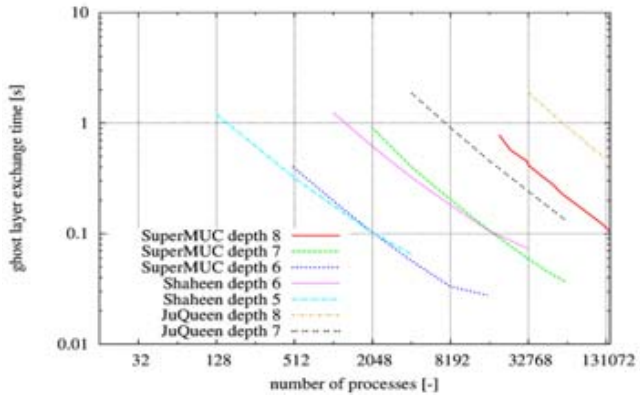

Fig. 1. Total ghost layer exchange times up to a resolution of $4096 \times 4096 \times 4096$ (depth 8) with approx. 707 billion unknowns to be exchanged for different amounts of processes on different HPC systems

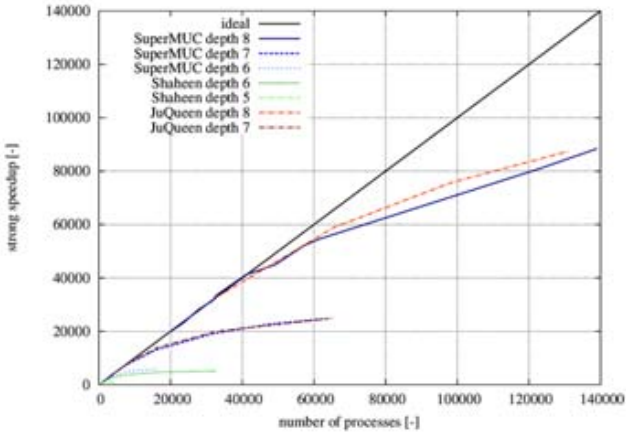

Fig. 2. Strong speed-up values for different resolutions up to $4096 \times 4096 \times 4096$ (depth 8) with approx. 707 billion unknowns on different HPC systems

The communication phase consists of three sequential steps as described in [10]. First of all, in a bottom-up step all d-grids that have not been updated yet during the computation phase are set to the averaged values of their corresponding child d-grids. In a second, horizontal step all adjacent d-grids update their ghost layers before in a last, top-down step all resulting ghost layers of d-grids on different levels (due to an adaptive grid refinement) are set properly. Here, the l-grid management must take care about flux conservation across d-grid boundaries in order to guarantee data integrity and consistency. Whereas the communication phase is not very time consuming – a full update for a domain with resolution $4096 \times 4096 \times 4096$ resulting to more than 700 billion unknowns to be exchanged takes around 0.1 s on 140,000 cores on SuperMUC, see Fig. 1 – according to [11] the parallel code spends more than 90 % of the time in the computation phase for solving the pressure Poisson equation.

Therefore, we have implemented a parallel multigrid-like solver following the ideas of Brandt [12] for solving elliptic partial differential equations. Multigrid-like, because we utilise the above communication schema – precisely the bottom-up and top-down update steps – as restriction and prolongation operators for setting up a cell-centred multigrid method, thus making use of the data structure's superior parallel performance and scalability properties. Nevertheless, the multigrid-like solver exhibits convergence instabilities for certain scenarios (in case of adaptive refinement, e. g.) which can be handled by different smoothing strategies such as doubling the amount of pre- and post-smoothing steps on coarser resolutions. Details about all performed analyses and comparisons can be found in [13]. Fig. 2 and Fig. 3 show the obtained strong speed-up and time-to-solution values of the multigrid-like solver for different domains on different architectures. Interestingly, for the curves in Fig. 3 both architectures SuperMUC and JuQueen reveal the same slope—any difference in time is only due to SuperMUC's higher clock frequency.

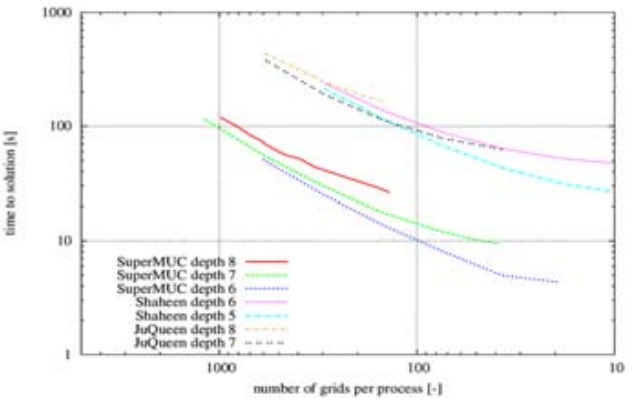

Fig. 3. Time-to-solution (one full time step) for different resolutions up to $4096 \times 4096 \times 4096$ (depth 8) with approx. 707 billion unknowns plotted against the number of d-grids per processes

C. Sliding Window

To reduce the data transfer between the CFD code running on an HPC system – called back end – and the user application for visual display – called front end – the previous data structure was extended with the idea of the sliding window concept. Main strategy is to select on the back end only subsets of the computed data in order to stay below the available network bandwidth between front and back end. Hence, any user has the possibility to choose a region of interest (the window) which can be moved around the computational domain and increased or decreased in size. The larger the window, the lower will be the density of data points to be considered for the visual display—even a higher density of data points would be available, depending on the window size every second, third, fourth, and so on data point will be dismissed. This approach allows to catch either global effects of the simulation (large window) or to explore local details (small window) without overloading the network with unnecessary data and, thus, harming the experience of an authentic interactive computing.

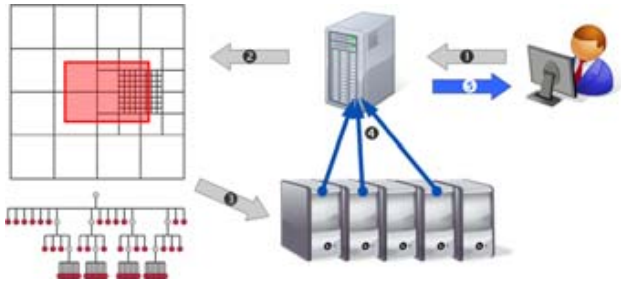

Fig. 4. Single steps of a full sliding window query: the client issues a request to the collector (1) that forwards it to the neighbourhood server for identifying all relevant d-grids (2) and informs respective computational processes (3) to send the desired information to the collector (4) that finally returns a data stream to the client (5) – based on [3]

The sliding window implementation practically consists of two components: a back end collector for gathering the desired data and a front end visualisation and interpreter tool for sending data requests. Therefore, a new process called collector was introduced at the back end, listening for user requests on a standard TCP socket. The collector forwards the query to the neighbourhood server which can easily identify all computational processes storing relevant d-grids intersecting with the window. These processes themselves send all selected data points to the collector which returns a compressed data stream to the front end application. On the front end we use ParaView [14] for the visualisation of the fluid flow data. A special ParaView plug-in allows the user to connect to a running simulation, to set all necessary sliding window parameters, and to retrieve the desired data for an interactive visual data exploration. Fig. 4 shows a full pass of a sliding window query.

III. I/O Kernel

In order to be able handling the large amount of data our CFD code produces, a dedicated approach to the code's I/O routines had to be employed. One major building block of this approach comprises the use of the high-level data format HDF5 (Hierarchical Data Format version 5) [15]. HDF5 is specifically tailored to store large amounts of array based data in a database-like view. The key concept behind HDF5's data model is based on datasets which contain the actual data and groups making up the general structure. Starting from a root group, groups may contain additional groups or datasets themselves, resulting in a hierarchical tree-like structure resembling a Unix file system. Attributes allow to describe the data and can be associated to a group or dataset. The layout of the data is specified by HDF5's storage model, while the HDF5 library takes care of the conversion between the database-like view of the data model and the storage model. The user may be completely oblivious to the way his data looks like on the file system. Considering performance, however, it is necessary to be aware of the layout of the data in an HDF5 file. We will emphasise on considerations concerning this in III-B.

HDF5 is a self-describing format, i. e. a file contains information about its structure as well as the used data types. Portability is a huge concern in today's diverse architecture landscape. Different machines and compilers employ different notions of endianess as well as different sized data types, making the transfer and processing of files between machines a non-trivial task. However, during the mapping from storage to data model and vice versa, the HDF5 library accounts for these discrepancies. Using the self-describing information, the data is converted from the source to the target machines' architecture without any attention required from the user.

Additionally, HDF5 provides functionality for distributed memory systems. Parallel HDF5 routines are based on an underlying implementation of MPI-IO, whereas the HDF5 libraries manage the application's I/O calls and in turn utilise MPI-IO's routines, providing easy-to-use parallel I/O functionality.

A. Design

The most important aspect of designing an I/O kernel were fixing the desired functionality and subsequently determining on how the tree-structure within the HDF5 file is conceived. Further on, the content and shape of the datasets had to be settled to support the intended functionalities. Due to the fact that structure, functionality, and considerations concerning the implementation strongly influence each other, this process was revised throughout the whole development. The final outcome is based mainly on the following conditions.

To reduce the management effort and the overall load on the file system, a shared file approach is used, in which each participating process reads and writes to a single file. Also, in its current iteration, this output file supports the complete set of intended functionality since most of these have overlapping requirements. However, this is subject to be revised in future iterations of the kernel to allow users turning off unnecessary functions and, thus, reducing the amount of data in the file.

The following functionality is currently supported by the kernel and the output file:

- *output*
  The main purpose of the I/O kernel is to output snapshots of the running simulation at user defined intervals. These snapshots give a complete view of the topological grid structure as well as the computed cell values. The file structure and the I/O routines were conceptualised to achieve a write-out as fast as possible, resulting in a minimal impact on the overall execution time of the CFD code. Apart from good programming practice this is achieved by using hardware specific optimisation. For details see section IV-A.
- *checkpointing*
  Using any of the written simulation snapshots, the code is able to recreate the topological grid structure from the HDF5 file and resume computation. This prevents costly data loss after a crash or a power outage for example or allows for splitting time intensive computations into smaller parts, to better utilise the sparse and expensive

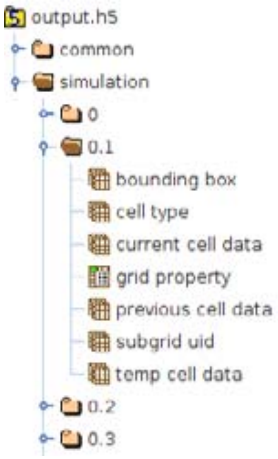

Fig. 5. Tree-like structure of the HDF5 output file

CPU hours on current high-performance machines. Also the codes' current domain decomposition and distribution strategy is done serially (by the neighbourhood server). To allow for an efficient usage of resources, this step can be prepared on a smaller machine while the simulation run is then started from the HDF5 file.

- *offline sliding window*
  The sliding window concept allows for visualisation during runtime using the neighbourhood servers complete view of the topological grid structure. The neighbourhood server is able to select a subset of simulation data of arbitrary resolution representing the desired section. This allows for an efficient limitation of data to transfer and display. However, this is possible for the currently computed time step only, i. e. online. Having the hierarchical grid structure also present within the snapshots of the HDF5 file allows for a sliding window scheme in a similar – offline – fashion. This enables users to visualise even largest datasets in a quick and efficient manner for every written time step.
- *time reversible steering*
  Using the client-server structure of the sliding window, one can in addition influence a running simulation in a computational steering approach. Combining the visualisation and restart functionality added on top of the HDF5 file, steering capabilities can be extended by also allowing the manipulation of previously computed time steps. Hence, the HDF5 file enables users to visualise the origins of developing flow characteristics, go back to earlier time steps in oder to change boundary conditions, and restart from here.

The current structure of the HDF5 file is illustrated in Fig. 5. Below the root group the structure splits into two branches. The *common group* encloses datasets that hold constant information such as the time discretisation step, refinement spacings, or fluid properties. It is therefore only accessed once at generation of the file or start of a simulation run. The *simulation group* holds further groups that enclose the structure information as well as the cell data itself for each time step. These time step groups are named with the elapsed time.

All datasets in the time step groups follow a simple paradigm. Every row in the two dimensional datasets corresponds to one grid in the codes' data structure. Additionally, the row indexes are constant across all datasets representing one time step, i. e. that the row index referring to one specific grid is identical in all datasets.

The grids are ordered by the respective ranks on which they reside in the code. Grids residing on rank 0 occupy row indexes starting from 0. Grids on subsequent ranks follow in the same manner. The root grid, that comprises the complete physical domain is by construction always at the top most position on rank 0. A very important consequence of the ordering is that the root grid will always be represented by index 0 in every dataset, providing the necessary starting point to traverse the hierarchical data structure.

The datasets *grid property*, *subgrid uid*, and *bounding box* make up the topological grid structure of the respective time step. *grid property* stores the Unique Identifier (*UID*) for every grid, encoding the residing rank, a rank unique identifier and its location in the structure. *subgrid uid* contains the *UID*s for grids created by refining the respective grid on the next finer level and *bounding box* encodes the physical extent of each grid.

The datasets *current cell data*, *previous cell data*, and *temp cell data* contain the data values for every cell of every grid such as velocities, pressure etc. *cell type* stores the boundary conditions. These last 4 datasets make up the vast majority of data in the file.

B. Implementation

All HDF5 functionality is encapsulated into one C++ class that each computing process instantiates. The current iteration of the kernel makes no use of any functionality of the neighbourhood server to avoid possible communication bottlenecks.

The first write creates the file and the aforementioned tree structure while subsequent writes only open the file and add the respective time step group and datasets.

In Parallel HDF5, the group structure as well as every dataset has to be created collectively by all participating ranks, while read and write operations can be carried out individually. To be able to generate the needed datasets, the total amount of grids in the complete domain must be known to all ranks. Additionally, to determine the non overlapping regions in the datasets for the read and writes of individual ranks, in HDF5 terminology so called hyperslabs, ranks must be aware of the cumulative amount of grids on previous ranks. This is achieved using a global MPI reduction, summing up all grids, followed by an MPI prefix reduction or scan to determine the amount added by all previous ranks to the global sum.

In the HDF5 storage model, each dataset is represented via a header followed by the actual data in form of a linear array, regardless of its actual dimensionality. The shape of the

dataset is defined by the header information. For optimised performance, a one to one mapping of data from the code to the HDF5 file is desirable. For this purpose, we chose to initialise a linear write buffer on each rank in which the grid data is copied. This additional storage requirement effectively cuts the amount of data to be handled by a single process in half. Since we emphasise on the added performance by this approach, the drawbacks of limiting the amount of data per rank was deemed acceptable.

Reading and restarting was conceived in a comparable fashion. Each rank reads the *UID*s in the *grid property* dataset to determine their range of grids according to the rank information encoded in the *UID*s. Each rank then generates the respective amount of grids, reads the data from its hyperslab, and copies it to the respective grids. The neighborhood server registers the topological structure and computation may commence. If restart from an intermediate snapshot is ordered, the I/O kernel creates a new branching file for subsequent write outs.

The sliding window approach in the code makes use of the ability of the neighbourhood server traversing the logical grid structure from the root downwards to subsequently refined grids. If a sliding window query is send to the neighbourhood server, it successively adds and removes d-grids to a list while traversing the tree until it has found the finest possible resolution fitting into a given limit of bandwidth and visualisation window. The sliding window on top of the HDF5 file uses the same approach of traversing the tree starting from the root grid at row index 0. This is achieved by assigning the *UID* information of a grid to its respective row index via the *grid property* dataset. Grids on subsequent refinement levels are found via the *subgrid uid* dataset. The routine ends up with a list of indexes referring to the grids that fit the determined criteria and allows for a selective visualisation of the corresponding grid data.

IV. Results

The following section highlights the performance measurements of the write routines of the kernel, the most performance critical aspect of the implementation as these are carried out continuously during a simulation run. The measurements were conducted on the JuQueen supercomputer located at Jülich Supercomputing Centre.

A. System Configuration

The JuQueen is an IBM BlueGene/Q system which combines 27,672 computing nodes and 458,752 cores in 28 racks. Each node employs a memory of 16 GB, amounting to 448 TB for the whole installation. The nodes communicate via a five dimensional torus intra-rack network. The systems theoretical peak performance is listed at 5.9 Petaflops, with a sustained Linpack performance at 5.0 Petaflops. For I/O, the system employs dedicated I/O nodes, grouped in I/O drawers installed in the top compartment of the racks. Each rack except the last one contains one I/O drawer with eight I/O nodes. Every I/O node has two PCIe ports for input and output connecting to the torus network, allowing 4 GB/s of raw data throughput one way. This sums up to a bandwidth of 32 GB/s per rack. Each I/O node is connected via two 10GbE Ethernet adapters to the file system, allowing a maximal bandwidth of 16 GB/s per I/O drawer. The underlying parallel file system is IBMs General Parallel File System (GPFS). A more comprehensive overview of the the JuQueen can be found in the official Documentation [16] and the best practice guide from PRACE [17].

To achieve optimised performance, the I/O kernel has to be aware of the underlying hardware and must be tuned accordingly. Most of the optimisations like alignment of data to the file system's block size lead only to comparably small improvements in write speed, however enabling collective buffering and disabling expensive file locking mechanisms of the GPFS have proven to be indispensable for the performance of the kernel.

Collective buffering utilises a subset of the computing nodes as aggregators, which collect data from the different processes and manage the file accesses. The JuQueen's node cabling qualifies exceptionally well for collective buffering. Each computing node employs 10 links to the intra-rack five dimensional torus network. However, only 16 of the 1024 nodes employ a single link to the I/O nodes. Doing independent I/O over the very scarce amount of I/O connections would lead to severe contention and minuscule performance. The natural choice for the aggregators are the nodes that employ the direct links to the I/O drawers. Data is collected over the very fast intra-rack network while the I/O links are utilised to their full extent.

To avoid contention by concurrent file accesses, the file driver of MPI-IO's current implementation on the JuQueen employs a very conservative file locking policy which poses detrimental to the performance of shared file approaches. Since each participating rank has its exclusive access region, it is safe to disable the file locking, thus leading to a tremendous increase in performance.

B. Measurements

To be able to classify our CFD code's I/O performance not only in terms of utilised bandwidth, we also compared *mpfluid* against another I/O kernel based on HDF5. As reference we employed the VPIC-IO (vector particle-in-cell) kernel [19] from ExaHDF used in the largest I/O run to date [20]. In order to get architecture independent measurements, comparable to the present implementation, the measurements using VPIC-IO were done similarly on the JuQueen, using the same optimisation and scaling the total amount of data for both kernels to be equal.

We used two test cases while varying the amount of computing processes involved. The first test case involved a fully refined 3D domain of resolution $1024 \times 1024 \times 1024$ (depth 6) leading to a total amount of about 300,000 d-grids in the domain. Each d-grid contains 16 cells in every dimension, making up 4096 cells per single d-grid and about 1.23 billion cells in the entire domain. Each written checkpoint claims a file size of 337 GB.

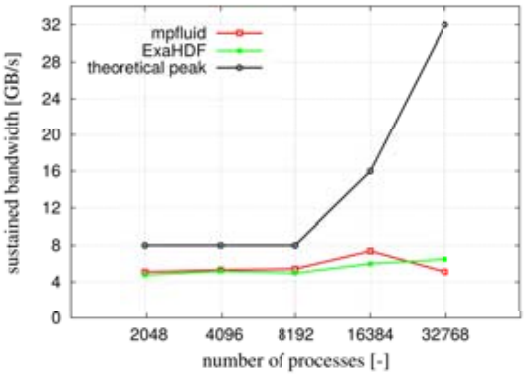

Fig. 6. Sustained bandwidth of a write out for a fully refined 3D domain of resolution $1024 \times 1024 \times 1024$ with approx. 11 billion unknowns and a total file size of 337 GB for different amounts of processes on the JuQueen

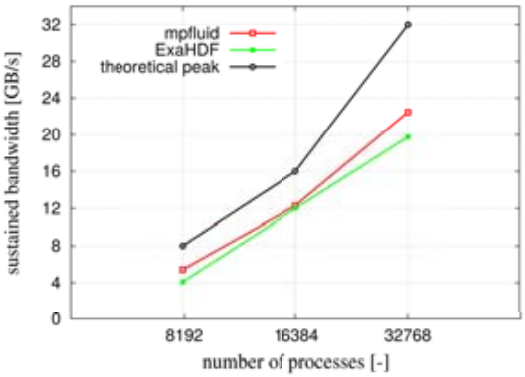

Fig. 7. Sustained bandwidth of a write out for a fully refined 3D domain of resolution $2048 \times 2048 \times 2048$ with approx. 90 billion unknowns and a total file size of 2,750 GB for different amounts of processes on the JuQueen

The second test case uses the same properties but resolves the domain one level further up to a resolution of $2048 \times 2048 \times 2048$ (depth 7), resulting in a total amount of about 2.4 million d-grids and 10 billion cells at a checkpoint size of 2.7 TB.

The refinement levels as well as the amount of cells per computational grid were taken in accordance to Frisch and Mundani [21], who determined the setup to perform very well for solving the underlying Navier Stokes Equations.

Fig. 6 shows excellent performance for both kernels in the range from 2048 to 8192 processes. The similar measurements are attributable to the equal I/O resources they have available. If no other application is using the I/O nodes, all four nodes, connected to one half of the rack are available within the intra-rack network. The discrepancy between the measured and the theoretical peak bandwidth is believed to be due to the wind up and wind down of write operations to individual datasets. Here more in-depth measurements are required. Using 16,384 processes and having a full drawer with 8 I/O nodes available the used bandwidth increases by about 20 %. Doubling the amount of processes and available I/O nodes again, reveals even worse scaling as only one forth of the estimated bandwidth compared to the first three measurements is achieved. We believe this is attributable to the amount of grids per process on the later test cases. While each individual process has fewer grids to manage, the communication overhead of filling the aggregators' write buffers increases, which is likely to be responsible for the bad scaling behaviour. Additionally, more aggregators, assuming less data each participate in I/O, certainly affects the performance negatively. On top of the fact that the problem is too small to show adequate scaling in the regions above 8192 processes for I/O performance, the same was observed for the actual computation in [21].

For the second test case, measurements were not possible below 8192 processes due to the aforementioned memory limitation introduced by the additional write buffers. Fig. 7 shows the measured results. As expected, the measurements show adequate scaling in the expected range, both for VPIC-IO's and our kernel.

V. Application

In order to show the possible applications for the time reversible steering functionality on top of the HDF5 output file we conducted a benchmark scenario initially proposed by Schäfer and Turek within the DFG priority research program 'Flow Simulation on High-Performance Computers' [18]. The setup consists of a two dimensional channel flow with a cylinder obstacle near the inlet on the left-hand side channel boundary. Using Reynold's numer $Re = 100$, an unsteady flow is generated which leads to the well known phenomena of vortex shedding behind the obstacle. We simulated the basic scenario for two seconds, went back to the first second mark, and altered the boundary conditions of the setup before restarting the simulation again. Fig. 8 shows visualisations at $t = 1.0\,\mathrm{s}$ and $t = 2.0\,\mathrm{s}$ of the basic setup as well as two cases with altered boundary conditions. We want to emphasise that these are not separate simulations but rather branchings within the framework.

VI. Conclusion

In this paper, we have presented a massive parallel CFD code that was successfully deployed on two of Germany's supercomputers, runnning on about 140,000 cores, solving a problem with more than 700 billion unknowns. Core of this work is a hierarchical data structure that not only supports distributed computing and the development of efficient numerical solvers, but further allows users an in situ visualisation (sliding window) of the computed results in order to leverage high-performance interactive data exploration. This code was now extended by an HDF5 I/O kernel to tackle the problem of reading and writing huge datasets during runtime, a typical

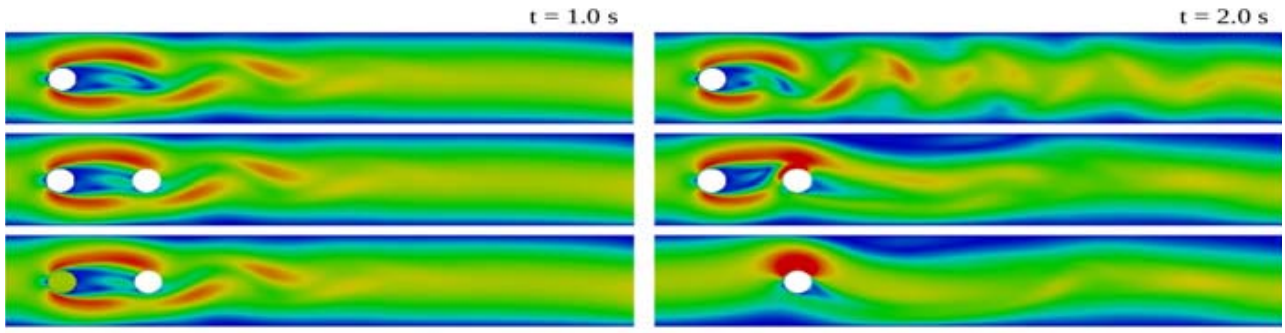

Fig. 8. Time reversible steering: comparative visualisation of a von Kármán vortex street according to [18] at elapsed time $t = 1.0\,\mathrm{s}$ and $t = 2.0\,\mathrm{s}$ for the basic setup and two altered scenarios – going back to a previous time step and modifying boundary conditions – restarting at $t = 1.0\,\mathrm{s}$

bottleneck for modern HPC applications, thus slowing down the overall performance due to long I/O latencies.

The current I/O kernel – using collective buffering for optimisation – shows a good scaling behaviour within the applicable range of grids, i. e. any limiting factor is due to the specifically used I/O hardware, so that with an increasing amount of I/O nodes a much better performance is to be expected. Further comparisons with ExaHDF's I/O kernel foster these results. By adopting above data structure for the design of the kernel's internal file structure, additional functionality such as a file-based sliding window could easily be implemented, now providing users the opportunity for a time reversal steering, i. e. to change boundary conditions and restart the computation at any previous time step.

Acknowledgment

The authors would like to cordially thank Leibniz Supercomputig Centre (LRZ) and Jülich Supercomputing Centre (JSC) for providing computing time on SuperMUC and JuQueen, resp., in order to carry out all computations and measurements during our researches. Without their kindly support, parts of this work would not have been possible.

References

[1] T. Kalil and J. Miller. (2015) Advancing U.S. Leadership in High-Performance Computing. [Online]. Available: https://www.whitehouse.gov/blog/2015/07/29/advancing-us-leadership-high-performance-computing

[2] M. Bader, H.-J. Bungartz, A. Frank, and R.-P. Mundani, "Space tree structures for PDE software," in *Computational Science*, ser. LNCS 2331, P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, Eds. Springer, 2002, pp. 662–671.

[3] R.-P. Mundani, J. Frisch, V. Varduhn, and E. Rank, "A sliding window technique for interactive high-performance computing scenarios," *Advances in Engineering Software*, vol. 84, pp. 21–30, 2015.

[4] M. Rivi, L. Calori, G. Muscianisi, and V. Slavnic, "In-situ visualization: State-of-the-art and some use cases," in *PRACE White Paper*. Belgium: PRACE Brussels, 2012.

[5] J. Ferziger and M. Perić, *Computational Methods for Fluid Dynamics*, 3rd ed. Springer, 2002.

[6] C. Hirsch, *Numerical Computation of Internal and External Flows*, 2nd ed. Butterworth–Heinemann, 2007, vol. 1.

[7] J. Lienhard IV and J. Lienhard V, *A Heat Transfer Textbook*, 4th ed., ser. Dover Civil and Mechanical Engineering. Dover Publications, 2011.

[8] F. Harlow and J. Welch, "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface," *Physics of Fluids*, vol. 8, no. 12, pp. 2182–2189, 1965.

[9] A. Chorin, "Numerical solution of the navier-stokes equations," *Mathematics of Computation*, vol. 22, no. 104, pp. 745–762, 1968.

[10] J. Frisch, R.-P. Mundani, and E. Rank, "Communication schemes of a parallel fluid solver for multi-scale environmental simulations," in *Proc. of the 13th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE Computer Society, 2011, pp. 391–397.

[11] J. Frisch, "Towards massive parallel fluid flow simulations in computational engineering," Ph.D. dissertation, Technische Universität München, 2014.

[12] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Mathematics of Computation*, vol. 31, no. 138, pp. 333–390, 1977.

[13] J. Frisch, R.-P. Mundani, and E. Rank, "Parallel multi-grid like solver for the pressure Poisson equation in fluid flow applications," in *Proc. of the IADIS Int. Conf. on Applied Computing*. IADIS Press, 2013, pp. 139–146.

[14] ParaView. [Online]. Available: http://www.paraview.org

[15] The HDF Group. HDF5 Software Documentation. Release 1.8.16. [Online]. Available: https://www.hdfgroup.org/HDF5/doc/

[16] JuQueen – Documentation. [Online]. Available: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/Documentation/Documention_node.html

[17] P. Wautelet, M. Boiarciuc, J.-M. Dupays, S. Giuliani, M. Guarrasi, G. Muscianisi, and M. Cytowski, *Best Practice Guide - Blue Gene/Q v1.1.1*, PRACE - Partnership for Advanced Computing in Europe, 2014.

[18] M. Schäfer, S. Turek, F. Durst, E. Krause, and R. Rannacher, "Benchmark computations of laminar flow around a cylinder," in *Flow Simulation with High-Performance Computers II*, ser. Notes on Numerical Fluid Mechanics, vol. 52. Vieweg, 1996, pp. 547–566.

[19] S. Byna and M. Howison. (2015) Parallel I/O Kernel (PIOK) Suite. [Online]. Available: https://sdm.lbl.gov/exahdf5/software.html

[20] S. Byna, A. Uselton, Prabhat, D. Knaak, and Y. He, "Trillion Particles, 120,000 cores, and 350 TBs: Lessons Learned from a Hero I/O run on Hopper," in *Cray User Group meeting*, 2013.

[21] R.-P. Mundani and J. Frisch, "Measuring and comparing the scaling behaviour of a high-performance CFD code on different supercomputing infrastructures," in *Proc. of the 17th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE Press, 2015.

[22] R. Thakur, R. Ross, E. Lusk, W. Gropp, and R. Latham, "User's guide for ROMIO: A high-performance, portable MPI-IO implementation," ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Tech. Rep., 2010.

[23] Top 500 Supercomputing Sites. Accessed: 2016-03-25. [Online]. Available: http://www.top500.org