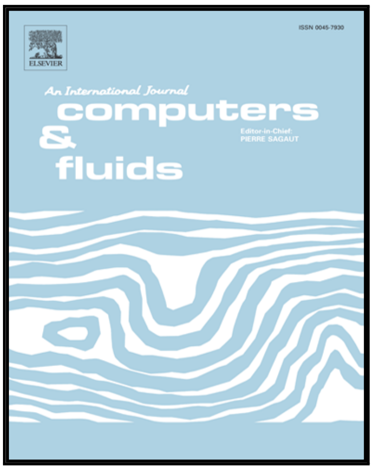

Load balance and Parallel I/O: Optimising COSA for large simulations

Adrian Jackson , M. Sergio Campobasso , jernej Drofelnik

PII: S0045-7930(18)30105-1
DOI: 10.1016/j.compfluid.2018.03.007
Reference: CAF 3768

To appear in: *Computers and Fluids*

Received date: 9 February 2017
Revised date: 12 February 2018
Accepted date: 1 March 2018

Load balance and Parallel I/O: Optimising COSA for large simulations

Adrian Jackson[a,*], M. Sergio Campobasso[b], Jernej Drofelnik[c]

[a]*University of Edinburgh, EPCC, JCMB, Kings Buildings, EH9 3FD Edinburgh, UK*

[b]*University of Lancaster, Department of Engineering, Engineering Building, LA1 4YW Lancaster, UK*

[c]*University of Southampton, Department of Aeronautics, Astronautics and Computational Engineering*, SO17 1BJ Southampton, UK

Abstract

This paper presents the optimisation of the parallel functionalities of the Navier-Stokes Computational Fluid Dynamics research code COSA, a finite volume structured multi-block code featuring a steady solver, a general purpose time-domain solver, and a frequency-domain harmonic balance solver for the rapid solution of unsteady periodic flows. The optimisation focuses on improving the scalability of the parallel input/output functionalities of the code and developing an effective and user-friendly load balancing approach. Both features are paramount for using COSA efficiently for large-scale production simulations using tens of thousands of computational cores. The efficiency enhancements resulting from optimising the parallel I/O functionality and addressing load balance issues has provided up to a 4x performance improvement for unbalanced simulations, and 2x performance improvements for balanced simulations.

Keywords: COSA, Load Balance, Decomposition, Optimisation, Parallel Performance, I/O

1 Introduction

The harmonic balance (HB) method is a nonlinear frequency-domain technique that reduces the runtime for calculating periodic solutions of ordinary differential equations with respect to the conventional time-marching approach. The reduction occurs because the HB method, unlike the conventional time-domain approach, determines directly the periodic solution of interest, bypassing lengthy transients. One of the pioneering

[*] Corresponding Author
Email address: a.jackson@epcc.ed.ac.uk

applications of the HB technology in Navier-Stokes (NS) Computational Fluid Dynamics (CFD) was reported in [1], a study presenting and demonstrating the HB NS technology for the rapid solution of turbomachinery unsteady periodic flows. In classical HB formulations, the sought periodic solution is expressed with a truncated Fourier series, as the sum of a user-given number *Nh* of complex harmonics. Using this approach, one has ($2Nh+1$) unknown harmonic flow components corresponding to the real and imaginary parts of N_h complex harmonics and the mean flow field. The implementation of the HB method in CFD codes, however, can be notably simplified by projecting the nonlinear frequency-domain equations back to the time-domain. By doing so, one has to compute ($2Nh+1$) snapshots of the sought periodic flow field by solving a coupled system of ($2Nh+1$) steady problems [1].These snapshots are equally spaced in time with a step $\Delta t=T/(2Nh+1)$, where T is the user-given period of the considered periodic excitation or forcing term.

In aerodynamic performance, aeroelastic and aero-acoustic assessments, the use of the HB NS technology rather than the conventional time-domain (TD) NS method to accurately determine periodic flows of turbomachinery blade rows, vibrating aircraft wings, and helicopter rotors was shown to reduce runtimes by one to two orders of magnitude [1].

In the case of bladed rotors, the HB speed-up is particularly high due to the possibility of using multi-frequency periodic boundary conditions enabling one to model the flow past a single blade rather than the whole rotor [3]. The HB NS COSA solver is pioneering the development and exploitation of this technology in wind turbine engineering worldwide.

In the past 10 years, the use of the HB NS technology has greatly reduced the runtimes of complex unsteady aerodynamic, aeroelastic and aeroacoustic NS simulations[1] [3] [4]. This is substantially strengthening turbomachinery, helicopter and aircraft design technologies, due to the possibility of modelling (and thus understanding and controlling) unsteady flow mechanisms that previously could not be modelled due to unaffordable runtimes of conventional time-domain simulations [2].

A larger scale deployment of the HB NS technology in the aeronautical and turbomachinery industry, and in new sectors such as renewable energy fluid machinery engineering (e.g. wind and tidal current turbine design and verification) is expected in the next 5 to 10 years, but this will require further research into serial and parallel aspects of the HB NS technology.

Large scale deployment of simulation codes requires scaling to large numbers of computational cores. It is at this stage that parallel applications often run into performance issues. Scaling to large numbers of cores requires all parts of an application to be efficiently parallelised, and work to be evenly decomposed across workers.

This paper outlines work undertaken [5] to take a parallelised HB CFD application, COSA [6], and enable it to scale efficiently to large numbers of cores. COSA is a finite volume NS code featuring a steady solver, a TD solver for the solution of general unsteady flows [7] [8], and a HB solver for the rapid solution of highly nonlinear unsteady periodic flows [9] [10][11]. The general parallelisation of COSA is efficient and scales well, but the load balance and data input/output (I/O) functionality did not match the rest of the application and stood in the way of large scale research deployment.

Section 2 introduces the COSA code and its initial parallel performance characteristics. Sections 3 and 4 detail the performance of the original code and the work undertaken to optimise parallel I/O and load balancing functionality. A summary of the main findings of this work is reported in the closing Section 5**Error! Reference source not found.**.

2 COSA

COSA is a structured multi-block NS code featuring a steady, a TD and a HB solver, all using a finite volume space-discretisation and an efficient multigrid integration. All three solvers are parallelised using MPI [12] and the HB solver also with a hybrid MPI/OpenMP [13] paradigm which substantially reduces the runtime of the HB solver by enabling the use of more cores than possible with its pure MPI counterpart.

COSA exhibits good parallel performance, in general, but a number of areas of performance can be improved. Firstly, COSA uses parallel I/O to read input data (mesh and restart files), and write output data (restart and other large files for postprocessing) and checkpoint files (convergence history files), but this I/O was not structured in the most efficient way. This is demonstrated in Figure 1, which shows the parallel performance of a representative simulation (NACA0015_HB_plu_mb16384, a HB test case using 4 complex harmonics and a

16,384-block grid with 37,748,736 cells, as outlined in Section 2.1) with and without output I/O enabled (with all data referring to the code performance prior to the optimisation work reported herein). I/O wall-clock time is a significant portion of the overall runtime for the large core counts required for complex 3D simulations. It is also noted that the tests of Figure 1 were run using only 10 iterations. Since the code needs to read the mesh file in all cases, the scalability highlighted by the curve labelled ‘MPI scaling’, which has all I/O turned off except for the mesh read, is reduced by the poor scaling of this I/O operation at large core counts. Indeed, it was previously found that the code scalability for this test case using 400 iterations and turning off all I/O except for the mesh read was fully linear until 16,384 cores, the maximum core count used for this test case.

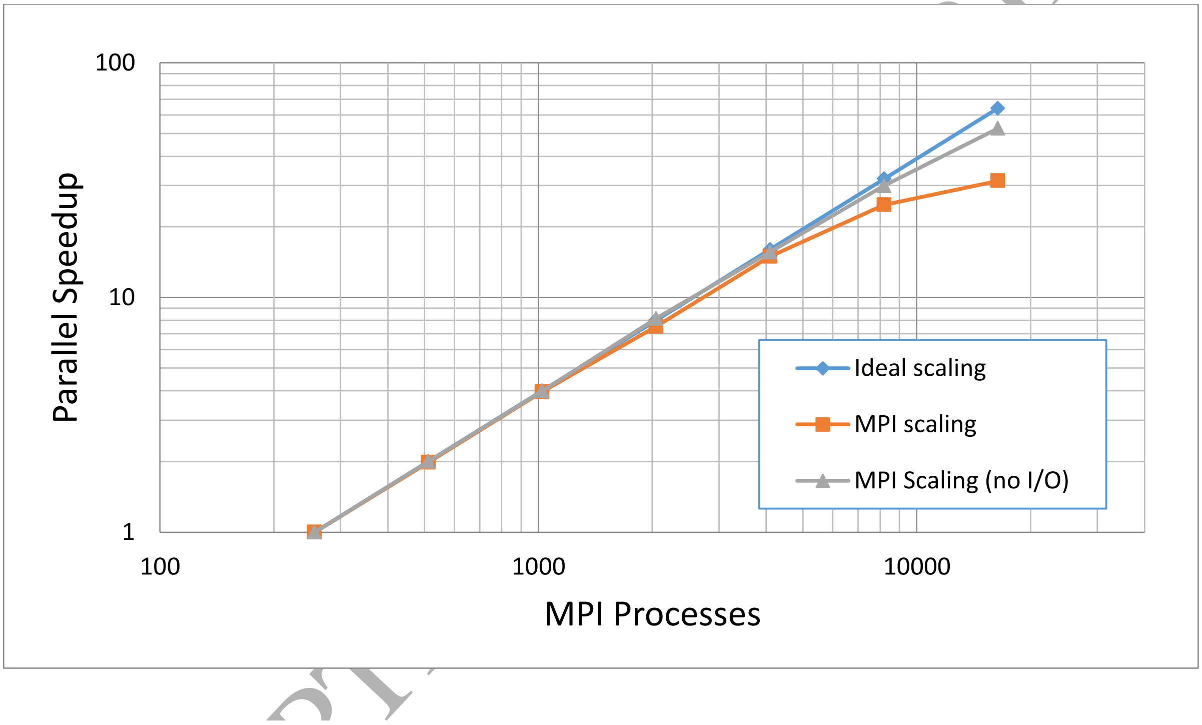

Figure 1: Parallel scaling of COSA with and without output I/O enabled

To achieve a scalability of the I/O operations comparable to that of the computing part, the code I/O needs to be restructured and re-parallelized making more efficient use of MPI-I/O [14] routines.

Secondly, COSA currently relies on a pre-defined data decomposition for the parallel decomposition. As COSA is a structured multi-block code, grid blocks are the standard unit for data decomposition across processes. Individual grid blocks are not split within COSA, as this would require significant functionality to ensure that geometry and boundary conditions are correctly handled. Instead, the existing load-balancing approach balances the number of grid blocks assigned to processes, aiming to have an equal number of blocks across processes (or as equal as possible depending on the number of blocks in the simulation grid and the number of processes used). Thus, the load-balance of the parallel simulation relies on the assumption that all grid blocks have the same or a similar number of grid cells, one of the main parameters on which the computational work associated with performing one solution iteration for each block depends.

The constraint of equal block sizes results in a significant additional effort for the user to generate the block decomposition (i.e. the multi-block grid) for the simulation. In the case of complex geometries such as wind turbine rotors [11], state-of-the-art multi-block grid generators lack the flexibility needed to easily generate multi-block grids whose blocks have the same or at least comparable number of cells, and instead yield grids whose blocks differ significantly in size. Thus, additional lengthy grid pre-processing operations, often requiring the use of several different grid pre-processors, are needed to obtain a grid with equal block sizes. To improve the code ease-of-use by removing additional grid pre-processing steps without damaging the CFD code parallel scalability, a dynamic parallel load-balancing capability is required, ensuring the number of grid blocks (geometric partitions) assigned to each MPI process takes into account the largely unequal block sizes of the default output grid of the mesh generation software. This enables the assignment of varying numbers of potentially unequal blocks to individual processes, ensuring a comparable amount of work for all processes running the simulation.

Enabling the parallel code to carry out balanced simulations even when the sizes of the input grid blocks differ significantly, and to do so at runtime, simplifies and speeds up the grid generation phase, presently constrained by the requirement that the simulation grid be split into blocks of equal, or nearly equal, size, and that this be done at a pre-processing stage.

2.1 Initial performance

In this section performance and scaling of COSA on representative simulations are presented. Performance is evaluated using two different benchmark cases, both using the harmonic balance solver, one with 3,144,712 grid cells decomposed into 800 grid blocks (NREL5MW_GRID32_HB_SECTOR), the other with 37,748,736 grid cells decomposed into 16,384 blocks (NACA0015_HB_plu_mb16384).

The former benchmark case, similar to the analyses in [11], refers to the simulation of the unsteady periodic flow field past a wind turbine rotor in yawed wind retaining four complex harmonics in the truncated Fourier series used to approximate the sought periodic flow field. The latter benchmark case refers to the analyses of the unsteady periodic flow field past an oscillating wing (concurrent plunging and pitching motion) for renewable power generation, it uses four complex harmonics, and it is similar to the TD analyses reported in [8].

A third benchmark case (NACA0015_Ogrid_UNBALANCED) is used to test the load balancing functionality and performance issues. This test case refers to the HB analysis of the periodic flow field past a plunging wing, and retains four complex harmonics in the Fourier series representation of the sought periodic flow field. The simulation uses a grid with 783,368 cells, and the grid is split into 256 blocks, with two different decompositions available, one where blocks have been carefully constructed to ensure they have similar numbers of grid cells, and the other where the block sizes are taken directly from a grid generation package and differ significantly. In the unbalanced grid, there is a difference of about 7x between the number of cells of the largest and smallest blocks.

Testing and benchmarking in this paper has been performed on the UK National HPC Service, ARCHER[15]. This is a 118,080 core Cray XC30 system, with 24 cores (two Intel Xeon 2.7 GHz, 12-core E5-2697v2 processors) and 64 GB per node, and the Cray Aries network. All benchmark data in this document were collected using a version of COSA compiled with the Intel compiler (version 15.0.2.164) on ARCHER using Intel's Maths Kernel Library (mkl) (version 11.2.2) [16].

When benchmarking on ARCHER, the parallel filesystem (Lustre [17]) is configured with a maximum stripe count (count of -1) for the directories each simulation is run from. This ensures that parallel I/O functionality within an application can use all the available I/O resources (I/O servers) available in the filesystem being used.

For benchmarking the results presented in this paper, each simulation is run three times and the fastest run is presented in the graphs. For all the collected data, the difference between the fastest and slowest runs was less than 5%, and, for this reason, data variability (error bars) are omitted from the graphs of the paper.

Given the initial MPI decomposition functionality in COSA was designed to distribute blocks across processes as evenly as possible the following is a list of sensible process counts for the three benchmark cases we have outlined:

- NACA0015_Ogrid_UNBALANCED: 256 blocks: 32, 64, 128, 256
- NREL5MW_GRID32_HB_SECTOR: 800 blocks: 20, 40, 80, 100, 200, 400, 800
- NACA0015_HB_plu_mb16384: 16384 blocks: 1024, 2048, 4096, 8192, 16384

The 256 and 800 block simulations will run on a single node of ARCHER, but the 16,384 block simulation is too large to fit into the memory on a single node so benchmarking is started from 1024 cores (43 nodes).

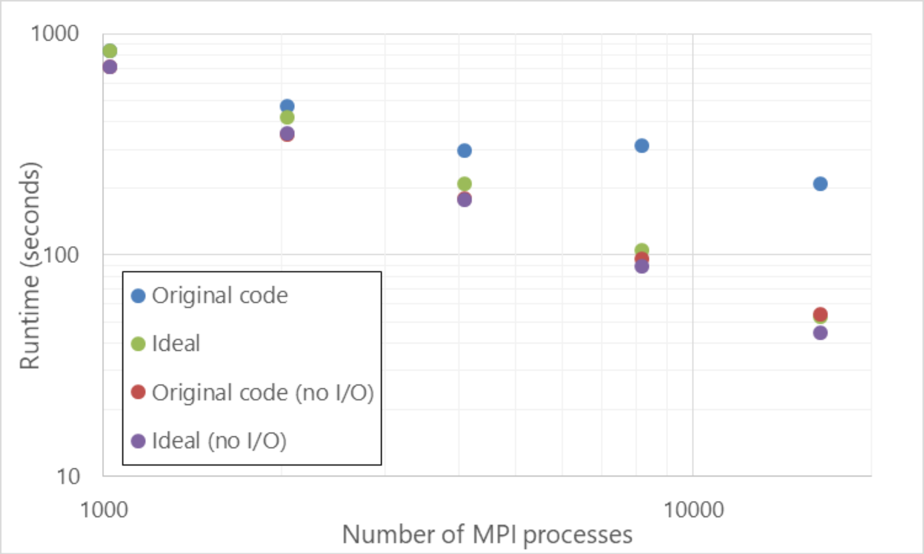

Figure 2: Runtime for 100 iterations of 16,384 block simulation, with and without output I/O enabled

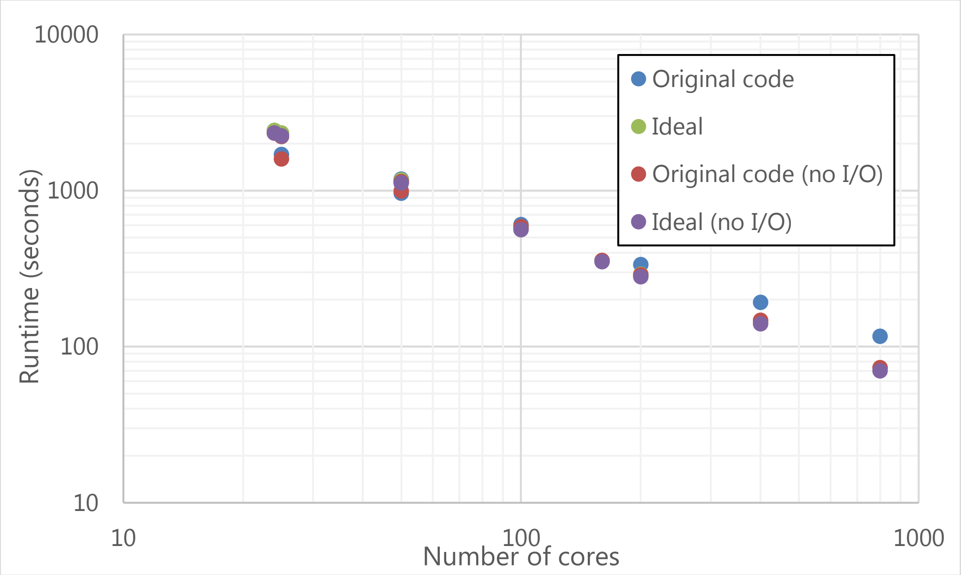

Figure 3: Runtime for 100 iterations of 800 block simulation, with and without output I/O enabled

It is evident from Figure 2 and Figure 3 that COSA scales very well, even up to the maximum block counts, when writing output data is disabled. However, especially for the large test case, it is evident that I/O really dominates performance, and even for the smaller simulation the I/O is costing around 40 percent of the runtime for the highest core counts.

It should be noted that simulations are only run for small numbers of iterations of the algorithm for these benchmark cases, meaning I/O output will have a larger impact of performance than when a normal, production, simulation is undertaken (where thousands of iterations are used). The particular choice of the number of iterations is made to illustrate the issue with I/O in these simulations.

The other feature of simulation codes, like COSA, that often needs to be optimised when scaling to large production jobs, is the load balance of the domain decomposition approach used. Therefore, the code is also benchmarked using a simulation (NACA0015_Ogrid_UNBALANCED) where load balance is an issue.

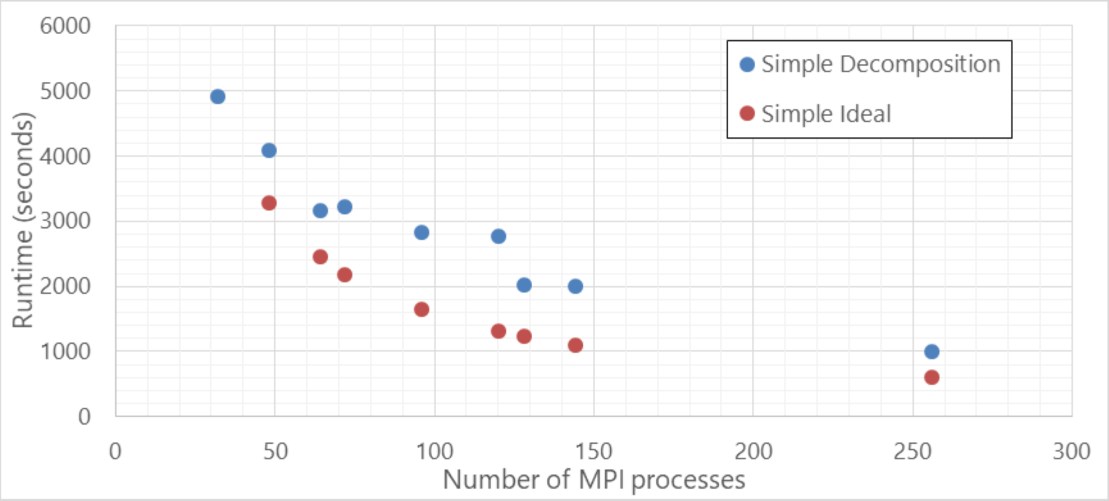

Figure 4: Runtime for 1000 iterations of 256 unbalanced block simulation, with output I/O enabled.

Figure 4 outlines the parallel performance of this simulation alongside the ideal curve (which is calculated by taking the time on 32 cores and dividing the runtime by 2 every time the number of cores doubles). It is evident from the graph that there is a significant loss of parallel performance when running with an unbalanced decomposition using the current data decomposition functionality in COSA.

3 I/O Optimisations

The current I/O functionality in COSA is parallel, at least on output, with a solution file (*restart* file) produced with user-given iteration frequency for checkpoint and restart purposes, and one or more large data files produced at the end of the simulation for flow visualisation purposes (*flowtec* files). Additional small files reporting residual and force convergence histories are also produced with user-given iteration frequency throughout the iterative solution process. On the input side, the code reads in a relatively large file containing the coordinates of the entire grid (*mesh.dat* file), a very small control file (*input.dat*) containing physical and numerical control parameters and the block and boundary condition connectivity data, and, for restarted jobs, also the restart file mentioned above.

These large output files (restart and flowtec files) are written on a per-block basis, with data from consecutively ordered blocks stored adjacently in the files. As such, each process can write the blocks it owns to different parts of the output file in parallel without interfering with each other. This means I/O is undertaken in parallel, but without using collective MPI-I/O functionality, because each process may have different numbers of blocks, or different sized blocks.

The existing parallel I/O is structured to replicate serial I/O, enabling files written by the parallel code to be read by the serial version of the code, or the parallel code to read restart files generated by the serial version of the code. The Fortran I/O functionality used in the serial code exploited the unformatted Fortran file format. This is a binary file format where each line of data is preceded and finished by a record of the number of bytes stored on the line.

For the large simulation considered in this paper (16,384 blocks), the restart file is of the order of 40GB and there are nine flowtec files, each around 6GB in size. The smaller simulation (800 blocks) has a restart file of approximately 3 GB with nine flowtec files of around half a GB each.

In the case of the restart file, the *Nh* solution snapshots `q` at the centres of the cells of each grid block is written by the serial code with the instructions in Figure 5, which are executed within a loop over all grid blocks. The parameter `npde` denotes the number of partial differential equations (PDEs), and equals 7, corresponding to one PDE for the conservation of mass, three PDEs for the momentum conservation in all three directions, one PDE for the conservation of energy and two PDEs for the shear stress transport turbulence model used by COSA [[8] [11].

```
do n = 0,2*Nh
 write(fid)((((q(i,j,k,ipde,n),i=-1,imax1),j=-1,jmax1),k=-
1,kmax1),ipde=1,npde)
end do
```

Figure 5: Original serial restart output code

This has been translated into parallel I/O functionality using MPI-I/O as outlined in Figure 6, and requires `npde*(kmax1+2)*(jmax1+2)` MPI-I/O operations for each harmonic in the simulation, plus two extra operations to write the line lengths before and after the data, with each operation adding a performance overhead to the I/O. The routine `setupfile` simply moves the file pointer for a given process to ensure they write the data at the correct place in the file (using the `MPI_FILE_SEEK` routine).

```
do n = 0,2*Nh
 call setupfile(fid,disp,MPI_INTEGER)
 call mpi_file_write(fid, linle, 1, MPI_INTEGER, MPI_STATUS_IGNORE, ierr)
 disp = disp + integersize
 do ipde=1,npde
  do k=-1,kmax1
   do j=-1,jmax1
    call setupfile(fid,disp,MPI_DOUBLE_PRECISION)
    call mpi_file_write(fid, q(-1,j,k,ipde,n), imax+3,
&   MPI_DOUBLE_PRECISION,MPI_STATUS_IGNORE, ierr)
    disp = disp + doublesize*(imax+3)
   end do
  end do
 end do
 call setupfile(fid,disp,MPI_INTEGER)
 call mpi_file_write(fid, linle, 1, MPI_INTEGER, MPI_STATUS_IGNORE, ierr)
 disp = disp + integersize
end do
```

Figure 6: Original parallel restart output code

The restart file includes the halo data from the blocks (the extra data in the data arrays used to store data from adjacent blocks required for the simulations), hence the array indices spanning from, for instance, `-1` to `imax1` rather than `1` to `imax` (where `imax1=imax+1` and `imax` is the number of grid points in the `i`-direction). Since the number of geometric cells in the `i`-direction is `imax`, the `i`-loop including two halo cells on both sides of the i-interval has starting and ending indices `-1` and `imax1` respectively. Saving the halo data in the restart file is not strictly necessary, but doing so simplifies the internal structure of the code section carrying out the initialisation of the flow variables associated with the two-equation *k-ω* Shear Stress Transport turbulence model used by COSA [7], and has negligible impact on both the serial and parallel code performance. Conversely, the halo data is not written to the flowtec files, since this information is irrelevant to flow visualisation and additional solution post-processing.

The pseudo-code in Figure 6 highlights that, for both the restart and flowtec files, far more MPI-I/O operations are undertaken than is required; maintaining compatibility with the serial file format used by COSA is impacting I/O performance. It is also noted that non-collective MPI-I/O functionality is being used. MPI-I/O has the potential to provide more efficient I/O if collective I/O functionality is used (where all processes write the same amount of data to the file at the same time).

Ideally, this would be implemented in COSA to improve I/O performance. However, as each block owned by a process may be different in size, and each process may have a different number of blocks, it is not straight forward to write collective I/O operations to do this.

There is a possibility of defining MPI datatypes for each block, and then write each block in a single operation using collective MPI-I/O routines, but some global book keeping would be required to ensure each process has the same number of blocks and to revert to non-collective routines for non-matching numbers of blocks, which would add communication overheads to the I/O operations. Therefore, it was decided to continue using non-collective I/O functionality but to optimise the way I/O is performed to improve performance, as discussed in further detail in the next section.

The first I/O optimisation implemented was to move from writing each row of the restart file data in a single I/O operation to writing each harmonic for a whole block in a single I/O operations. Some of the flowtec header data (used by CFD post-processors) was also removed for each block, retaining only what is required to re-construct the data file in a post-processing step independent of the simulation, and reducing the flowtec file writing code.

Whilst the flowtec files still require almost the same amount of data to be written, the number of MPI-I/O operations required to output this data has been halved. Likewise, the same size of restart data is still being written, but there has been a dramatic reduction of the number of operations required to perform this I/O.

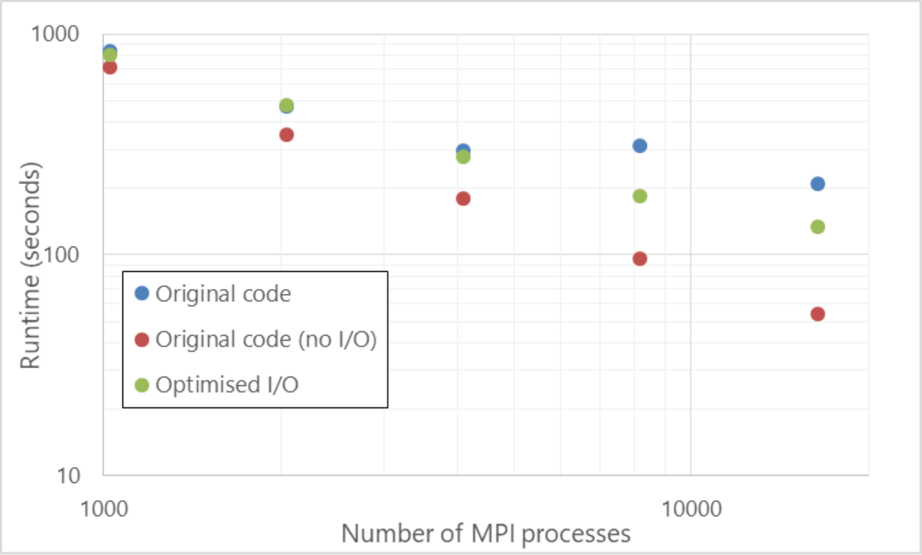

Figure 7: Performance of 16384 block simulation with optimised I/O (100 iterations)

The performance of the optimised I/O is shown in Figure 7 for the NACA0015_HB_plu_mb16,384 benchmark case. It can be seen that the impact of optimising the I/O, removing large numbers of I/O operations, is not significant at lower core counts, but at higher core counts the optimised I/O has a significant impact. The code is around 70% faster at 8,192 cores and around 50% faster at 16,384 cores.

The optimised I/O functionality was also benchmarked with the smaller test case, as shown in Figure 8. Note for this benchmark the number of iterations of the simulation was reduced from 100 to 20 to highlight the I/O costs in the simulation.

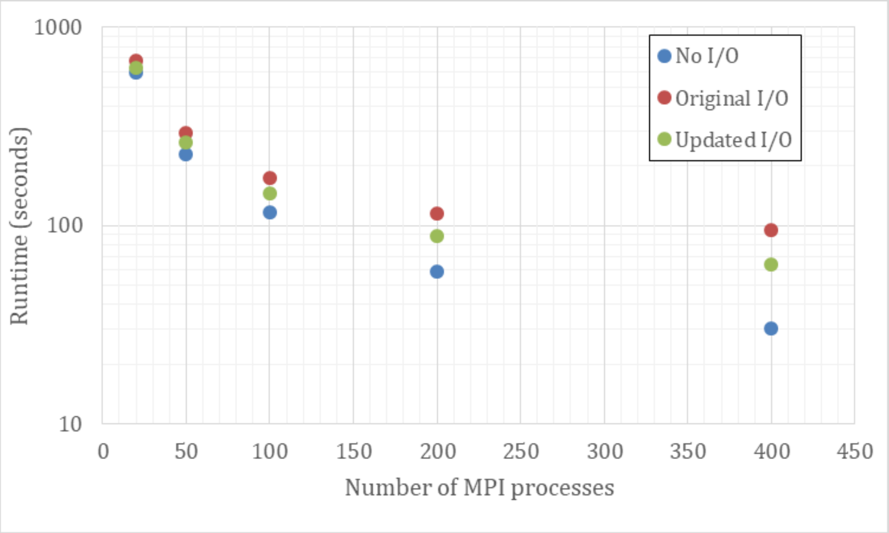

Figure 8: Performance of 800 block simulation with optimised I/O (20 iterations)

As with the larger test case, one notes that the optimised I/O is faster, around 50% when using 400 MPI processes (400 cores). However, it is observed that I/O is still expensive, with the code performance when not producing output data still significantly better than when I/O is undertaken.

To further optimise the I/O performance of COSA, the way the flowtec files are written was restructured. Firstly, the per-block header data has been completely removed, replacing it with a per-file header, written by one process, including all the block sizes at the start of the file. This reduces the number of write operations per-block to a single write.

Furthermore, it was also recognised that the code was structured in such a way that the data to be written to the flowtec files was first collected together into a large temporary array, requiring all blocks owned by a process to be iterated through, data collated into this temporary array, then passed to the I/O routines for output. However, this means that all block data is iterated over once (to construct the temporary array), then the I/O routines iterate over that temporary array one block at a time, copy that data into another temporary array and then write that out for each block.

Therefore, a new output routine was constructed that does not do the initial collection of data to be written into a large temporary array, instead it takes in all the source data for the flowtec files, and collects it into a temporary array one block at a time, which is then immediately written out to the flowtec file.

This has two benefits, firstly there is no longer a need for a temporary array large enough to store all the data for all the blocks a process owns, instead the temporary array only needs to be large enough to store the data for a single block. Secondly, the re-use of data that has been collected into the temporary array is enabled, optimising cache usage and therefore reducing computational costs.

Finally, it was also recognised that whilst the majority of I/O time is attributable to outputting data, when scaling to large numbers of MPI processes, the reading of the mesh required for the simulation can be an overhead. The reading of the mesh was originally done using serial Fortran I/O, although each process only reads the sections of the file they require for the blocks they have been assigned.

The serial reading of the input mesh was limited in performance by automatic file locking that was undertaken when a given process read the file. Whilst this locking (ensuring exclusive access to the file for a single MPI process) was not a large issue with small process counts, when large numbers of MPI processes were trying to read the same file this resulted in reductions in the performance of this initial I/O operation. Therefore, the reading of the mesh file has been parallelised using MPI-I/O, removing this file locking and the associated process synchronisation.

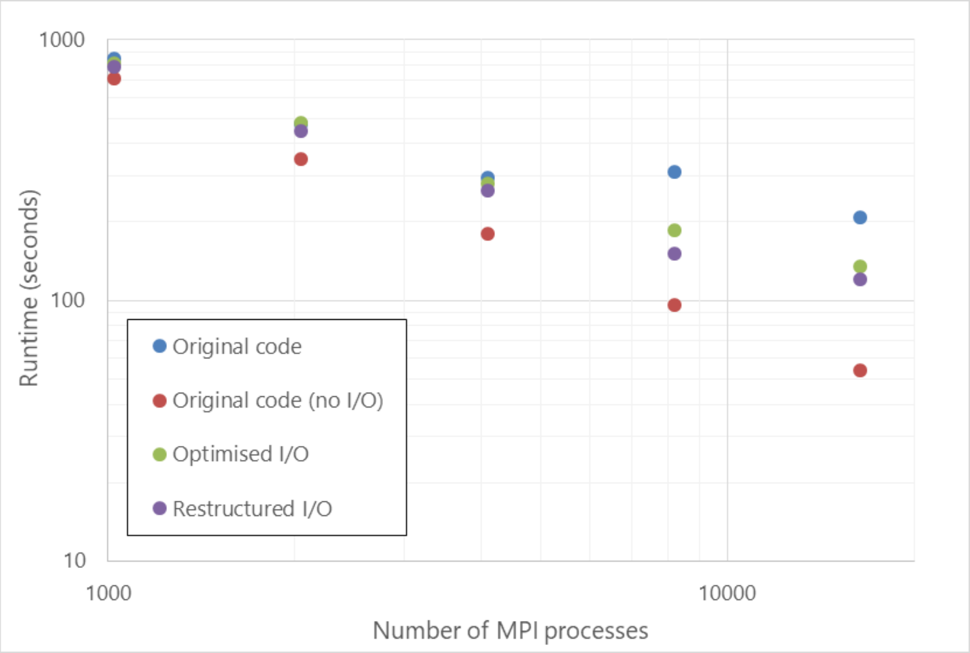

Figure 9: Performance of 16384 block simulation with restructured I/O (100 iterations)

Figure 9 and Figure 10 show the performance of the restructured and optimised I/O functionality. It is evident that these optimisations have further improved performance for COSA, with the code now around 100% faster on 8,192 cores and 70% faster on 16,384 cores with the large test cases, and 70% faster on 400 cores for the small test case.

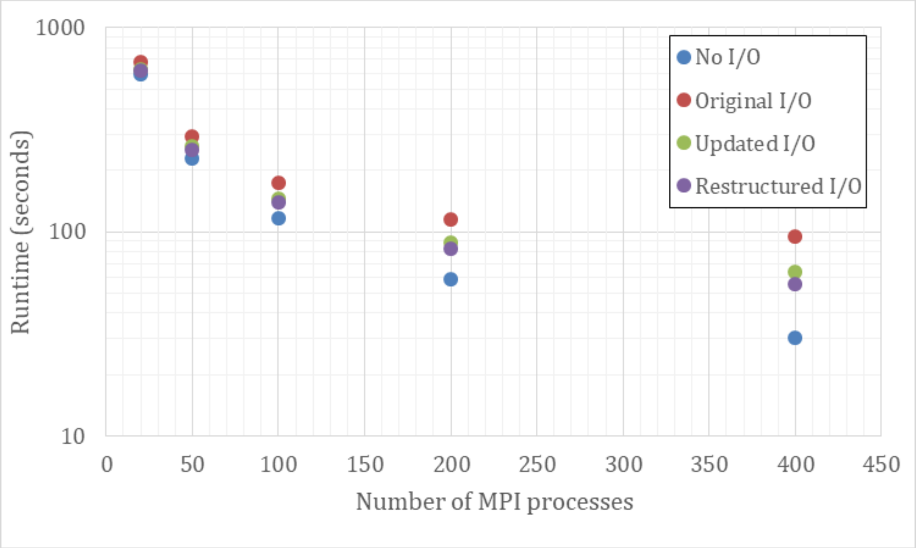

Figure 10: Performance of 800 block simulation with restructured I/O (20 iterations)

4 Load Balancing

By default, COSA distributes the grid across processes as evenly as possible. The grid is divided into blocks, specified in the input file. For N_G blocks and P processes, all processes will have at least $\left\lfloor \frac{N_G}{P} \right\rfloor$ blocks. If P does not exactly divide N_G, the first R processes (going by MPI rank, ascending order) gain one additional block, where R is the remainder of $N_G$modulo P. However, this assumes each block has equal work associated with it, and an unbalanced grid can significantly impact performance (see Figure 4).

To account for variable block sizes, it is possible to define the total work W of a given process to be the sum of the sizes of the blocks it owns (ignoring other factors such as communication). Load balance can now be achieved by distributing blocks across processes so that each process has roughly the same amount of work W, as opposed to a similar number of blocks. It is noted that this is only possible if there are less processes than blocks in the simulation.

To determine which blocks should go to each process, the serial graph partitioning library METIS[18] was used. A weighted graph was constructed from the COSA input block data, where each block is a graph node with a "weight" equal to its size (number of points within the block), and vertices exist between neighbouring blocks.

The graph data is then passed to METIS through a subroutine call, and METIS returns a partitioned block graph so that each partition has approximately the same weight i.e. the same W. Specifically, the approach taken to get a load balanced distribution is:

1. An initial block distribution is done (using the original decomposition algorithm in COSA).
2. Each process then constructs three arrays: an array with the number of neighbours for each block it owns, an array with the process ranks/ID of these neighbours, and an array with the size, i.e. "weight", of each of these neighbours.
3. Each process runs METIS with these input arrays (and other parameters). METIS produces a partition vector specifying the process rank/ID that has been assigned each block.
4. Processes now know which blocks they own and can read them from the input mesh file using existing COSA I/O functionality.

This does not involve communication between processes since each process has all block information for the entire mesh. This does not require processes to read the full mesh file, only the small input file that specifies the size and number of blocks within the grid (the grid metadata).

Using a serial graph partitioner does necessitate all processes replicating this load balancing work, but the trade-off is that it allows us to implement this functionality without communication, meaning it should not impact scaling of COSA to large process counts. Indeed, the time to run this functionality (perform the graph partitioning) was measured on our large test case (16,384 blocks) and METIS completed in under 3 seconds, independent of the number of MPI processes to be used in the simulation.

Another issue that should be highlighted is that load balancing such as that implemented here relies on there being more units of work to distribute to processes that there are processes. If there are 16 simulation blocks and 16 MPI processes then there is no scope for distributing blocks in a load balanced way (if there are different amounts of work within them) as we need at least 2 blocks per process to be able to distribute them (some may still have a single block, with others having multiple).

The load balancing was implemented using the METIS routine `METIS_PartGraphKway`. This routine takes the following data as input:

- `nvtxs`: The number of vertices in the graph. For COSA this is the number of blocks in the simulation
- `ncon`: The number of balancing constraints (weights on each vertex to consider when partition), which we set to 1
- `xadj, adjncy`: The adjacency structure of the graph, specifying which blocks a block is connected to. We use the block and cut (inter-block connectivity) data in the COSA input file to construct these
- `vwgt`: The weights of the graph vertices. This is the number of grid points per block (the size of each block).
- `vsize`: The size of vertices used for communication volume calculations. This is null for COSA.
- `adjwgt`: The weights of the edges between vertices (rather than the vertices themselves). For standard load balancing this is set to 1 for every edge. For load balancing that take communications into account this is set to the size of messages required between the two connected vertices.

- nparts: The number of partitions to split the graph into. For COSA this is the number of MPI processes being used for the simulation.
- tpwgts: Target weight partition. This is set to 1.0/nparts
- ubvec: Load balance tolerance for the partitioning. This is set to 10% (1.1)

METIS_PartGraphKway returns a partition array which has an entry for every block in the simulation, corresponding to the number of the partition that the block has been assign to. These partitions are mapped to MPI processes using the MPI ranks of those processes.

This partition array is then used by the standard COSA data decomposition code to assign blocks to processes. The only modification that was required was to enable assigning non-contiguous block numbers to processes. The original decomposition code assigns blocks to processes in contiguous chunks, but the same is not guaranteed with the METIS functionality.

The new functionality was benchmarked using the NACA0015_Ogrid_UNBALANCED test case and compared to the performance of the original code, with results presented in Figure 11. The benchmarking is undertaken with the original I/O functionality, not the restructured I/O implementation outlined in Section 3. As the load balancing decomposition requires there be more blocks than processes (so varying numbers of blocks can be assigned to each MPI process), the code cannot benefit from load balancing using 256 MPI processes (the same number as the total number of blocks in the simulation).

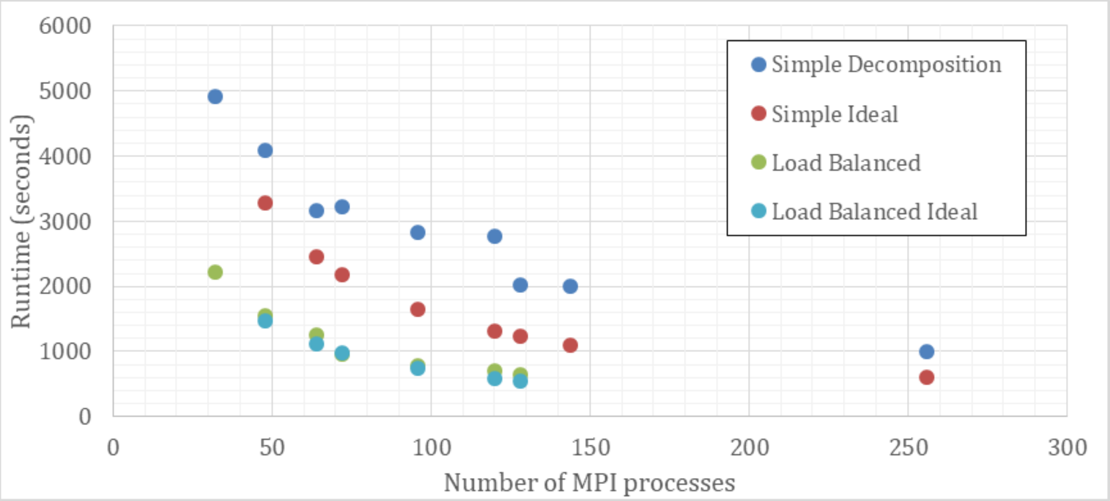

Figure 11: Comparison of the unbalanced simulation runtime vs the same simulation run with the load balanced decomposition

It is evident from Figure 11 that the load balanced code is significantly faster than the original decomposition. Indeed, using 64 cores the simulation can be completed in the same time as the original code using 256 cores and at 128 cores the load balanced code is approximately 3x faster than the original code.

4.1 Compute node usage

The load balancing decomposition also enables the use of numbers of MPI processes that do not evenly divide the number of blocks in the simulation without suffering significant performance impacts.

Whilst this may not seem useful functionality, and is likely to be irrelevant for very large simulations, because there are 24 cores per compute node on ARCHER (and other similar HPC systems have similar number of cores per node) production simulations often encounter the scenario where cores are left empty when running simulations. For instance, if a simulation with 256 blocks is undertaken, with the original decomposition functionality, users would aim to run 16, 32, 64, 128, or 256 MPI processes, as this would give an even number of blocks per MPI process. However, as ARCHER has 24 cores per node, running 16 MPI processes leaves 8 cores idle. Likewise, 32 MPI processes leaves 16 idle on one of the two nodes being used. 64 MPI processes leaves 8 cores idle on one node, and 128 MPI processes leaves 16 idle.

As nodes are allocated exclusively to users on most production HPC systems, leaving cores idle represents a loss of efficiency, with resources that are provisioned but not used. Using the new load balancing functionality, a number of MPI processes that matches the available cores in the nodes being used can be chosen to reduce this inefficiency. For instance, for the NACA0015_Ogrid_UNBALANCED test case, we can use the load balancing functionality to ensure there are no idle cores without impacting runtime, as demonstrated in the following table:

ARCHER nodes used	MPI processes used	Runtime (seconds) using original decomposition	Runtime (seconds) using load balance decomposition
2	32	4924	2220
2	48	4081	1544
3	64	3157	1247
3	72	3231	968
4	96	2833	792
5	120	2764	714
6	128	2016	646

As shown in the table above, if 5, rather than 6, nodes are used, in the load balanced case, a 10 percent performance reduction is incurred with a 20 percent reduction in the cost of our simulation, i.e. a smaller number of nodes is utilised, with all cores used in the nodes, and good performance is still achieved. If the same is tried for the original data decomposition, a significant performance impact is incurred (30 percent slower for a saving of about 15 percent in computing resources) meaning it is not a cost effective approach. The load balancing code enables choosing MPI process counts that match the number of computational cores in the nodes being used without adversely impacting performance, as would have happened with the original decomposition functionality.

Therefore, not only has the load balancing functionality enabled simple grid decompositions to be used directly by COSA without losing performance, it has also enabled more efficient use of the systems (such as ARCHER) that COSA is run on.

5 Conclusions

The I/O optimisation and restructuring undertaken on COSA has reduced the cost of I/O at large core counts by as much as 70 percent, significantly reducing parallel overheads and saving simulation time. Furthermore, the new load balancing functionality has demonstrated up to 3x performance improvement and up to a 4x reduction in resource utilisation when compared to the initial code using an unbalanced, but realistic, simulation.

New load balancing functionality also enables the efficient use of compute nodes, rather than relying on using a number of MPI processes that evenly divides the number of blocks in the simulations. Crucially, it also greatly reduces the time and effort required to generate input grids or meshes for COSA, enabling the output of standard grid generation tools to be used directly in balanced COSA simulations even when the block sizes of the structured multi-block grid differ significantly.

Although load balancing is a relatively well established technology, not all major CFD research codes use it. In light of the fact that several options exist to develop a load balancer, it is believed that the methods and experience reported in this paper on the particular load balancing solution adopted will benefit further development in this area. The development of parallel I/O for large scale applications is a much newer area. Several large CFD research codes still have each MPI process reading their mesh partition from individual grid files, and also writing their solution to separate output files. The additional user time required to pre-process grids and generate individual files for each partition, and to work with a possibly very large number of output files referring to the same solution is a burden that can be avoided using parallel I/O. Even in this case, many diverse options (the high computational efficiency of which is not obvious in all cases) exist, and the parallel I/O methods and demonstrations provided herein provide a valuable contribution to this area.

Acknowledgements

This work was funded by EPSRC under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk), and supported by the Intel and EPCC IPCC centre (https://www.epcc.ed.ac.uk/projects-portfolio/intel-parallel-computing-center).

References

[1] Hall KC, Thomas JP, Clark WS, "Computation of Unsteady Nonlinear Flows in Cascades Using a Harmonic Balance Technique", AIAA Journal 2002, Vol. 40, No. 5, pp. 879-886.

[2] Van der Weide E, Gopinath, A, Jameson, A, “Turbomachinery applications with the time spectral method”, AIAA Paper 2005-4905, *35th AIAA Fluid Dynamics Conference and Exhibit*. June6–9., 2005. Toronto, Canada.

[3] Ekici K, Hall KC, Dowell EH, “Computationally fast harmonic balance methods for unsteady aerodynamic predictions of helicopter rotors”, *J. of Comp. Phys*. 2008, Vol. 227, pp. 6206-6225.

[4] Da Ronch A, McCraken AJ, Badcock KJ, Widhalm M, Campobasso MS, “Linear Frequency Domain and Harmonic Balance Predictions of Dynamic Derivatives”, *J. of Aircraft* 2013, Vol. 50, No. 3, pp. 694-707.

[5] Jackson A, Campobasso MS, “Improving the ease-of-use and portability of the COSA 3D solver for open rotor unsteady aerodynamics”, Technical report eCSE03-002, 2017. Available at http://www.archer.ac.uk/community/eCSE/eCSE03-02/eCSE03-02.php

[6] Jackson A, Campobasso MS, “Shared-memory, Distributed-memory and Mixed-mode Parallelization of a CFD Simulation Code”, *Computer Science Research and Developmen*t 2011, Vol. 26, No. 3-4, pp. 187-195.

[7] Campobasso MS, Piskopakis A, Drofelnik J, Jackson A, “Turbulent Navier-Stokes Analysis of an Oscillating Wing in a Power Extraction Regime Using the Shear Stress Transport Turbulence Model”, *Computers and Fluids* 2013, Vol. 88, pp. 136-155.

[8] Drofelnik J, Campobasso MS, “Comparative Turbulent Three-Dimensional Navier-Stokes Hydrodynamic Analysis and Performance Assessment of Oscillating Wings for Renewable Energy Applications”, *International Journal of Marine Energy* 2016, Vol. 16, 2016, pp. 100-115.

[9] Campobasso MS, Baba-Ahmadi MH, “Ad-hoc Boundary Conditions for CFD Analyses of Turbomachinery Problems with Strong Flow Gradients at Farfield Boundaries”, ASME Journal of Turbomachinery 2011, Vol. 133, no. 4.

[10] Campobasso MS, Drofelnik J, Gigante F, “Comparative Assessment of the Harmonic Balance Navier-Stokes Technology for Horizontal and Vertical Axis Wind Turbine Aerodynamics”, *Computers and Fluids* 2016, Vol. 136, pp. 354-370.

[11] Drofelnik J, Da Ronch A, Campobasso MS, “Harmonic balance Navier-Stokes aerodynamic analysis of horizontal axis wind turbines in yawed wind, *Wind Energy* 2018, DOI: 10.1002/we.2175. In press.

[12] MPI Forum, “MPI: A Message-Passing Interface Standard”. Version 2.2, September 2009. Available at: http://www.mpi-forum.org.

[13] Dagum L, Menon R, “OpenMP: an industry standard API for shared-memory programming", *IEEE Computational Science and Engineering* 1998, Vol. 5, No. 1, 46-55.

[14] Corbett P et al, “Overview of the MPI-IO Parallel I/O Interface”. In: Jain R., Werth J., Browne J.C. (eds) *Input/Output in Parallel and Distributed Computer Systems*. The Kluwer International Series in Engineering and Computer Science, vol 362. Springer, Boston, MA, 1996.

[15] ARCHER Service: http://www.archer.ac.uk

[16] Wang E et al, “Intel Math Kernel Library”. In: *High-Performance Computing on the Intel® Xeon Phi*™. Springer, Cham, 2014.

[17] Braam, PJ, “The Lustre storage architecture”, 2004.

[18] Karypis G, Vipin Kumar V, “METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0”, Technical report 1995, University of Minnesota, USA.