Reducing the runtime and improving the ease-of-use and portability of the COSA 3D harmonic balance Navier-Stokes solver for open rotor unsteady aerodynamics

Document Title: Technical Report

Authorship: Adrian Jackson

Date: 20th January 2017
Version: 1.0

Table of Contents

1 Introduction

This project aimed at substantially improving:

1. the **computational performance** (reducing runtimes while improving parallel scalability from small to very high core counts)
2. **ease-of-use** and **portability** (adopting portable I/O standards)

of the COSA harmonic balance (HB) Navier-Stokes (NS) computational fluid dynamics solver, the key tool of the COSA finite volume compressible NS Fortran code.

COSA is used for accurate unsteady aerodynamic analysis of fluid flows and fluid/structure interaction problems (e.g. flow-induced structural vibrations) in renewable energy, mechanical and aeronautical engineering.

COSA is being developed for a wide class of low-, high- and multi-speed flows, with strong emphasis on open rotor unsteady aerodynamics. The 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 transient effects. In aerodynamic performance, structural integrity 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.

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

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.

In this project, the speed of the HB solver has been substantially increased by developing and parallelising *multi-frequency periodicity boundary conditions* into COSA. The resulting additional speed-up with respect to the existing implementation equals the number of rotor blades (3 to 20, depending on the application).

To improve the code ease-of-use without damaging parallel scalability, a *dynamic parallel load-balancing* capability has been developed, ensuring the number of grid blocks (geometric partitions) assigned to each MPI process takes into account block size, enabling the MPI processes to use different numbers of blocks to ensure a comparable amount of work for all processes. This simplifies and accelerates the grid generation phase, presently constrained by the requirement for all blocks having equal size, as the original code allocates equal numbers of blocks to MPI processes (or as equal as possible depending on the number of blocks in the simulation and the number of processes used).

The scalability of the parallel I/O in the original code is limited, and the I/O wall-clock time is a significant portion of the overall runtime for the large core counts required for complex 3D simulations. To achieve a scalability of the I/O operations comparable to that of the computing part, the code I/O has been restructured and re-parallelized making more efficient use of MPI I/O routines.

Furthermore, we have investigated increased MPI communication performance by overlapping communications with computations, and we have optimised the serial performance by addressing vectorisation issues.

The ease-of-use and portability of the code has be addressed by creating tools to convert the existing I/O format used in the code to other CFD I/O standards, such as CGNS and TecPlot I/O. We investigated writing these formats directly from the simulation code, but the performance of these libraries proved not to be adequate, so this functionality has been retained as external tools for the time being.

The rest of this document will describe COSA in more detail and characterise its existing performance (Sections 2 and 3). We then go on to discuss the technical work undertaken in more detail (Sections 4 to 7) and finally summarise the overall functionality and improvement that has been achieved in the project in Section 8.

2 Simulation Functionality

COSA supports steady, time-domain (TD), and frequency-domain (harmonic balance or HB) solvers, implementing the numerical solution of the Navier-Stokes (NS) equations using a finite volume space-discretisation and multigrid (MG) integration. It is implemented in Fortran and has been parallelised using MPI, with each MPI process working on a set of grid blocks (geometric partitions) of the simulation.

In the HB solver there exists an additional dimension with respect to the steady and TD solvers, which can be viewed as a harmonic varying from 1 to `Nh`, a user specified number of elemental flow harmonics. However, the code solve directly such elemental harmonics, but rather `Nh` equally time-spaced snapshots of the required periodic flow field, linked to the `Nh` elemental harmonics using a Fourier transform.

The code is structured so that the core computational kernels can, for the most part, be reused for the steady solvers and HB simulations, with HB simply requiring an outer loop over the Nh snapshots using the steady solver kernels. The other main difference between the HB and the steady solver is that all large arrays (e.g. solution and residual at all grid cells) of the HB solver have an additional dimension over harmonics.

3 Initial performance

Prior to any optimisation work it's important to understand the current performance of the code. In this section we aim to capture the performance and scaling of COSA on a representative simulation, and evaluate where performance problems exist. The simulation parameters (COSA input files) used to collect the performance results presented in this report is included in Appendix A at the end of the report.

We have performed simulations using 2 different simulations, both harmonic balance, one with 800 blocks and 3,689,952 grid cell (NREL5MW_GRID32_HB_SECTOR), the other with 16,384 blocks and 47,071,232 grid cells (NACA0015_HB_plu_mb16384).

We use a further simulation (NACA0015_Ogrid_UNBALANCED) to test the load balancing functionality and performance issues investigated during this project. This has 256 blocks and 951,808 grid cells, and there are two different versions, 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 mesh generation package. In the unbalanced grid (where block sizes are taken directly from the mesh generator) there is around a 7x difference in number of cells between large and small blocks.

All benchmark data in this document was collected using a version of COSA compiled with the Intel compiler (version 15.0.2.164) on ARCHER using `mkl` (version 11.2.2). Benchmarks are run using a Lustre stripe count of -1 for the directories each simulation is run from.

For benchmarking data we have run each benchmark 3 times and present the fastest run in the graphs. For all the collected data there was a less than 5% difference between the fastest and slowest runs, so data variability (error bars) are omitted from the graphs.

3.1 Process Counts

Given the MPI decomposition functionality in COSA is designed to distribute blocks across processes as evenly as possible the following is a list of sensible process counts for both sample simulations

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

The 256 and 800 block simulation will run on a single node of ARCHER, the 16384 block simulation is too large to fit into memory on a single node so we have started our benchmarking from 1024 cores (43 nodes).

3.2 Basic performance

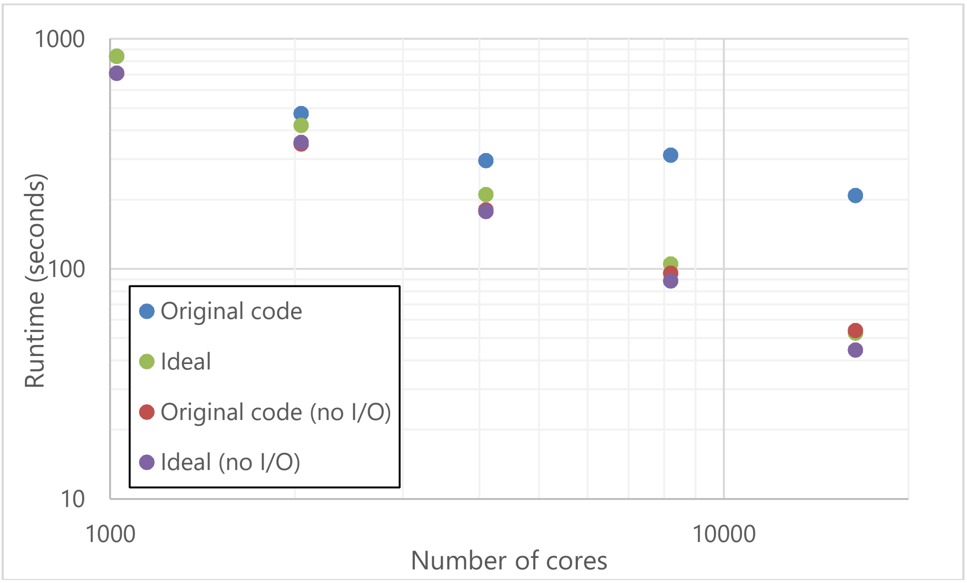

Figure 1: Runtime for 100 iterations of 16,384 block simulation, with and without output I/O enabled.

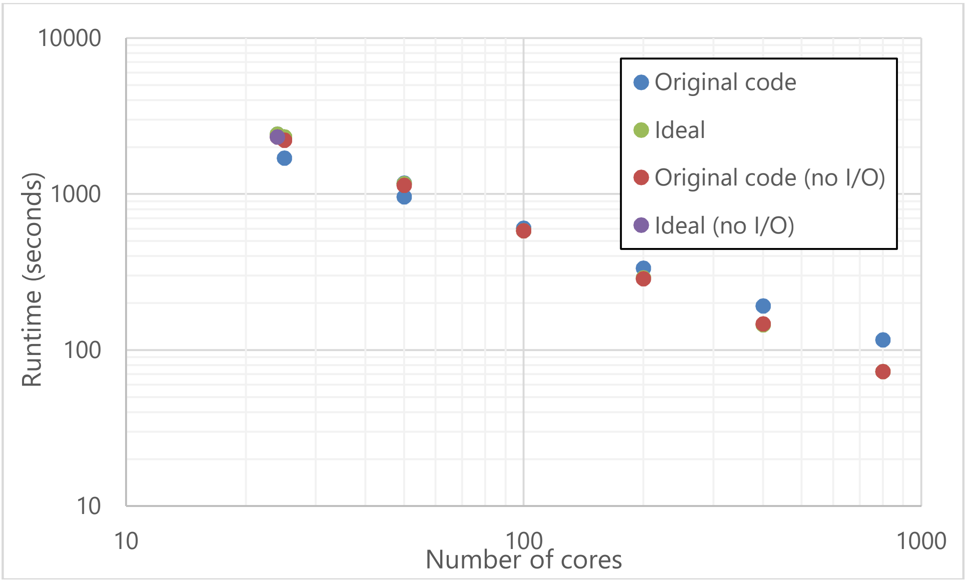

Figure 2: Runtime for 100 iterations of 800 block simulation, with and without output I/O enabled.

It is evident from Figure 1 and Figure 2 that COSA scales very well, even up to maximum block counts, when output I/O is disabled (writing check-pointing and data files). 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% of the runtime for the highest core counts.

It should be noted that we have 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 simulation is run (where thousands or tens of thousands of iterations are used). However, it does illustrate the issue with I/O in these simulations.

3.3 Load balance performance

Part of this project is to investigate and optimise the load balance of the domain decomposition approach used. Therefore, we have also benchmarked the code using a simulation (NACA0015_Ogrid_UNBALANCED) where load balance is an issue.

The simulation is run using the standard decomposition strategy in COSA, which aims to give an equal (or as close to equal as possible) number of blocks to each MPI process. The blocks are pre-defined in the input file.

Figure 3 outlines the parallel performance of this simulation alongside the ideal curve (which is calculated 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 unbalance decomposition.

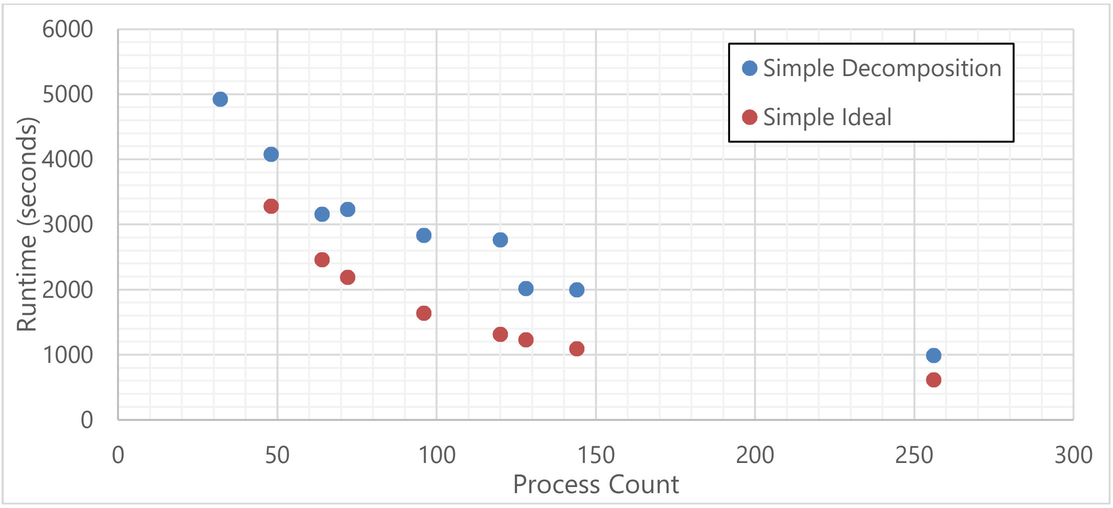

Figure 3: Runtime for 1000 iterations of 256 unbalanced block simulation, with output I/O enabled.

3.3.1 Profiling data

Investigations of the detailed performance of COSA were undertaken with CrayPat to identify the subroutines consuming the most computational time for a given run, the amount of MPI communications performed, and the time spent in I/O for the same run.

Profiling result for the small test case:

100 processes

```
  Samp% |      Samp |      Imb. |  Imb. |Group
        |           |      Samp | Samp% | Function
        |           |           |       |  PE=HIDE

 100.0% | 61,775.2 |        -- |     -- |Total
|-----------------------------------------------------
|   78.1% | 48,217.5 |        -- |     -- |USER
||----------------------------------------------------
```

```
||  19.0% | 11,747.3 |  3,485.7 | 23.1% |vflux_
||   8.8% |  5,418.5 |    857.5 | 13.8% |roflux_
||   6.1% |  3,764.0 |  1,434.0 | 27.9% |muscl_
||   4.7% |  2,909.9 |  1,479.1 | 34.0% |q_face_
||   4.1% |  2,555.1 |    364.9 | 12.6% |tridi_
||   3.9% |  2,380.3 |  1,327.7 | 36.2% |bresid_
||   3.7% |  2,302.4 |  1,445.6 | 39.0% |muscl_bi_
||   3.2% |  1,972.9 |  1,146.1 | 37.1% |rtst_
||====================================================
|  19.0% | 11,742.3 |       -- |    -- |MPI
||----------------------------------------------------
||  11.1% |  6,831.8 | 33,844.2 | 84.0% |mpi_waitany
||   5.3% |  3,262.0 |    910.0 | 22.0% |MPI_FILE_WRITE
|=====================================================
```

800 processes

```
  Samp% |     Samp |    Imb. |  Imb. |Group
        |          |    Samp | Samp% | Function
        |          |         |       |  PE=HIDE

 100.0% | 10,461.0 |      -- |    -- |Total
|---------------------------------------------------
|  56.3% |  5,889.8 |      -- |    -- |USER
||--------------------------------------------------
||  13.1% |  1,375.1 |   464.9 | 25.3% |vflux_
||   6.5% |    683.7 |   132.3 | 16.2% |roflux_
||   4.5% |    469.1 |   199.9 | 29.9% |muscl_
||   3.4% |    355.9 |   192.1 | 35.1% |q_face_
||   3.1% |    320.7 |    82.3 | 20.4% |tridi_
||   2.8% |    293.7 |   193.3 | 39.7% |bresid_
||==================================================
|  41.4% |  4,333.4 |      -- |    -- |MPI
||--------------------------------------------------
||  14.2% |  1,482.2 | 1,111.8 | 42.9% |MPI_FILE_WRITE
||  10.5% |  1,093.8 | 4,369.2 | 80.1% |mpi_waitany
||   8.4% |    877.7 |   885.3 | 50.3% |mpi_file_open
||   7.0% |    730.0 |   802.0 | 52.4% |MPI_BARRIER
|===================================================
```

4 WP1 Parallel multi-frequency periodicity boundary conditions (MFPBCs)

Milestone*: the HB runtime analysis of rotor flows will be reduced by a factor Nb. This will be verified by comparing the runtime of the HB analysis of the provided test case using the complete rotor grid without MFPBCs and that using a single grid sector with MFPBCs.*

Theoretically, multi-frequency periodic boundary conditions (MFPBCs) reduce runtimes of the HB analysis of rotor flows by a factor equal to the number of rotor blades `Nb`. For some particular applications when the flow field is periodic, it is sufficient to simulate the flow only within one repeating region of the whole computational domain.

The interaction of repeating region with the remaining physical domain is provided through the periodic boundary conditions. We focus on rotational periodicity, where one periodic boundary is transformed into the other periodic boundary by the coordinate rotation. Figure 4 represents one repeating region or sector, which depicts two rotationally periodic boundaries, (boundary 1 and boundary 2), where θ is the rotation angle between these two periodic boundaries, and $\underline{\Omega}$ is constant angular velocity given.

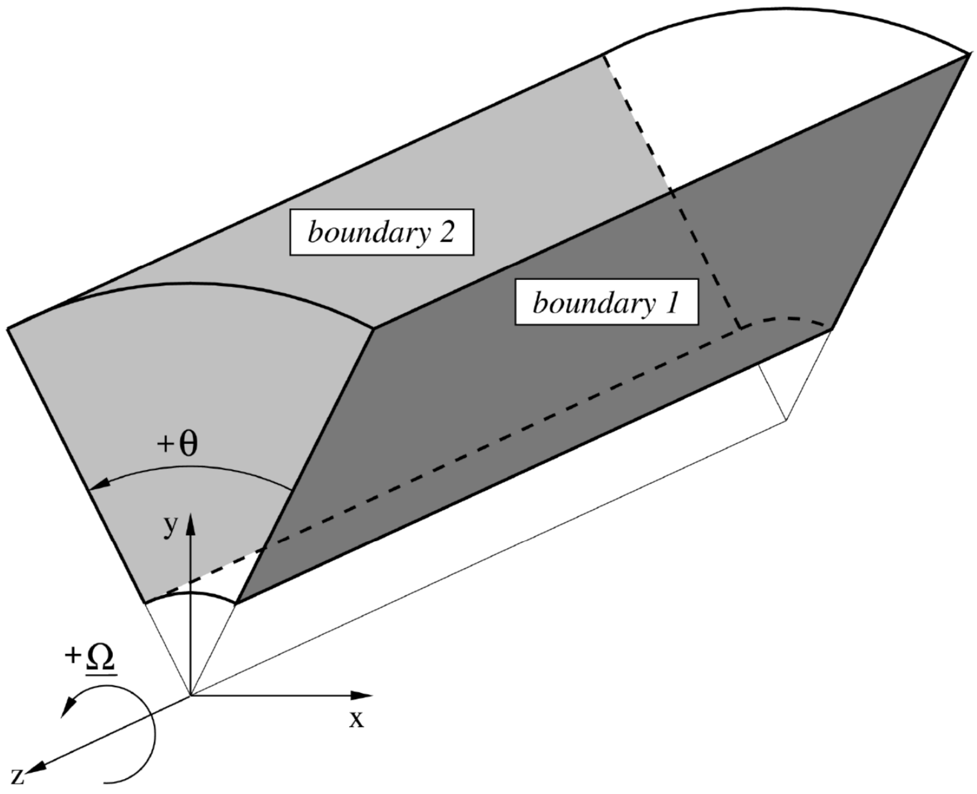

Figure 4: Rotational periodic boundaries

For this work, we have assumed that the rotational axis coincides with the z-axis, and all grid nodes are matching. The data exchange between two periodic blocks is done across the patches on the block surfaces, and it follows exactly the same principle that involves cut boundary condition.

Steady periodicity boundary conditions depend only on the rotation of the coordinate system, means that all scalar quantities (density, pressure, turbulent kinetic energy, and specific dissipation rate) are invariant with respect to the coordinate rotation.

When such variables are copied from the interior domain of first block to halo cells (parts of the data arrays used to store boundary data) in the second block and vice versa, they remain unchanged.

Moreover, all vector quantities (velocity or gradients of scalars) need to be transformed when data exchange between two periodic blocks takes place. The rotation matrix of transformation is the following:

$$\tilde{R} = \begin{vmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix}.$$

Periodic flows in turbomachinery applications satisfy a certain spatial periodicity in addition to temporal periodicity, meaning that the flow about one blade is the same as the flow about neighbouring blade with a time shift. These boundary conditions must be applied in the frequency-domain.

These MFPBC have been implemented in COSA, and benchmarked against running the same simulations without such boundary conditions (where the full domain has to be simulated rather than one rotor).

5 WP2 I/O re-parallelisation and standardisation

Milestone*: the standardised parallel I/O functionality of COSA will have a parallel scalability comparable to that of the computing part of the code. This will be assessed by repeating all I/O scalability tests …, and verifying that the mean deviation of the new I/O-only scalability curve from the ideal speed-up curve is of the same order of magnitude as that of the computation-only curve.*

The current I/O functionality in COSA is parallel, at least on output, with a restart file produced periodically (for checkpoint and restart purposed) and data (also known as `flowtec`) files produced at the end of the simulation (along with a handful of other smaller files).

These output files are write on a per-block basis, with data from consecutively ordered blocks being adjacent in the files. As such, each process can write its blocks to different parts of the output file (be it restart or flowtec) in parallel without interfering with each other. This means I/O is undertaken in parallel, but using non-collective MPI I/O functionality.

Furthermore, 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 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 (16384 blocks) we presented benchmark data on in Section 3 the restart file is of the order of 40GB and there are 9 `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 ½ GB each.

5.1 I/O Functionality

An example of the structuring of I/O to replicate the Fortran file written by the serial code is the original code for writing the restart blocks, which loops over each block a process owns and then does the following:

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

This has been translated into parallel I/O functionality using MPI I/O as follows:

```
call setupfile(fid,disp,MPI_INTEGER)
call mpi_file_write(fid, linelength, 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, linelength, 1,
     &        MPI_INTEGER, MPI_STATUS_IGNORE, ierr)
disp = disp + integersize
```

The above code requires `npde*(kmax1+2)*(jmax1+2)` MPI-I/O operation for each harmonic in the simulation, plus two extra operations to write the line lengths before and after the data, with each operation adding an 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 `MPI_FILE_SEEK`)

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`. For the restart file (which is used for continuing simulations) this is acceptable as the halo data is required for continuing simulations. However, for the output data files (the `flowtec` files) this halo data is not required, so flowtec files are written without the halo data.

The original `flowtec` file functionality is of the following form (note, each harmonic is written to a separate `flowtec` file):

```
write(line1,'(''ZONE T="arturo",I='',i4,'', J='',i4,'',
K='',i4,'',F=POINT, DT=(SINGLE SINGLE SINGLE DOUBLE
DOUBLE DOUBLE DOUBLE DOUBLE DOUBLE DOUBLE)'')')
imax1,jmax1,kmax1
write(fid(n),'(a)') line1
do k=0,kmax
  do j=0,jmax
    do i=0,imax
      write (fid(n),10)
(var1(i,j,k,ipde,n),ipde=1,npde),(var2(i,j,k,ipde,n),ipde
=1,npde)
     end do
   end do
 end do
10 format(3e16.8,7e22.14)
```

Which was translated into MPI I/O as follows:

```
write(line1,'(''ZONE T="arturo HB, mode
'',i2,''",I='',i4,''
&,J='',i4,'',F=POINT, DT=(SINGLE SINGLE DOUBLE DOUBLE
DOUBLE DOUBLE
& DOUBLE DOUBLE)'')') n,imax1,jmax1
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),integersize,1,
&    MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),typechar,1,
&    MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),integersize,1,
&    MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),integersize,1,
&    MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),111,1,
&    MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),integersize,1,
&    MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),111*charactersize,1,
&            MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_CHARACTER)
call mpi_file_write(fid(n),line1,111,
&    MPI_CHARACTER,MPI_STATUS_IGNORE,ierr)
disp = disp + charactersize*111
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),111*charactersize,1,
&    MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
     disp = disp + integersize

do k=0,kmax
  do j=0,jmax
    do i=0,imax
      do ipde=1,npde
         tempdata(tempindex) = var1(i,j,k,ipde,n)
         tempindex = tempindex + 1
      end do
      do ipde=1,npde
         tempdata(tempindex) = var2(i,j,k,ipde,n)
         tempindex = tempindex + 1
      end do
    end do
  end do
end do
call setupfile(fid(n),disp,MPI_DOUBLE_PRECISION)
call mpi_file_write(fid(n),tempdata(1),datasize,
&    MPI_DOUBLE_PRECISION,MPI_STATUS_IGNORE,ierr)
disp = disp + datasize*doublesize
```

Note, the large number of `mpi_file_write` and `setupfile` operations prior to `tempdata` being written are simply to write the block header including the size of the block into the file.

It is evident for both the `restart` and `flowtec` files far more MPI I/O operations are happening than is efficient, 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 are used (where all processes are writing the same amount of data to the file at the same time). Ideally, we would implement this 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 possible to write collective I/O operations to do this.

It could be possible to define 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 none matching numbers of blocks, which would add communication overheads to the I/O operations. Therefore, we decided in this project to continue using non-collective I/O functionality but to optimise the way I/O is performed to improve performance.

5.2 I/O Optimisation

The first I/O optimisation we 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. i.e. from this:

```
do n = 0,2*nharms
  call setupfile(fid,disp,MPI_INTEGER)
  call mpi_file_write(fid, linelength, 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, linelength, 1,
&        MPI_INTEGER, MPI_STATUS_IGNORE, ierr)
  disp = disp + integersize
end do
```

to this:

```
do n = 0,2*nharms
  call setupfile(fid,disp)
  call mpi_file_write(fid, linelength, 1,
&        MPI_INTEGER, MPI_STATUS_IGNORE, ierr)
   disp = disp + integersize
   call setupfile(fid,disp)
   call mpi_file_write(fid, q(-1,-1,-1,1,n),
&        linelength/doublesize,
&        MPI_DOUBLE_PRECISION, MPI_STATUS_IGNORE, ierr)
   disp = disp + linelength
   call setupfile(fid,disp)
   call mpi_file_write(fid, linelength, 1,
&        MPI_INTEGER, MPI_STATUS_IGNORE, ierr)
  disp = disp + integersize
end do
```

We also removed some of the data written out in the `flowtec` headers for each block, retaining only what is required to re-construct the data file afterwards in a post-processing step, and reducing the `flowtec` file writing to this functionality:

```
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),n,1,
&           MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),imax1,1,
&           MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),jmax1,1,
&           MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),kmax1,1,
&           MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize

do k=0,kmax
  do j=0,jmax
    do i=0,imax
      do ipde=1,npde
         tempdata(tempindex) = var1(i,j,k,ipde,n)
         tempindex = tempindex + 1
      end do
      do ipde=1,npde
         tempdata(tempindex) = var2(i,j,k,ipde,n)
         tempindex = tempindex + 1
      end do
    end do
  end do
end do
call setupfile(fid(n),disp,MPI_DOUBLE_PRECISION)
call mpi_file_write(fid(n),tempdata(1),datasize,
&    MPI_DOUBLE_PRECISION,MPI_STATUS_IGNORE,ierr)
disp = disp + datasize*doublesize
```

Whilst the `flowtec` files still require almost the same amount of data to be written (the `tempdata` is not reduced in size) we have halved the number of MPI I/O operations required to output this data. Likewise, the same size of restart data is still be written, but we have dramatically reduce the number of operations required to perform this I/O.

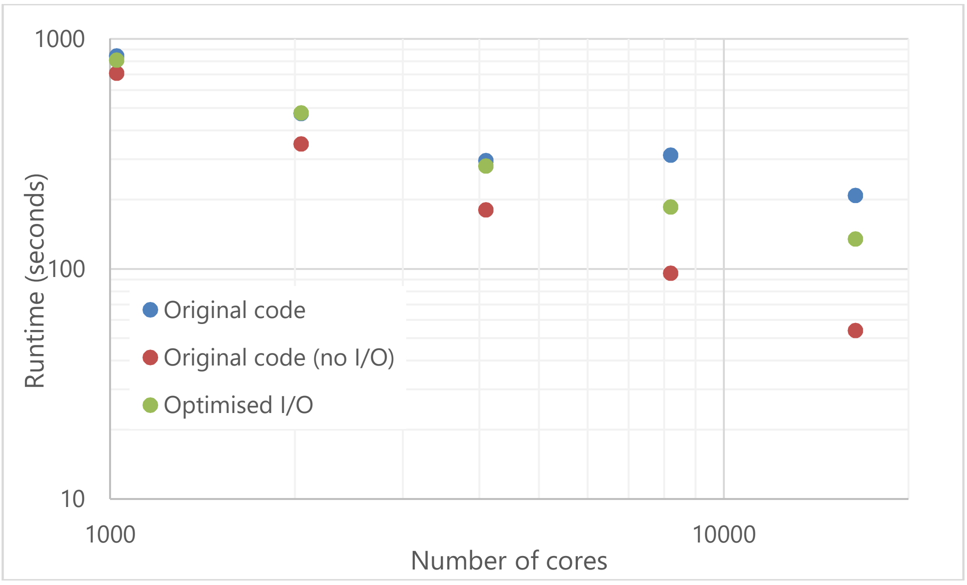

Figure 5: Performance of 16384 block simulation with optimised I/O (100 iterations)

The performance of the optimised I/O is show in Figure 5 for the large simulation. We can see that the impact of this optimisation at lower core counts is not significant, but when we get to large core counts the optimised I/O is having a significant impact. The code is around 70% faster at 8192 cores and around 50% faster at 16,384 cores.

We benchmarked the optimised I/O functionality with the smaller test case as well, as shown in Figure 6. Note for this benchmark we reduce the number of iterations of the simulation from 100 to 20 to highlight the I/O costs in the simulation.

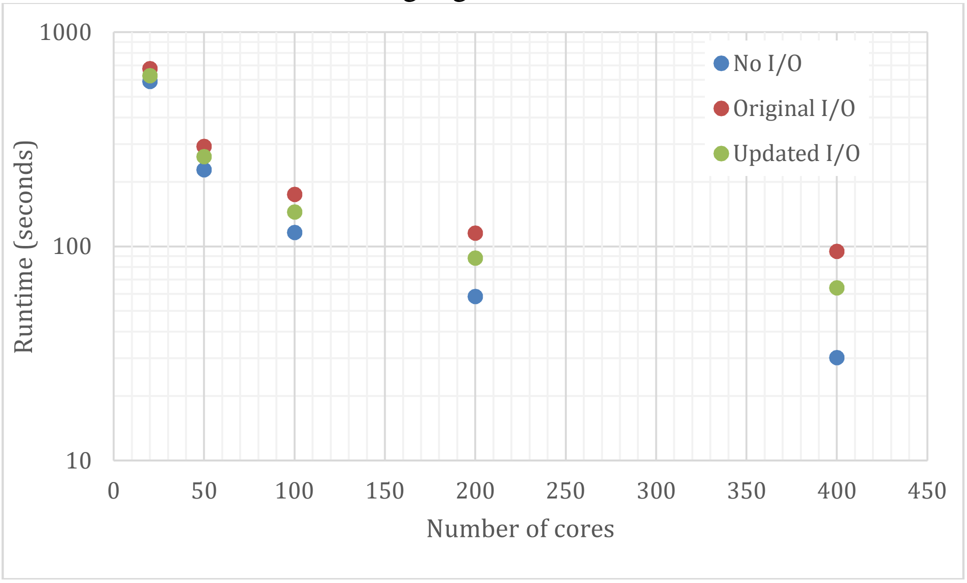

Figure 6: Performance of 800 block simulation with optimised I/O (20 iterations)

As with the larger test case we can see the optimised I/O is clearly faster, around 50% when using 400 MPI processes (400 cores). However, it's clear that the I/O is still expensive, with the code performance when not producing output data still much faster than the optimised test cases.

5.3 Restructured I/O

To further optimise the I/O performance of COSA we looked at restructuring the way the `flowtec` files were written. Firstly we remove the per block header data completely, replacing it with a per file header, written by one process, that includes all the block sizes at the start of the file. This removed the need for this code within each `flowtec` block write:

```
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),n,1,
&            MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),imax1,1,
&            MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),jmax1,1,
&            MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
call setupfile(fid(n),disp,MPI_INTEGER)
call mpi_file_write(fid(n),kmax1,1,
&            MPI_INTEGER,MPI_STATUS_IGNORE,ierr)
disp = disp + integersize
```

Reducing the number of write operations per harmonic per block to a single write for the data. Furthermore, we 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.

We constructed a new output routine 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, one no longer need a temporary array large enough to store all the data for all the blocks a process owns, we simply need a temporary array that can hold the data for a single block. Secondly, we can enable re-use of data that has been collected into the temporary array, optimising cache usage and therefore reducing computational costs.

Finally, we 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. This is 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 is limited in performance by file locking that is undertaken when a given process is reading the file. Whilst this locking (ensuring exclusive access to the file for a single MPI process) is not a large issue with small process counts, when 16 thousand MPI processes are trying to read the same file it can slow down this initial I/O.

Therefore, we parallelised the reading of the mesh file using MPI I/O, removing this file locking and the associated process synchronisation.

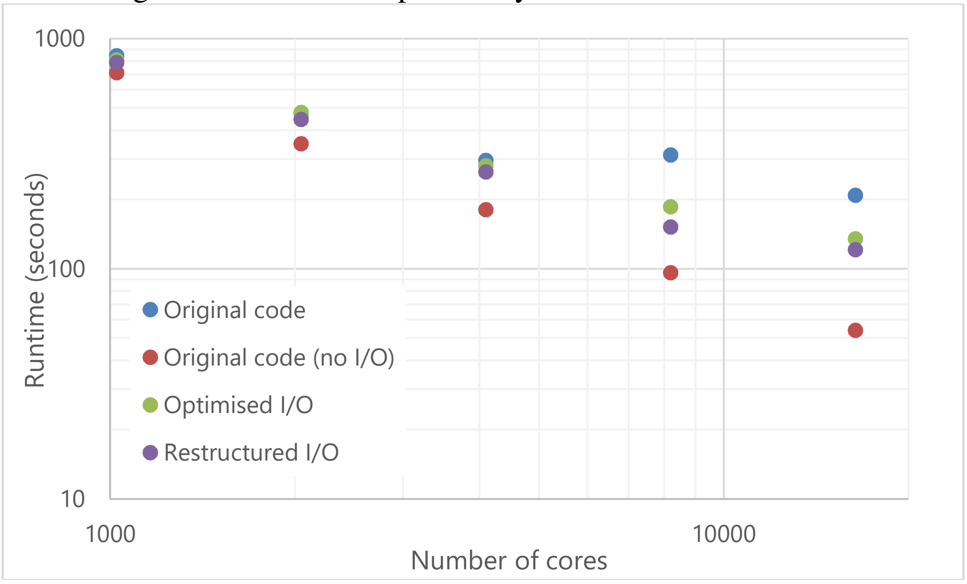

Figure 7: Performance of 16384 block simulation with restructured I/O (100 iterations)

Figure 7 and Figure 8 show the performance of the restructured and optimised I/O functionality. We can see that it has further improved performance for COSA, with the code now around 100% faster on 8192 cores and 70% faster on 16384 cores for the large test cases, and 70% faster on 400 cores for the small test case.

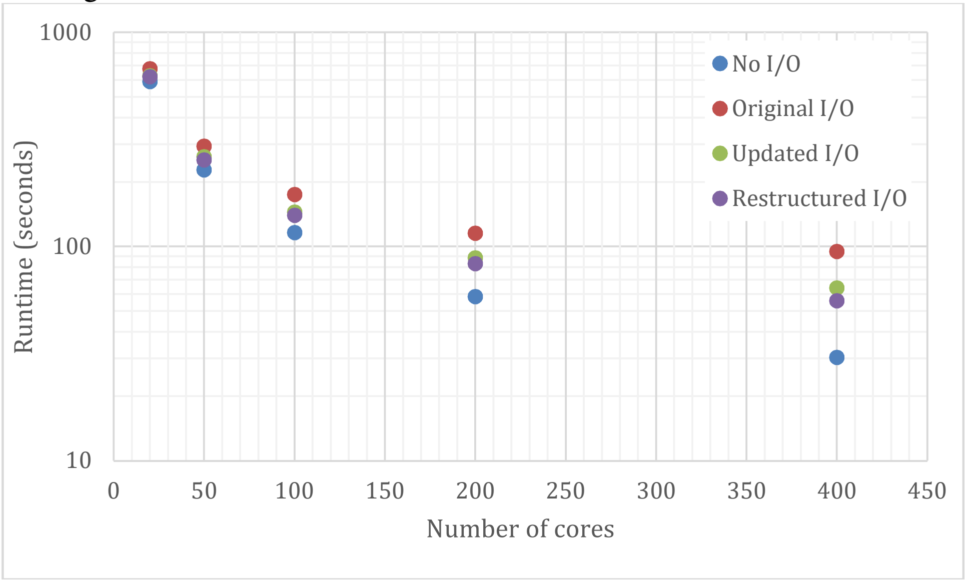

Figure 8: Performance of 800 block simulation with restructured I/O (20 iterations)

5.4 I/O Standardisation

There are a number of standard I/O formats commonly used in CFD applications, with CGNS[1] and TecPlot[2] being two of the most common. Both have parallel I/O functionality, and are widely used by visualisation, grid generation, and meshing packages for file creation and reading.

As such, it would be useful for COSA to be able to produce and consume data in these formats, as it would allow direct visualisation of simulation data without conversion from the COSA I/O format to TecPlot or CGNS, and it would also using meshes produced from generators without having to convert them into the COSA binary mesh format first.

However, for this functionality to be usable in COSA the performance needs to be close to that of MPI I/O, so we investigated both CGNS and Tecplot parallel I/O functionality to see what kind of performance could be achieved.

5.4.1 CGNS

CGNS builds on HDF5 to provide I/O functionality. It stores data in trees, with different types of data about the simulation stored in different branches and leaves of the tree, along with associated metadata.

CGNS allows both serial and parallel I/O functionality, with metadata and data treated separately (both can be written in serial and in parallel). Not all the CGNS functionality has associated parallel versions, but the functionality COSA requires has been parallelised.

We created a CGNS version of the parallel restart file functionality, with the metadata about the restart file written in serial by a single process, and the data written in parallel by all processes.

The metadata I/O was implemented using the following code, which creates a block in the file for each block in the simulation:

```
if(amcontroller) then
   call cg_open_f('rest.cgns',CG_MODE_WRITE,fid,ierr)
   if(ierr .ne. CG_OK) then
      write(*,*) 'cg_open_f restart error'
      call cg_error_print_f()
   end if

   call cg_base_write_f(fid,'gridbase',3,3,basenum,ierr)
   if(ierr .ne. CG_OK) then
      write(*,*) 'cg_base_write_f error'
      call cg_error_print_f()
   end if
```

[1]CFD General Notation System https://cgns.github.io/
[2] http://www.tecplot.com/products/tecplot-360/

```
   do i = 1,blocksize
      blocknum = i
      write(zonename, "(A5,I6)") "block",blocknum
      call cg_zone_write_f(fid,basenum,zonename,sizes,
     &           Structured,zonenum,ierr)
     if(ierr .ne. CG_OK) then
       write(*,*) 'cg_zone_write_f error'
       call cg_error_print_f()
     end if
     call
cg_goto_f(fid,basenum,ierr,'Zone_t',zonenum,'end')
     if(ierr .ne. CG_OK) then
       write(*,*) 'cg_goto_f error'
       call cg_error_print_f()
     end if
     write(linkpath,'(a,i6,a)') 'gridbase/block',zonenum,
     &            '/GridCoordinates'
     call cg_link_write_f('GridCoordinates','mesh.cgns',
     &           linkpath,ierr)
     if(ierr .ne. CG_OK) then
        write(*,*) 'cg_link_write_f error'
        call cg_error_print_f()
     end if
     call cg_user_data_write_f('User Data',ierr)
   end do

   call cg_close_f(fid,ierr)
   if(ierr .ne. CG_OK) then
      write(*,*) 'cg_close_f error'
      call cg_error_print_f()
   end if
end if
```

Then the following code was used to write the data into the restart file, where a block is written in a single call (as in the MPI I/O functionality):

```
basenum = 1
call cgp_open_f('rest.cgns',CG_MODE_MODIFY,fid,ierr)
solnum = 1
do i = 1,blocksize
  zonenum = i
  write(zonename, "(A5,I6)") "block",zonenum
  call cg_zone_read_f(fid,basenum,zonenum,
&         tempzonename,tempsizes,ierr)
   if(trim(tempzonename) .ne. zonename) then
       write(*,*) 'error block name:
',zonename,tempzonename
   end if
   call cg_goto_f(fid,basenum,ierr,'Zone_t',zonenum,
&         'UserDefinedData_t',solnum,'end')
   call cgp_array_write_f(fieldname,RealDouble,5,qsizes,
&          arraynum,ierr)
   if(ierr .ne. CG_OK) then
       write(*,*) 'cg_array_write_f error'
       call cg_error_print_f()
   end if
end do
```

Whilst CGNS is relatively easy to implement, the performance is not comparable to MPI I/O. A test on 512 MPI processes with a 40GB restart file showed the following performance:

MPI I/O File Read	3 seconds
CGNS File Read	233 seconds
CGNS File Write	533 seconds

Therefore, it was decided not to implement I/O directly in COSA for reading and writing CGNS data. Instead, we created some serial applications that could covert CGNS files in COSA format, and vice versa, for pre/post processing of files outside the parallel simulation.

We investigate why the CGNS performance is so much slower than MPI-I/O. It is apparent that CGNS is design for simulations where there are a single, or small number of, block(s) in the simulation, with each MPI process responsible for part of a block. This is different to COSA, where there are lots of blocks in a simulation, with each process responsible for one or more blocks.

The metadata and operational costs of handling blocks in CGNS format seem to account for the slow functionality, and as they are not generally encountered in applications using CGNS (because they will only have a single block) the way the I/O is structured does not lend itself for efficient multi-block I/O.

5.4.2 Tecplot

Tecplot provide a number of commercial tools for visualising, analysing, and generating CFD data. To support this they have an I/O format, TecplotIO, that includes a library for I/O operations.

TecplotIO supports three different file formats; a legacy ASCII format they their tools can read but requires conversion by the tools to process, a binary format known as `plt`, and a partitioned binary format `szplt` designed for large scale parallel I/O.

We implemented the same restart file functionality we implement for CGNS, targeting `szplt` files as these enable parallelisation of I/O:

```
if(tecini142(trim(titlename),trim(varlist),
&         filename,
&         trim(pwd)//char(0),
&         fileformat,filetype,debug,isdouble) .ne. 0) then
  write(*,*) 'error initialising tecini'
end if
do j=1,nblocks
```

```
  imax1 = blockindexes(1,j)
  jmax1 = blockindexes(2,j)
  kmax1 = blockindexes(3,j)
  datasize = imax1*jmax1*kmax1
  write (blocknumname, "(I5)") j
  if(teczne142('block'//trim(blocknumname)//char(0),
&    zonetype, imax1, jmax1, kmax1, imaxmax,
&    jmaxmax, kmaxmax, simtime, strandid, parentzone,
&    isblock, nfconns, fnmode,
&    tnfnodes,ncbfaces,tnbconns,Null, Null, Null,
&    shrconn) .ne. 0) then
     write(*,*) 'error setting up zone'
  end if

  if(tecdat142(datasize*10,tempdata,isdouble) .ne. 0)
then
     write(*,*) 'error writing block data'
  end if

end do

if(tecend142() .ne. 0) then
   write(*,*) 'error calling tecend'
end if
```

Unfortunately, whilst the above functionality does write a Tecplot file, writing the `szplt` file takes around 7 times as long as it did to write the CGNS file. The `plt` performance is not as bad, indeed it is around 4x quicker than the CGNS functionality in serial, but does not have parallel functionality so is not suitable for inclusion in the simulation code.

We have, as with CGNS, created pre-/post-processing utilities to convert the data to and from Tecplot format. The issue with `szplt` I/O performance has been reported to Tecplot and they are working on a fix for this, so it is possible that future versions of Tecplot may be suitable for integrating into COSA.

6 WP3 Improvement of MPI communications

Milestone: *Improved parallel communication performance on COSA, ideally removing the MPI communication costs altogether (10-20% of the runtime of the code). This will be evaluated using profiling and benchmarking as with the other work-packages.*

COSA uses nonblocking MPI communications (i.e. `MPI_ISend` and `MPI_IRecv`) to send and receive halo data between processes. The nonblocking communications are structured in such a way as to enable all communications for a given iteration to be started, the MPI library to progress them, and then the code waits until they have finished.

This was implemented to ensure the ordering of the boundary communication for blocks a process owns is not inhibiting parallel performance (i.e. so that we do not enforce a particular order of halo communications between blocks and processes). However, the same functionality can also be used to overlap communication and computation for COSA.

Currently the initiation of communications (the nonblocking subroutine calls) and the checking the communications have finished (the wait subroutine calls) are implemented within the same routines in COSA, the `cut` routines that handle halo exchanges.

In this work we separated out the initiation of communications, and the checking the communications have finished (with associated data handling) into two separate routines; communication and waiting.

The communication routines collate data to be sent and start the send and receive nonblocking function calls. The waiting routines check that communications have finished and then unpack received data into the appropriate data array. All that needs to be passed between these two routines are the nonblocking handles generated by the nonblocking send and receive calls that need to be checked by the wait routines.

We tested this new functionality to ensure correctness and investigated performance of this overlapping. We tried with and without Cray's nemesis helper threads, designed to asynchronously progress MPI messages for applications, and enabled using these environment variables when running the application:

```
export MPICH_NEMESIS_ASYNC_PROGRESS=1
export MPICH_MAX_THREAD_SAFETY=multiple
export MPICH_GNI_USE_UNASSIGNED_CPUS=enabled
```

Unfortunately, there were no significant performance improvements from splitting the MPI communications. Whilst the new functionality allows for potential overlapping of communication and compute, the way COSA is structured means there is very little work that can be overlapped with the communication. Values to be communicated are calculated, sent, and then used immediately in the simulation, there is no real scope for doing other calculations whilst that data is being sent.

To properly exploit this functionality, significant restructuring of the computational kernels in COSA would be required to intermingle different types of calculation, and thus give potential for undertaking calculations of one type whilst data for a different type of calculation is communicated.

6.1 Dynamic allocation

We also re-organised the memory used in MPI communications. The existing code uses static arrays with compile time defined limits to store data being communicated between processes (the temporary storage required to enable nonblocking MPI communications).

As these limits can only be changed by re-compiling the code, and the code will crash if the arrays are too small, these arrays are generally larger than required which is wasteful of memory, but also risks failed jobs when the limits aren't sufficient.

We have re-organised the code and calculated the exact amount of memory required for the MPI communications. Now we dynamically allocate it for the communication routines, ensuring exactly the right amount of memory is being used.

7 WP4 Dynamic load balancing

***Milestone**: the new DLB capability will yield runtime reductions of about one order of magnitude for simulations using production multi-block grids generated with state-of-the-art grid generators.*

As COSA is a multi-block code, simulation blocks are the standard unit for data decomposition across processes. We do not split blocks within COSA as this would require significant functionality to ensure that geometry and boundary conditions are correctly handled, we simply distribute existing blocks to processes in as even a way as possible, trying to ensure each process has the same number of blocks.

However, this places significant burdens on the process of creating simulation datasets. A given simulation grid must be manually split into blocks prior to running, and this splitting must ensure that blocks have equal, or close to equal, amounts of work to ensure the parallel simulation is load balanced.

7.1 Load Balancing Optimisation

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 $\left\lfloor \frac{N_G}{P} \right\rfloor$ number of blocks. If P does not exactly divide N_G, the first R processes (going by rank ascending order) gain one additional block, where R is the remainder of $N_G$modulo P.

To account for variable block sizes, we 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 work, W, as opposed to a similar number of blocks.

To determine which blocks should go to each process, we use the serial graph partitioning library METIS[3]. Whilst we could have implemented our own partitioning algorithm, there is nothing within the functionality we required that necessitated developing this functionality given packages such as METIS are already well established and tested. METIS can quickly and efficiently find the optimal partition of a graph, using constraints specified by the user.

[3] http://glaros.dtc.umn.edu/gkhome/metis/ Karypsis Lab, University of Minnesota

We construct a weighted graph from the COSA input block data, where each block is a graph node with "weight" equal to its size (number of grid cells 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, 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

It is worth noting, that, this does not involve communication between processes since each process has all block information for the entire mesh. This does not mean that each process needs to read in all the input data, simply all processes need to read the input file that specifies the size and number of blocks within the grid.

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, we measured the time to run this functionality on our large test case (16,384 blocks) and METIS completed in under 3 seconds.

Another issue that should be highlighted is that load balancing such as that we are implementing here relies on there being more units of work to distribute to processes that there are processes. If we have 16 simulation blocks and 16 MPI processes then we have no scope for distributing blocks in a load balanced way (if they different in the amount 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).

We implement the load balancing 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 (boundary) 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. We set this to be `1.0/nparts`
- `ubvec`: Load balance tolerance for the partitioning. We set this to be 10% (1.01)

It return a partition array which has an entry for every block in the simulation with the number of the partition that has been assign to. These partitions are mapped to MPI processes (the partition number is equivalent to the MPI process rank + 1 of the process that will own that block).

This partition array is then used by the standard COSA data decomposition code to assign blocks to processes. The only modification that we need to make was to enable assigning none 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. All that is required to enable this is to keep an array for blocks assigned to this process with the block number of each block we own.

Finally, we also implemented functionality to read a pre-defined block decomposition from file, to enable the load balancing to be done outside the simulation and simply provided as input.

7.1.1 Communication costs

The functionality discussed previously will attempt to distribute grid cells evenly across processes, i.e. assign blocks to processes so they have roughly equal numbers of grid cells to compute. This will balance computational work across MPI processes, however it does not take into account the communication cost associated with blocks and their halo exchanges. As a consequence, it could generate domain decompositions that balance the amount of work between MPI processes but significantly increases the amount of communication required for the simulation.

This is because if neighbour blocks are owned by the same MPI process then associated communications can be undertaken through simple memory copies. However, if they are owned by different MPI processes then communications through the MPI library will be required. Even if these are undertaken through memory copies inside the MPI libraries (i.e. the MPI processes are on the same node) this will still involve copying the data to the temporary array for sending via MPI and then unpacking it after the receive has happened.

We can extend our domain decomposition functionality to take communications into account. Using the same METIS functionality as before, we can adjust the adjacency weights of each edge to take into account the communications required between blocks. This type of load balancing can be enabled at run time by specifying a flag in the input file.

7.1.2 Load balancing performance

We benchmarked the new functionality and compare the performance to the original code using the unbalanced grid we used in the initial benchmarks. Because the load balancing decomposition requires there to be more blocks than processes (so varying numbers of processes can be assigned to each MPI process) we cannot benefit from load balancing using 256 MPI processes (the same number as the total number of blocks in the simulation). Performance results are shown in Figure 9.

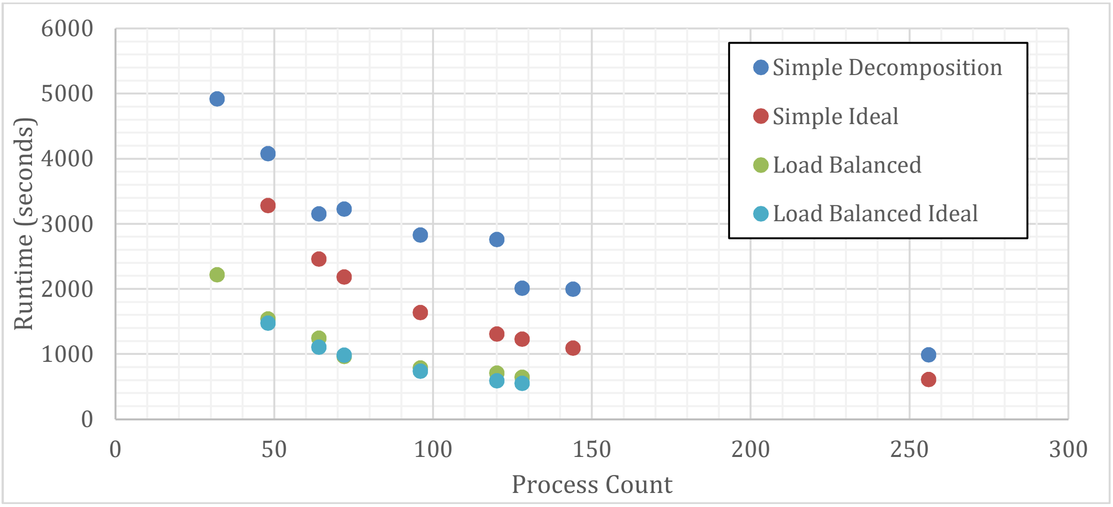

Figure 9: Comparison of the unbalanced simulation runtime vs the same simulation run with the load balanced decomposition

It is evident from the above graph that the load balanced code is significantly faster than the original decomposition. Indeed, using 64 cores we can complete the simulation in the same time as the original code using 256 cores and at 128 cores we are 50% fast with the load balanced code that 256 cores using the original code. At 128 cores the load balanced code is ~3x faster than the original code.

7.1.3 Compute node usage

Furthermore, the load balancing decomposition also enables us to use 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 a none power of two number of cores in the ARCHER compute nodes (24 cores per node) we often have the scenario that cores are left empty when running simulations.

For instance, if I am running a simulation with 256 blocks, with the original decomposition functionality I would want to run 16, 32, 64, 128, 256 MPI processes, because this would give me an even number of blocks per MPI process. However, as ARCHER has 24 cores per node, running 16 MPI processes leaves 8 cores spare. Likewise, 32 MPI processes leaves 16 cores spare on one of the two nodes being used. 64 MPI processes leaves 8 cores spare on one node, and 128 MPI processes leaves 16 spare.

The load balancing functionality lets us fill such spare cores without having to impact runtime, as demonstrated in the following table:

ARCHER Nodes Used	MPI Processes Used	Original decomposition	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

We can see from the table above that if we use 6 nodes instead of 5 in the load balanced case we get around a 10% performance improvement with a 20% increase in the cost of our simulation. We can drop back to the smaller number of nodes, fill them up completely, and still get good performance. If we do the same for the original data decomposition we get a significant performance impact, thus making it not cost effective.

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.

7.1.4 Communication cost decomposition performance

We also benchmarked the functionality that takes into account communication costs as well as the number of grid cells within a block when the decomposition is constructed. Using the same benchmark we compared performance to the standard load balancing functionality, with the results presented in Figure 10.

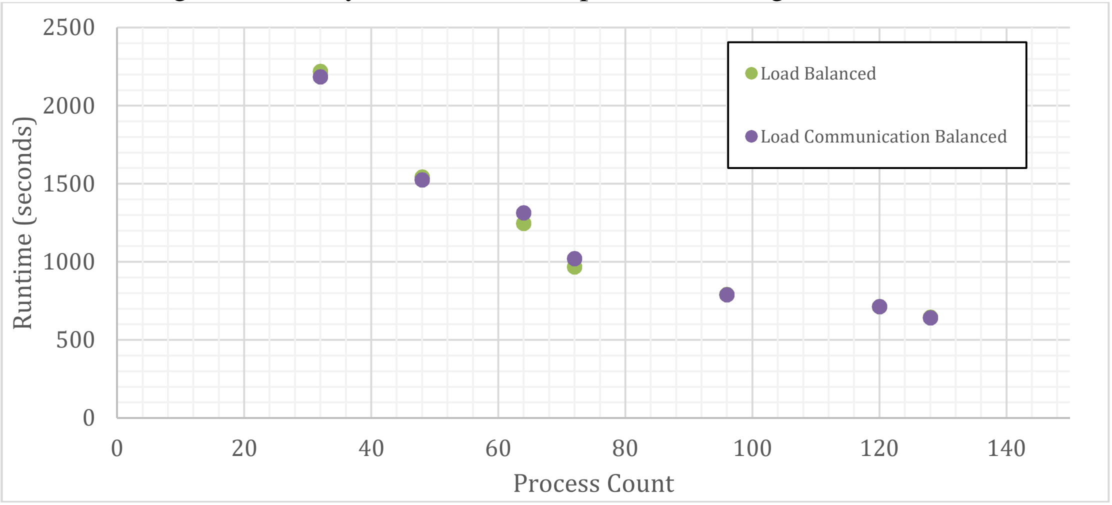

Figure 10: Comparison of communication load balancing vs workload balancing

As can be seen in the above graph the communication balancing functionality does not significantly change the performance of the application for the simulations we have undertaken. However, for larger simulations it may be more important.

8 WP5 Serial Improvements

***Milestone**: the core computational kernels will vectorise using the Intel and Cray compilers on ARCHER, which is expected to greatly reduce runtimes of all COSA simulation types.*

We have investigate optimising the top computational kernels in COSA, namely those listed in the profiling data we presented in Section 3.3.1:

- `vflux`
- `rhoflux`
- `muscl`
- `q_face`
- `tridi`
- `bresid`
- `muscl_bi`
- `rtst`

In these routines we looked to improve vectorisation and reduce computational costs. This work involved restructuring some loop calculations to reduce temporary arrays and boost cache re-use, removing loop invariants, and replacing divisions by reciprocals such as:

```
do ipde = 1,4
     fac1 = fact * vol(i,j)/dt
end do
```

becomes:

```
recip = 1.0d / dt
do ipde = 1,4
      fact1 = fact * vol(i,j) * recip
end do
```

Whilst the above restructuring does have an impact on the calculated results, we validated that simulation results with the new functionality were close enough to the original code to be acceptable.

After implementing the optimisations we re-ran out benchmark cases and observed between a 3-5% performance improvement for this work.

9 Summary

We have made a number of improvements to COSA to improve the performance and usability of the code. The new boundary conditions (MFPBC) enable harmonic balance simulations to fully realise the performance savings this frequency domain approach allows for turbo machinery applications.

Our I/O work has reduced the cost of I/O at large core counts by as much as 70%, significantly reducing parallel overheads and saving simulation time.

We have provided tools to convert COSA data to and from standard CFD data formats (CGNS and Tecplot) enabling data generated from other packages to be read by COSA and data produced by COSA to be read by other packages.

We have implemented load balancing functionality that has demonstrated up to 3x performance improvements and up to a 4x reduction in resource utilisation when compared to the existing code using an unbalanced simulation. It also enables the efficient use of full compute nodes, rather than relying on using a number of MPI processes that evenly divides the number of blocks in the simulations.

Crucially, though, it also greatly reduces the time and effort required to generate input grids or meshes for COSA, enabling the output of standard grid/mesh generation tools to be used directly in COSA.

Finally, we have made the code easier to use through dynamic memory allocation, and enabled future communication optimisation by splitting sending/receiving and waiting for message completion. Altogether these performance improvements and functionality upgrades significant increase the potential for COSA to be a highly useful, usable, and scalable simulation package.

10 Future Work

There are a number of areas where further optimisation or functionality is could be added to COSA. If the `szplt` I/O format performance issues are fixed by Tecplot then this I/O format could be evaluated for performance and added to COSA if it performs sufficiently well.

A more extensive refactoring of the code would enable exploiting the splitting of communication we have implemented in this project.

There is also potential to do some hardware specific optimisation in COSA, porting to KNL and optimising where the memory is allocated on the hardware would potential bring benefits, as would porting to GPUs.

11 Appendix A

COSA input files used for this project

11.1 16384 block input file header:

```
Input file for 3D Euler/NS code
debug      model     flow-type  id
n          sst       external   aircraft
gamma      reyno     pranl      machfs     alpha    beta
1.4d0      1.0d+7    0.71d0     0.1        5.00     0.d0
```

```
prant       tkefar     mutfar      wall        roughk
0.9d0       1.d-6      0.1         wilcox      0.d0
posprd      lim_ptke   prdlim      lim_pome
n           n          100         n           minimum
second
flow-mode   solver     rk option   nharms
unsteady    hb         rkex        4
move        freq.      dh0x        dh0y
plunge      0.01       1.0         0.d0
irest       srest      cfl         cdff        lmax
iupdt       toler
 0          10000      1.5         4           100         1
1.d-14
rkap        irs-typ    cfli        psi
-1.         cirs_v1    3.0d0       0.25
cflt        cflit      ramp-opt    n(3)        n(2)
n(1)
2.00        4.00       ramping1    1000        500         250
cfli(2)     cflit(2)   cfli(1)     cflit(1)    stop
1.5         2.0        0.5         0.5         50000
lim         epslim     cntrpy      etpfxtyp    entfxctf
4           1.d-6      0.d0        0           0.95d0
nlevel      nl_crs     nl_fmg      nstart      npre
npost       ncrs
3           4          1           3           3           3
6
prol.type              restr.type  lim.type
bilinear               ho_rest     lim_corr3
flow-speed
nolomach
tref
288.2
functional
default     0.25       0.0 0.5
lref1       lref2      lref3
1.0         1.0        1.0
1
224  4609 4610 4611 4612 4613 4614 4615 4616 4865 4866
4867 4868 4869 4870 4871 4872 5121 5122 5123 5124 5125
5126 5127 5128 5377 5378 5379 5380 5381 5382 5383 5384
5633 5634 5635 5636 5637 5638 5639 5640 5889 5890 5891
5892
 5893 5894 5895 5896 6145 6146 6147 6148 6149 6150 6151
6152 6401 6402 6403 6404 6405 6406 6407 6408 6657 6658
6659 6660 6661 6662 6663 6664 6913 6914 6915 6916 6917
6918 6919 6920 7169 7170 7171 7172 7173 7174 7175 7176
742
5 7426 7427 7428 7429 7430 7431 7432 7681 7682 7683 7684
7685 7686 7687 7688 7937 7938 7939 7940 7941 7942 7943
7944 8193 8194 8195 8196 8197 8198 8199 8200 8449 8450
8451 8452 8453 8454 8455 8456 8705 8706 8707 8708 8709 87
```

```
10 8711 8712 8961 8962 8963 8964 8965 8966 8967 8968 9217
9218 9219 9220 9221 9222 9223 9224 9473 9474 9475 9476
9477 9478 9479 9480 9729 9730 9731 9732 9733 9734 9735
9736 9985 9986 9987 9988 9989 9990 9991 9992 10241 10242
 10243 10244 10245 10246 10247 10248 10497 10498 10499
10500 10501 10502 10503 10504 10753 10754 10755 10756
10757 10758 10759 10760 11009 11010 11011 11012 11013
11014 11015 11016 11265 11266 11267 11268 11269 11270
11271 1
1272 11521 11522 11523 11524 11525 11526 11527 11528
16384
```

11.2800 block input file header

```
Input file for 3D Euler/NS code
debug      model      flow-type  id         nblade
n          sst        external   shawt      3
gamma      reyno      pranl      machfs     alpha    beta
1.4d0      7.7d+5     0.71d0     0.0335     0.00     20.00
prant      tkefar     mutfar     wall       roughk
0.9d0      1.d-4      0.1        menter     0.d0
posprd     lim_ptke   prdlim     lim_pome   pr.type
turb.ord.
n          y          10         y          minimum
second
flow-mode  solver     rk option  nharms
unsteady   hb         rkex       4
move       freq.      xrotc      yrotc      frame
rotating  -0.000593   0.d0       0.d0       relative
irest      srest      cfl        cdff       lmax
iupdt      toler
0          5000       1.5        4          20         1
1.d-12
rkap       irs-typ    cfli       cutcirs    psi
-1.        cirs_v1    3.0        0          0.0625
cflt       cflit      ramp-opt   n(3)       n(2)
n(1)
2.0        4.0        ramping1   2000       1000       500
cfli(2)    cflit(2)   cfli(1)    cflit(1)
1.5        2.0        0.1        0.1
lim        epslim     cntrpy     etpfxtyp   entfxctf
4          1.d-6      1.d0       0          0.3d0
nlevel     nl_crs     nl_fmg     nstart     npre
npost      ncrs
1          4          1          2          3          2
2
flow-speed
nolomach
tref
288.2
functional
```

```
default 0.0 0.0 0.0
lref1       lref2       lref3
1.0         1.0         1.0
1
76  33 37 41 42 49 50 57 58 125 126 161 165 169 170 197
198 233 234 241 242 341 342 351 356 357 362 371 372 385
390 395 396 405 406 413 418 455 460 465 470 475 480 481
486 499 504 505 510 519 524 529 534 596 601 606 611 616
621 626 631 636 640 645 650 662 667 672 676 681 686 700
705 710 715 720 725
800
```

11.3 256 block unbalanced decomposition input file header

```
*** input file for 3D Euler/NS COSA solver ***
debug       model      flow-type  id
n           sst        external   aircraft
gamma       reyno      pranl      machfs     alpha    beta
1.4d0       1.0d+7     0.71d0     0.1        5.00     0.d0
prant       tkefar     mutfar     wall       roughk
0.9d0       1.d-6      0.1        wilcox     0.d0
posprd      lim_ptke   prdlim     lim_pome
n           n          100        n          minimum
second
flow-mode   solver     rk option  nharms
unsteady    hb         rkex       4
move        freq.      dh0x       dh0y
plunge      0.01       1.0        0.d0
irest       srest      cfl        cdff       lmax
iupdt       toler
 0          10000      1.5        4          200        1
1.d-14
rkap        irs-typ    cfli       cutcirs    psi
-1.         cirs_v1    3.0d0      0          0.25
cflt        cflit      ramp-opt   n(3)       n(2)
n(1)
2.00        4.00       ramping1   1000       500        250
cfli(2)     cflit(2)   cfli(1)    cflit(1)   stop
1.5         2.0        0.5        0.5        50000
lim         epslim     cntrpy     etpfxtyp   entfxctf
4           1.d-6      0.d0       0          0.95d0
nlevel      nl_crs     nl_fmg     nstart     npre
npost       ncrs
3           4          1          3          3          3
6
prol.type              restr.type lim.type
bilinear               ho_rest    lim_corr3
flow-speed
nolomach
tref
```

```
288.2
functional
default    0.25      0.0 0.5
lref1      lref2     lref3
1.0        1.0       1.0
1
32  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30 31 32
256
```