Enhancement of high-order CFD solvers for many-core architecture

Phil Ridley and Yiqi Qiu,
Numerical Algorithms Group Ltd,
Wilkinson House, Jordan Hill Road,
Oxford, OX2 8DR, UK
October 7, 2013

Abstract

In turbomachinery design, Reynolds averaged Navier-Stokes (RANS) models commonly encounter problems when dealing with turbulent flows dominated by unsteady features such as convecting wakes, large scale separations and complex vortical structures. These render both RANS and unsteady RANS theoretically questionable, even for basic jets. On the other hand, Large eddy simulation (LES) is showing tremendous promise, because the turbulent flow structures are fully resolved numerically, rather than approximated (as in the case with RANS). However, LES requires the use of meshes of the order of 100 million grid points, in order to capture the intricate turbulent flow structures. Typical simulations for fourth order space-time correlations of turbulent fluctuations also require long run times (>12hrs). The aim of this project is to improve the performance of the structured CFD codes BOFFS and NEAT, for better use of HECToR and other many-core HPC architectures. A flexible parallel data decomposition will be implemented in BOFFS to improve scalability with complex-grids. Optimisation of the solver and the implementation of thread safe OpenMP in NEAT will also be performed in order to improve performance and scalability.

Table of Contents

1 Introduction

1.1 Overview

CFD is very important in the design of an aircraft engines as full scale tests are too costly to create within a reasonable time. Hence, high-resolution computational aerodynamic models and the use of large HPC resources are essential. Aero engine flows are turbulent and challenging to model because of their wide range of temporal and spatial scales. The turbulent features are usually resolved using the Reynolds Averaged Navier-Stokes (RANS) equations. However, RANS and unsteady RANS have been shown to be inadequate both in accuracy and reliability for design [1]. An exciting and promising method for accurately modelling and hence understanding these types of flows is that of Large-Eddy Simulation (LES) [2]. This does not place such high demands on explicit turbulence models, as most of the flow physics is resolved in time and space rather than attempting to model the mean effect as in RANS.

However, the main disadvantage of LES is that it places huge demands on the computational resources to solve the intricate flow structures. Grids with the order of 100 million points are required to capture near wall streak structures. Furthermore, to accumulate statistical averages such as those used in acoustics, long run times are required to accurately acquire low frequency data.

For this project, two applications which are mainly for aircraft engine design will be optimised for improved performance on HPC architectures: One of which is The Block Overset Fast Flow Solver (BOFFS) and the other is NEAT. These codes are used to model complex flow scenarios with structured multi-block decompositions. They carry less overhead in comparison to unstructured solvers for what is already an expensive solution (5,000 kAUs per run on Phase 3).

BOFFS is used on HECToR to study a wide variety of gas turbine engine components. These include high pressure turbine blades, low pressure turbines, rim seals and labyrinth seals. The code is also used to characterize the effects of surface roughness on turbine blades. Recent simulations have enabled a high fidelity CFD strategy to be developed for turbines that has sufficient accuracy and consistency to (potentially) replace much rig testing, at a reduced cost. In turn, this will enable increased automation within design cycles for the accurate modelling of the complex flows encountered in turbines. The high order numerical scheme and overset approach used in BOFFS allows arbitrarily complex design configurations for structured grids, making it an ideal LES tool.

NEAT is also used to study a variety of complex flows which involve the use of LES, RANS and Direct Numerical Simulation. NEAT makes use of numerous RANS and LES models for structured Cartesian grids, with numerical qualities defined by the discretisation schemes, including several temporal and spatial discretisations with a flexible higher order cell-face interpolation stencil. The code is especially suited to efficiently modelling more canonical turbulent flows using LES and refining RANS models for understanding flow physics. NEAT has also been used internationally, and recently it has been used to accurately model heat transfer in turbine cooling passages and highly accelerated flows in labyrinth seals.

1.2 Description of the BOFFS code

BOFFS uses a high-order method for solving the 3D unsteady Navier-Stokes equations on structured hexahedral meshes, combined with a multi-block overset (or Chimera) grid capability for arbitrarily complex geometries. BOFFS can be used for RANS, Unsteady RANS and LES of turbulent flows. The code is mostly used with overset structured grids to perform LES of turbulent flows, mainly for turbomachinery applications.

The convection terms are solved using flux difference splitting methods based on the Monotone Upstream Centred Scheme (MUSCL) approach of Van Leer [3]. Up to sixth-order central differencing is possible for the inviscid terms. For the time integration, a Galerkin scheme [4] is used with dual time-stepping and sub-iterations in pseudo-time with an infinite pseudo timestep. For the solution of the linear equations required by the implicit relaxation scheme over planes or lines, a tri-diagonal matrix algorithm (TDMA) and a Gauss-Seidel (GS) scheme are available.

The multi-block nature of BOFFS provides a convenient platform for parallel data decomposition, which is a common feature shared by many multi-block codes. Therefore, BOFFS may be instrumented in four different ways: serial (no parallelisation and no block decomposition), OpenMP-parallel (no block decomposition), MPI-parallel (with a decomposition over the grid blocks) and MPI/OpenMP parallel (MPI with the addition of OpenMP for the TDMA and GS routines). In the serial case each block is processed sequentially. With the parallel cases, at each sub-iteration (for unsteady calculations) or global iteration (for steady calculations), pressure, velocity components and scalar variables are transferred between blocks via MPI.

1.3 Description of the NEAT code

NEAT is a 3D incompressible flow solver which is primarily used for LES. The code uses structured Cartesian meshes that are staggered in space. Arbitrary solid regions can also be defined, allowing complex flows to be studied. In addition to LES, NEAT can also be used for RANS and unsteady RANS containing a wide variety of turbulence models for each. The code also has a multi-grid capability for faster convergence. Currently, NEAT is mainly used for LES involving heat transfer applications including turbomachinery, electronics cooling and other areas.

The core solver uses second order central differencing to obtain cell face fluxes and the second order Crank Nicolson scheme is used for time advancement. Higher order discretisation is also available. Each time step will involve a number of inner iterations to converge for each variable. Pressure and velocity are coupled using the Semi-Implicit Method for Pressure Linked Equations (SIMPLE) scheme. The staggered grid helps to avoid numerical issues (such as pressure checker-boarding). Finally, the GS and TDMA schemes are used to solve for lines, planes and volumes of grid points.

1.4 Outline of this project

This software development project will develop both BOFFS and NEAT so that they may both run more efficiently on HECToR and other many-core HPC architectures. The overall aim of the work is to further turbomachinery research by enabling the efficient use of these complex flow solvers with grids with a 100 million points and thousands of blocks. Although both BOFFS and NEAT are structured multi-block codes, there are several functional differences between them. NEAT differs from BOFFS in that it uses a staggered grid for pressure velocity coupling, whereas BOFFS uses curvilinear meshes and explicit smoothing with a special scheme to avoid pressure checker-boarding. The original objectives for this project were as follows.

WP1: Implement a flexible approach to MPI process allocation in BOFFS

The original parallel decomposition in the BOFFS code was designed such that each grid block was assigned to a single MPI process. Additional parallelism within each MPI process would be achieved by assigning a team of OpenMP threads for computations within the grid block. A new MPI decomposition will be implemented such that several blocks may be combined together, forming multi-block data for each MPI process.

There are two communications subroutines in BOFFS: one which takes care of the pack and send (packvar_send) and the other for receive and unpack (recvvar_unpack) between buffers. These subroutines will be redeveloped to allow a flexible approach to specifying a varying number of blocks per MPI processes and the facility to read in a user specified configuration file 'mpi_cfg.in' which specifies the block allocation will also be implemented.

To demonstrate the effectiveness of the flexible decomposition, a test case with 50 million cells and 100 grid blocks with a complex structure will be run on HECToR. This will show at least 20% reduction in run-time for the flexible decomposition compared with the original one block per MPI process decomposition.

WP2: Implement OpenMP sections in the TDMA and GS routines in NEAT

Since the structured multi-block nature of NEAT is shared with BOFFS, the parallel decomposition was also one grid block per MPI process. However, unlike BOFFS, multi-block process assignment is not the foremost concern to improve run-time in NEAT, since typically 50% or more of the overall run-time is consumed by the linear equations solver and subsidiary routines. There are two linear solvers available in NEAT: TDMA and GS. Either solver may be used for each plane (x,y or z) in order to achieve optimal numerical convergence, and this will depend upon the nature of the flow.

The TDMA and GS routines are in the subroutine solve3, which consists of over 1400 lines of code. This part of the work will be to implement thread safe OpenMP within these routines, and there will be at least 10 regions where loops will need to be developed. The solve3 routine will allow most of the variables to be declared PRIVATE within threads allowing the use of CRITICAL to be avoided wherever possible and DYNAMIC scheduling with the NO WAIT clause to be used wherever possible. To achieve efficient mixed-mode performance within the MPI parts of the code, only the master thread will make the MPI calls. For large enough problems, this work will enable NEAT to utilise more cores per MPI process.

The new mixed-mode parallel decomposition will be such that each MPI process will still work with a single block but with the addition of intra-block OpenMP parallelisation for the TDMA and GS solvers. To demonstrate the effectiveness of the new mixed-mode developed code, a test case for an unsteady flow using 128 blocks and 8 OpenMP threads (128 MPI processes with 8 threads per process) will be compared to the original code. The expected performance improvement will be a 20% reduction in run-time.

WP3: Implement OpenMP in the subsidiary routines of NEAT

The 50% overall run-time consumed by the linear equations solver and subsidiary routines in NEAT is roughly made up as 40% and 10% respectively. Therefore the remaining part of the work will be to implement thread safe OpenMP in the subsidiary routines. These are used to calculate the coefficients and source terms to weight the nodal variables for the U,V,W velocities, the pressure and the temperature: coeffu, coeffv, coeffw, coeffp and coefft. Each one of these routines contains around 200 lines of code and either two or three loops which will require OpenMP parallelisation.

To demonstrate the performance of the combined OpenMP developments, the same unsteady flow test case as in WP2 will be demonstrated. A further 3% reduction in run-time will be expected when compared with the code developed from WP2.

2 Implementation on HECToR Phase 3

2.1 BOFFS

BOFFS is written in Fortran 90 and consists of around 22,000 lines of code. MPI is used for communication of the physical flow variables: pressure, velocities and temperature, when the code is run with a block decomposition. OpenMP is used for the optional parallelisation of the TDMA and GS routines. The main sections of code concern grid pre-processing (and flow processing for restarts), time advancement and solution post processing. The output files for visualisation are written as binary in PLOT3D format along with restart files.

For the main calculation firstly the boundary conditions and metric terms are set, then for the main time-step loop, the RHS side of the equations will be constructed and then solved using either a TDMA and GS implicit relaxation scheme. The choice of solver will depend upon the mesh and flow characteristics. The boundaries and arrays will then be updated prior to the next iteration. This simple method of solution is a popular choice for many CFD solvers since it has a low storage requirement relative to efficiency.

2.2 Performance of BOFFS

BOFFS is capable of scaling well to at least 1000 cores with non-complex grids (i.e. those grids having either one or two neighbours per block). Such grids may be divided into 'blocks', where each one of these 'blocks' is assigned to a single MPI process. However, for complex grid configurations with overset meshes, there will usually be a variation in the number of neighbours per block, which is unsuitable for this method of block decomposition.

In such configurations, there will usually be a central section of the grid e.g. several cooling holes, each of which must be treated as separate blocks due to their specific mesh type, and 'core' blocks which have fewer grid points. The block division is governed by the mesh topology and therefore, the original MPI data decomposition for the grid, resulted in a very uneven distribution of neighbours per block, which severely limits scalability. Therefore, WP1 will be to implement a flexible decomposition to allow multiple blocks to be assigned per MPI process and therefore improve the performance of the data transfers.

2.3 NEAT

NEAT is also written in Fortran 90 and consists of around 30,000 lines of code with around 20 subroutines. As with BOFFS, a block decomposition is used with MPI for the communication of the physical flow variables: pressure, velocities and temperature.

At the beginning of each simulation, the pre-processing includes wall distance calculations and the calculation of polynomial coefficients for higher order schemes (if required). The implicit solver used in the main time-step loop is performed using either a TDMA and GS implicit relaxation scheme. At the end of a simulation, flow data is written to binary restart files and ASCII files in Tecplot360 format for visualisation.

2.4 Performance of NEAT

The parallel performance and efficiency of NEAT is good for certain grids and problem sizes, however this reduces for larger grids and for more than 256 cores. Profiling analysis showed that around 46% of the run-time is taken up by solve3 (for the TDMA or GS matrix solver), followed by a combined time of around 29% for the subsidiary routines which calculate the coefficients and source terms to weight the nodal variables for the U,V,W velocities, the pressure and the temperature (coeffu, coeffv, coeffw, coeffp and coefft). The inefficiency in the solve3 routine causes an imbalance in the timings for each MPI process, with the slowest MPI process taking twice as long as the fastest.

Therefore WP2 will be to improve the scalability performance of NEAT by implementing OpenMP within the main loops for the solvers in solve3 and also in the subsidiary routines (WP3). A red-black approach with OpenMP will be used for the TDMA and GS solvers to enable each MPI process (or block) to run in an efficient hybrid mixed-mode. The use of OpenMP also has the advantage of allowing higher core counts with fewer grid blocks for larger sized problem.

3 Development of a flexible approach to the block allocation in BOFFS

3.1 Implementation

The first development was to replace the array, exchangematrix(1:MaxNBlkMPI, 1:MaxNBlkMPI). This was used to store the size of the MPI message buffers for data passed between neighbouring processes. This array was replicated on each of the total number of MPI processes, MaxNBlkMPI. As each process only needs to know the size of neighbouring data buffers for receiving from or sending to, it is more efficient to store these sizes in the local arrays RecSize(1:MaxNBlkMPI) and SendSize(1:MaxNBlkMPI).

The original MPI decomposition in BOFFS was fixed at one grid block per MPI process, so the second development was to add the facility for a user configuration file, mpi_cfg.in, for storing the data for describing the flexible MPI process allocation. This file will be read in by the master process and then broadcast to the others. The following data will be read in (where MaxNBlk$_i$ is the actual number of blocks on each process and MaxNBlkMPI is the total number of MPI processes):

value of MaxNBlk$_i$ for process i (n_i), …, (i=0, MaxNBlkMPI-1)
list of block numbers for process i (n_i entries), …, (i=0, MaxNBlkMPI-1)

(where $\sum n_i$ = block_number i.e. total number of blocks)

E.g. to assign 12 blocks to 12 MPI processes, with 1 block per MPI process

```
1,1,1,1,1,1,1,1,1,1,1,1
1,2,3,4,5,6,7,8,9,10,11,12
```

Or, to assign 12 blocks to 5 MPI processes

```
3,3,3,2,1
1,2,3       4,5,6       7,8,9       10,11       12
```

The list of block numbers for process i must be in ascending numerical order due to the method which will be used for determining the block per process assignment. The data will be copied to the array num_rankblock(1:MaxNBlkMPI), which will store the values of MaxNBlk for each process; and rank_block(1: num_rankblock(1: MaxNBlkMPI),MaxNBlkMPI), for the list of local to global block indices. These arrays will be replicated on each process.

By comparing the serial form (mpi_flag==0) and the parallel form (mpi_flag==1), it was observed that those parts of BOFFS affected by any modification to the original decomposition, would have the following loop structure (where numblocks is the total number of blocks):

```
 IF (mpi_flag==0) THEN
   DO i = 1, numblocks
       …
       CALL currentblock(i) ! set offsets for data
   Perform calculations on local arrays, e.g. pval(i,:,:,:)
       …
      END IF
   END DO
 ENDIF
 IF (mpi_flag==1) THEN
   …
   CALL currentblock(my_rank) ! set offsets for data
  Perform calculations on local arrays, e.g. pval(my_rank,:,:,:)
   …
 ENDIF
```

In parallel form, each process deals only with its own block (i.e. originally one block and several blocks for the new flexible approach). But the serial form works sequentially through all blocks. This was a useful observation, since for the new block decomposition, the main difference will be that MaxNBlk may vary in size for each MPI process. So the next update involved replacing all instances of the previous loop structure, with the following:

```
DO i = 1, numblocks
    CALL currentblock(i)   ! set offsets for data
    setblock = .FALSE.
     IF (mpi_flag==0) THEN
       setblock = .TRUE.
       nblock = i
     ENDIF
     IF (mpi_flag==1) THEN
        nrcheck = num_rankblock(my_rank)
        DO rcheck = 1, nrcheck
          IF (rankblock(rcheck,my_rank)==i) THEN
             setblock = .TRUE.
             nblock =  nblock+1
          ENDIF
       END DO
  ENDIF
IF (setblock) THEN
…
Perform calculations on local arrays, e.g. pval(nblock,:,:,:)! where nblock<MaxNBlk
…
ENDIF
ENDDO
```

Therefore, each particular MPI process will only deal with its own blocks (when setblock is TRUE). To implement this new loop structure throughout was time consuming, as there are over 105 such cases. Three further routines were also updated, to take account of the extra data storage required: externalblockboundariesmpi, packvar_send and recvvar_unpack.

The subroutine externalblockboundariesmpi performs updates to the local variables. This subroutine is called before sending updates to the neighbouring block boundaries, and after receiving updates for the boundaries from neighbouring MPI processes. The original block decomposition was fixed at one block per MPI process, but for the new flexible approach, there may be more than one block. Therefore, externalblockboundariesmpi would need to be updated to handle both local blocks (i.e. those blocks assigned to the same MPI process) and external blocks (i.e. those blocks assigned to neighbouring MPI processes).

To implement a method for distinguishing between local and external blocks, a LOGICAL array, mpineighs(1: MaxNBlk) was defined on each MPI process. The entries for this array were defined by performing a once only check between each local block and all other local blocks. This was implemented using the replicated array, rank_block (list of local to global block indices), such that mpineighs(i) is FALSE for a local block, or TRUE for an external one. Therefore, the new method for updating the local variables in externalblockboundariesmpi, will firstly be performed for the external blocks, and then secondly, for the other local block boundaries (i.e. excluding the local block itself).

The subroutines packvar_send and recvvar_unpack perform the packing and unpacking of the data buffers. Before describing the final developments, it is worthwhile to mention that there will need to be some consideration for the order of the blocks when packing and unpacking. This was not required with the original allocation of one block per MPI process. So for the ordering, the local array rankneigh(1:3, 1:MaxNBlk) has been added for each MPI process. This array contains the global indices of the block data that will need to be copied from (from_block), the global indices of the block data that needs to be copied to (to_block) and the indices of the neighbouring MPI processes (so that the global number of the MPI process can then be determined).

To achieve the correct order when unpacking the buffers for the block data, a further local array, unpackord(1: MaxNBlk) was also added. This contains a look up table for the indirect addressing of the blocks to be unpacked. It is necessary to perform the unpacking of the blocks in their correct order since every single MPI buffer may contain the data for several blocks. Therefore each MPI process will need to know the order in which the original data was packed. E.g. if MPI process 1 and MPI process 3 contain neighbouring blocks, to send data from process 1 to 3, it may be that data from process 1 is packed in the order of (1,8) (1,9) (2,9) (3,7) but received from process 3 as (7,3) (8,1) (9,1) (9,2) (i.e. the same order that it was packed on process 3); in this case, the correct unpacking order for data from process 3 would be found by accessing unpackord.

Finally, to correctly handle the buffers when packing or unpacking the data, the offset of where the new data is to be added to and taken from the buffers, must also be stored. Several local variables for the block boundaries need to be communicated to one or more different blocks, therefore the array countn(1: MaxNBlk) was added. This is for storing the starting location of where the next local block data from each neighbouring MPI process is stored within the data buffer.

3.2 Comparison of Performance

Performance of the new BOFFS code with the flexible allocation of blocks was demonstrated on HECToR Phase 3. The test case was a subsonic jet LES, with 50 million cells and 108 grid blocks. A transient state for this case is shown in Figure 1. A complex grid configuration is used, as shown in Figure 2(a) and 2(b), where different blocks are represented by separate colours. To avoid an axis singularity there are central ‘core’ blocks and this region is represented by the thin rectangular area, which is marked in black just above the horizontal line in Figure 2(a). It is worthwhile to note that the non-central ‘core’ blocks contain relatively fewer points in the x and y directions, which is extended throughout the domain. The original method of allocating one block per MPI process would lead to a large number of neighbouring blocks, as shown by the separately coloured blocks which surround the central ‘core’ region in Figure 2(a). Using the new flexible allocation of blocks, the ‘core’ region may be split into many smaller blocks (i.e. multiple ‘core’ blocks), as shown in Figure 2(b). Here, assigning localised surrounding blocks to the same MPI processes leads to a similar number of neighbours for each MPI process which gives improved load balancing and scalability.

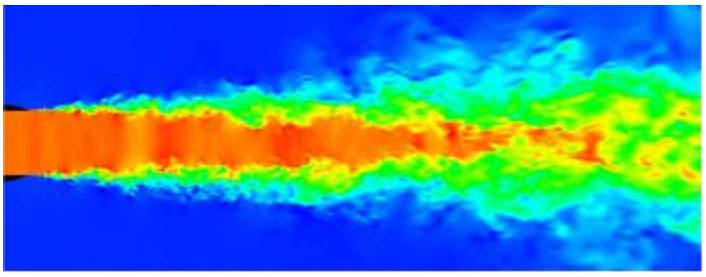
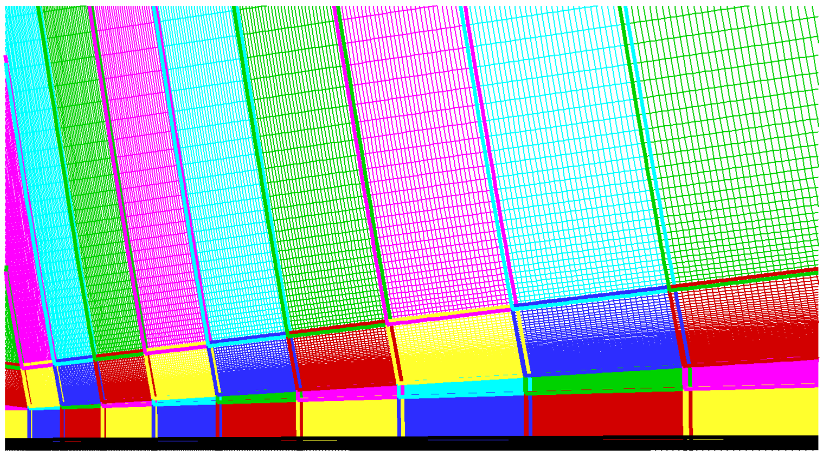
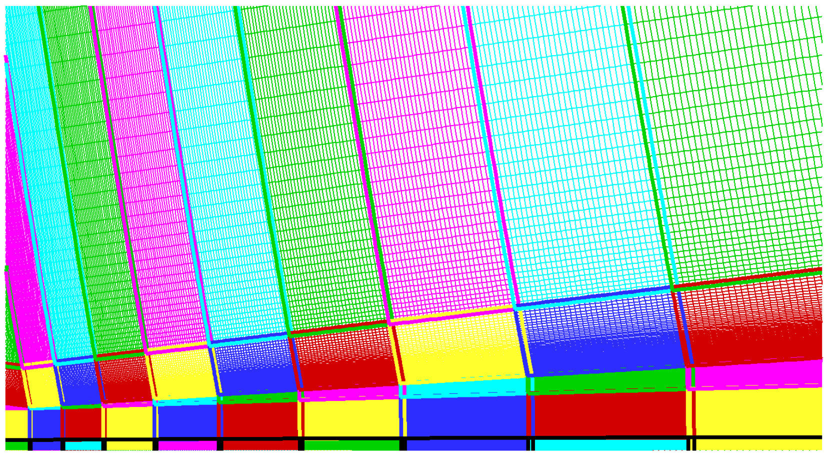


Figure 1: U-velocity contours for a high LES of a jet, using 50 million cells.



(a)



(b)

Figure 2: (a) block structure with single 'core' block for the jet mesh, (b) block structure with multiple 'core' blocks for the jet mesh.

The wall clock timings to perform 10 time step iterations are given in Table 1, along with the parameters used for aprun. The results are shown for GCC 4.7.2 and PGI 12.10.0. With PGI, the flags used were -fast Mipa=fast, and with GCC, -Ofast -funroll-loops -ftree-vectorize.

The rows where N=108 concern the original method of allocating one block per each MPI process, and those with N=54 are for the new flexible allocation of blocks. It is worthwhile to note that GCC 4.7.2 gives better overall performance than PGI 12.10.0. Also, the maximum number of MPI processes per HECToR node is limited to 4 for the block structure with a single 'core' block, this is because of the memory requirements of the larger blocks. For the block structure with multiple core blocks, the memory required per block is less and therefore it is possible to allocate 8 MPI processes per node (with a maximum of 2 threads per process).

N	n	d	S	mppwidth	Time(s) GCC	Time(s) PGI
108	4	1	1	864	144.92	151.81
108	4	2	1	864	102.79	114.13
108	4	4	1	864	78.76	89.23
108	4	8	1	864	67.28	78.87
54	8	1	2	224	231.61	243.08
54	8	2	2	224	124.05	132.77
54	4	4	1	448	72.18	77.70
54	4	8	1	448	50.10	53.84

Table 1: Timings (in seconds) to perform 10 iterations with 50 million cells and 108 grid blocks.

The aim of this work was to be able to show at least 20% reduction in run-time for the new flexible decomposition compared with the original one block per MPI process decomposition. Far better performance than expected can now be achieved with BOFFS, e.g. comparing the use of 8 threads for the single 'core' block case (where N=108) with the multiple core blocks (where N=54), the wall clock time is reduced from 67.28s to 50.10s (with GCC) using nearly half the number of cores.

4 Development of OpenMP for the Linear Solvers in NEAT

4.1 Implementation

In NEAT, the MPI data decomposition is based on the multi-block structure which works well. However, for a typical run at least 50% of the overall run-time is consumed by the linear equations solver and subsidiary routines. The two linear solvers available in NEAT: TDMA and GS, may be employed in each plane (x,y or z), and either solver may be used in order to achieve optimal numerical convergence, depending upon the nature of the flow. Both the TDMA and GS schemes consist of calculations over 3D-nested loops, and although these are widely used schemes, NEAT uses modified versions in order to exploit flow features for faster convergence.

The TDMA and GS routines are in the subroutine solve3, which consists of over 1400 lines of code with around 18 main loops for TDMA x (non-periodic), TDMA x (periodic), GS x, TDMA y, GS y, TDMA z (non-periodic), TDMA z (periodic), GS z. Before implementing the OpenMP code, it was noticed that restructuring the main loops would be beneficial to performance.

For WP2, the first update for the new code (version 1) was to ensure that the loops for TDMA and GS were ordered for improved memory access. The TDMA / GS x with z,y,x; TDMA / GS y with x,z,y and TDMA / GS z with y,x,z (ordered outer most to inner most loop). These calculations use pre-determined coefficients for the off-diagonal terms and are stored within 3D arrays. Therefore, it was only possible to arrange the TDMA and GS calculations for x with the inner most loop accessing the first array index. This could not be done for y and z, since each inner most loop has to be performed within that particular plane.

Secondly, wherever possible IF … THEN conditions were either moved outside the loops or restructured to ensure that the minimum amount of branching was being performed within the loops, particularly with the inner-most loops. This was possible as most branches were related to boundary conditions which are fixed throughout the run e.g.

```
DO i=isq,id-1
IF (i==isq)THEN … ENDIF
IF (i==id-1)THEN … ENDIF
END DO
```

was replaced with

```
computation with i=isq
DO i=isq+1,id-2
computation with i=isq+1,id-2
END DO
computation with i=id-1
```

Thirdly, due to the removal of the IF … THEN conditions within the inner-most loop, it was noticed that vectorisation was achievable for the loops in TDMA x and GS x. However, a separate branch of the code (version 2) was developed for this, due to the routines in solve3 requiring a lot of extra code. The main development included the implementation of a new array, nblr, which is used as a replacement for an IF … THEN condition related to the boundary. This was achieved by setting nblr to 0.0 or 1.0 at each grid point, depending upon whether the point is a boundary one or not. The boundary condition can then be set via multiplication, rather than by performing the IF … THEN for each grid point.

The next part of the work was to Implement thread safe OpenMP for the outer loops for TDMA x (non-periodic), TDMA x (periodic), GS x, TDMA y, GS y, TDMA z (non-periodic) and TDMA z (periodic). This was performed using STATIC scheduling, which gave the best performance for the test case problem. As well as the addition of OpenMP to these loops, a red-black decomposition was also implemented for GS x, GS y and GS z. Initially the loops were in a natural ordering. The red-black ordering means that the outer most loops are split in two: one for the odd index and another for the even index. The calculation within each of these loops may then be performed in parallel with OpenMP, it is a widely used standard approach. A 7-point stencil (3D) is used for the solution of the Navier-Stokes equations, therefore care was required to ensure that the actual loop ordering does give thread safe answers.

Following a similar approach for WP3, the subsidiary routines in NEAT were also updated. These are in the files: p.F, t.F, u.F, v.F and w.F. The routines are used for calculating the coefficients and source terms which are used to weight the nodal variables for the U,V,W velocities, the pressure and the temperature: coeffu, coeffv, coeffw, coeffp and coefft. Many two-level loops were reordered for improved memory access e.g.

```
      DO i=2,id-1
        DO k=ks,ke
          IF(nbl(i,jd,k)==-1.AND.nbl(i-1,jd,k)==-1)THEN
              IF(((v(i,jd,k)+v(i-1,jd,k))/2)>0.0)THEN … ENDIF
           ENDIF
        ENDDO
      ENDDO
```

was replaced with

```
      DO k=ks,ke
        DO i=2,id-1
          IF(nbl(i,jd,k)==-1.AND.nbl(i-1,jd,k)==-1)THEN
             IF(((v(i,jd,k)+v(i-1,jd,k))/2)>0.0)THEN … ENDIF
           ENDIF
        ENDDO
      ENDDO
```

Overall, the OpenMP parallel decomposition works well with the original MPI data decomposition and because this level of parallelism is contained only within the TDMA and GS solvers, there is no need to be concerned about thread safety with any other parts of the code which use MPI.

4.2 Comparison of Performance

Performance of the new code when compared with the original is shown in Figure 3. This test case is for an unsteady flow over a structured grid with 12,642,048 points and 128 blocks (i.e. 98,766 grid points per block). The results from HECToR are shown for GCC 4.7.2 and PGI 12.10.0. With PGI, the flags used were -fast Mipa=fast, and with GCC, -Ofast -funroll-loops -ftree-vectorize. The horizontal axis shows results for three versions of the code: the original, new version 1 (with the developments described in the previous section, apart from the new array, nblr) and new version 2 (with the additional developments for the array nblr, to enable the compilers to achieve some further vectorisation).

Each run was performed for 1000 time steps and timings were taken for the computation only, with no I/O. The number of MPI processes was fixed at 128, which is also the number of blocks in the grid. However, the total number of cores used ranged from 128 to 1024 to vary process placement. With more than 128 cores, although some cores are not used, it was observed that between 28%-36% reduction in run-time can be achieved. With no OpenMP (d=1), it is more costly to run a single-threaded job in this way, however with OpenMP and for larger problem sizes, this could be beneficial.

Overall in Figure 3, even without any OpenMP, there is already a 24% reduction in run-time for the new code (version 1 and version 2) compared with the original code. Furthermore, for the MPI only NEAT, PGI 12.10.0 gave an overall better performance than GCC 4.7.2.

Performance of the new code with OpenMP is shown in Figure 4. Here, the same test case was used as the one used to generate the results shown in Figure 3. This time, NEAT was run in hybrid parallel mode, with OpenMP for the TDMA and GS routines, as developed in WP2. The first point to notice about the results was that here, GCC 4.7.2 gave an overall better performance than PGI 12.10.0. Scalability is reasonable for up to 8 OpenMP threads, comparing the run with mppwidth=1024 and 8 threads in Figure 4 with the corresponding four runs with mppwidth=1024 in Figure 3. For the original code (Figure 3), MPI only timings were 75.1357s with GCC 4.7.2 and 73.9947s with PGI 12.10.0. For the new version 1 (Figure 3), these were 63.736s with GCC 4.7.2 and 61.3346s with PGI 12.10.0. In Figure 4, using 8 OpenMP threads (with the same number of cores), the run-time is 36.1021s with GCC 4.7.2 and 45.7929s with PGI 12.10.0. Comparing these figures with the results for the original code, the total run-time for the new version is less than half that of the original. Please note that as this comparison is between the original code and new version, there was already a 24% reduction in run-time, without any OpenMP. Also, the section of code which has been developed consumes just over half of the total run-time, therefore this comparison has been made to highlight the original aim of achieving a 20% overall speedup, against the result of 50%, which is better than expected.

Performing the same test case again, but with OpenMP for the TDMA and GS routines, and also the subsidiary routines as developed in WP3, it was observed that a further 3% reduction in run-time could be achieved for this particular test case.

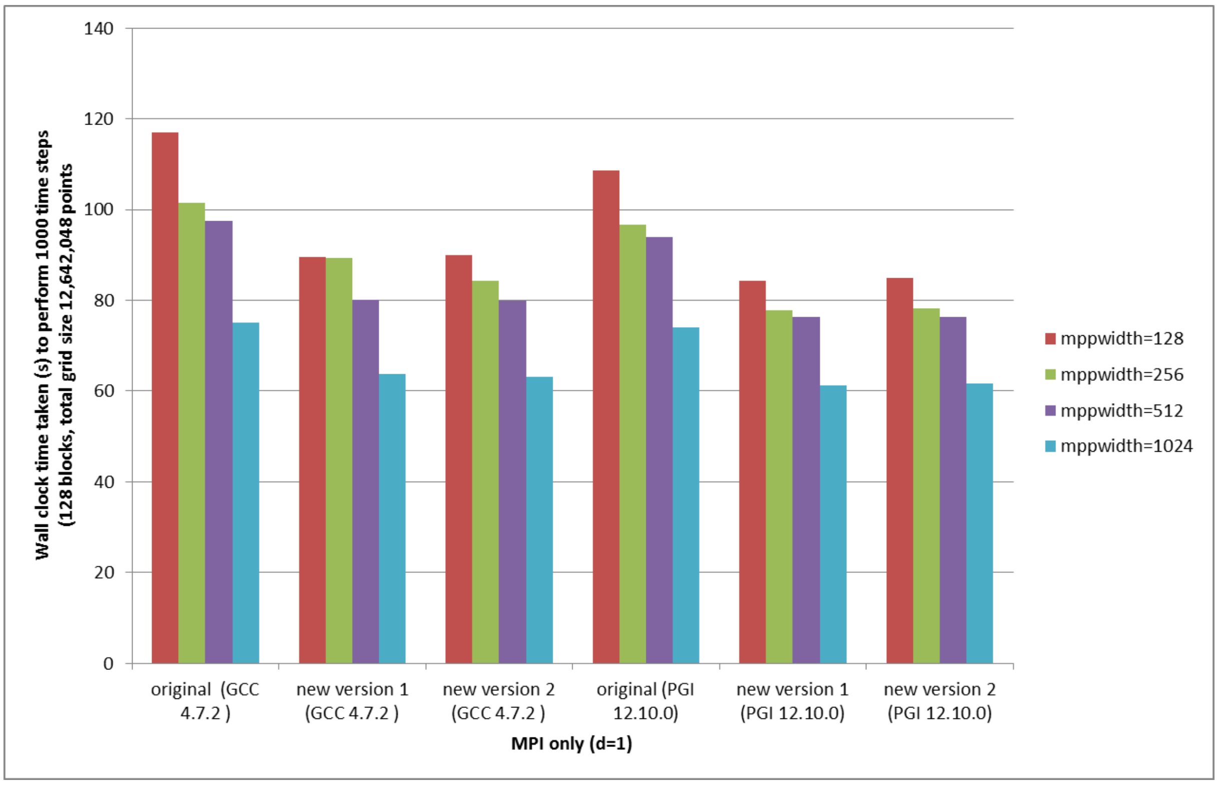


Figure 3: Comparison between compilers and versions with no OpenMP (MPI only).

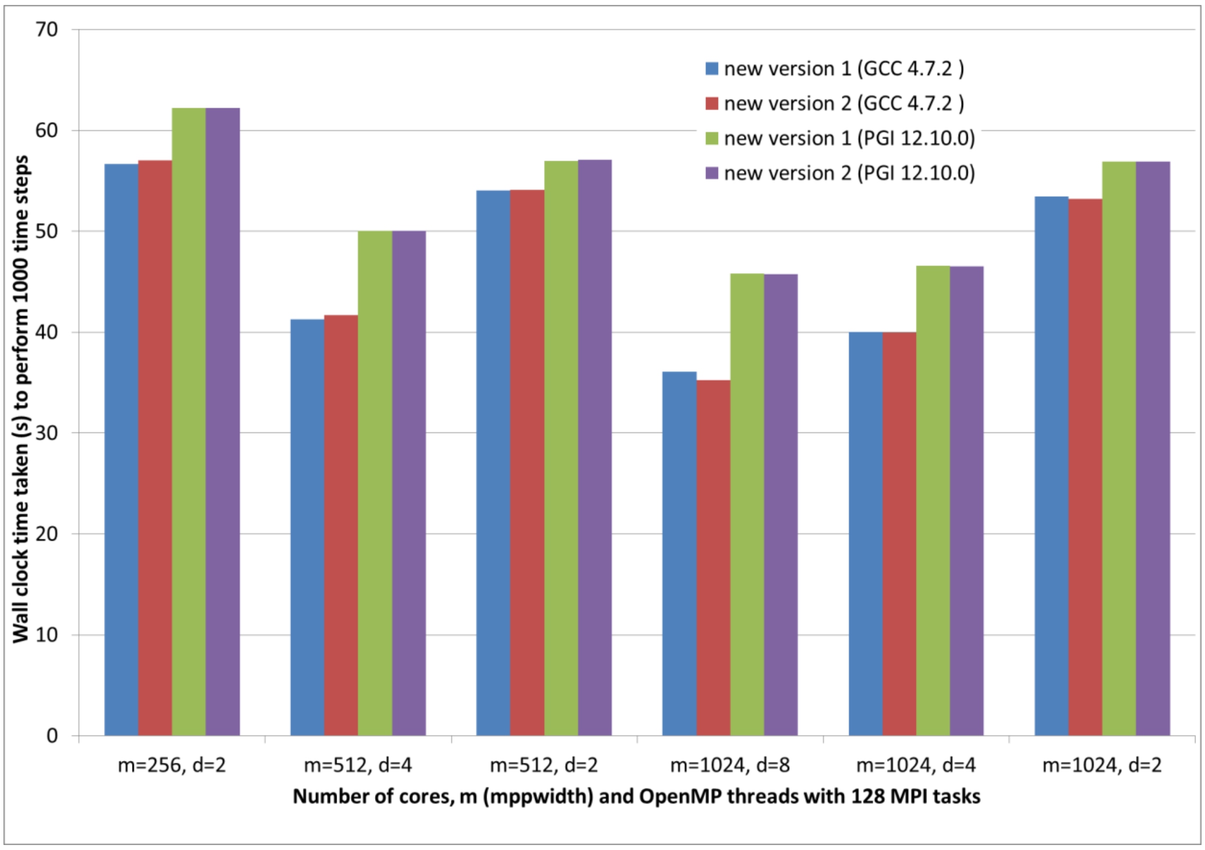


Figure 4: Comparison between compilers for the improved code with OpenMP.

5 Conclusions

The overall objective of this project was to improve the parallel implementation of two separate structured multi-block codes (BOFFS and NEAT) for improved performance on HECToR and future high-end HPC architectures. For BOFFS, this was achieved by implementing a flexible parallel data decomposition, and for NEAT by implementing hybrid parallelism within the most computational part of the code (the tri-diagonal matrix, Gauss-Seidel and subsidiary routines) via the use of OpenMP.

For BOFFS, the aim was to demonstrate at least 20% reduction in run-time for a representative 50 million cell test case using around 100 grid blocks with a complex structure, when comparing the new flexible decomposition with the original one block per MPI process decomposition. The new flexible data decomposition for BOFFS was successfully implemented, which now allows an arbitrary number of blocks to be assigned to the same MPI process. A demonstration of the capability of the flexible decomposition for a subsonic jet LES, with 50 million cells and 108 grid blocks showed that far better performance than expected can now be achieved with BOFFS. Comparing the use of 8 threads for the single 'core' block case (where N=108) with the multiple 'core' blocks (where N=54), showed that the wall clock time is reduced from 67.28s to 50.10s (with GCC) using nearly half the number of cores.

For NEAT, the aim was to achieve a 20% reduction in run-time for the new mixed-mode developed TDMA and GS solvers, for a representative unsteady flow test case with 128 blocks and 8 OpenMP threads (128 MPI processes with 8 threads per process), when compared with using 1024 cores for the MPI only code. A further 3% reduction in run-time would also be expected for the addition of OpenMP to the subsidiary routines.

A new threaded red-black ordering was implemented for the TDMA and GS routines in NEAT, along with improved memory access for the main computational loops (including those in subsidiary routines). A test case for an unsteady flow over a structured grid with 12,642,048 points and 128 blocks (i.e. 98,766 grid points per block) was performed for 1000 time steps. To demonstrate the better performance due to the improved memory access and restructured loops (i.e. without OpenMP), original and new versions of the MPI only code were compared and a 24% reduction in run-time was observed for the new code (version 1 and version 2). This was by a comparison with the original MPI only code, using fully populated nodes. Demonstrating performance with more cores (mppwidth=1024), only 16-18% reduction in run-time was observed as the original code also performed better when nodes were not fully populated. Furthermore, for the MPI only NEAT, PGI 12.10.0 gave an overall better performance than GCC 4.7.2.

To demonstrate performance of the new code in hybrid parallel mode, with OpenMP for the TDMA and GS routines, the same test case was used. Here, GCC 4.7.2 gave an overall better performance than PGI 12.10.0. Reasonable scalability for up to 8 OpenMP threads was observed for the run with mppwidth=1024 and 8 threads when compared with the four MPI-only runs, with mppwidth=1024. Comparing these figures with the results for the original code, the total run-time for the new version is less than half that of the original, which is good considering that the original aim was to achieve a reduction of 20% of the total run-time.

This work will enable BOFFS and NEAT to be used on high-end HPC architectures for a variety of problems for turbomachinery, heat and cooling film technology. In particular, this DCSE project will now allow BOFFS to use HECToR and ARCHER to facilitate the development and testing of new ideas which have been generated from previous research. So far HECToR usage of BOFFS stands at over 50,000 kAUs, this has been used for the investigation of the aerodynamics and aeroacoustics of complex geometry hot jets through EPSRC research grants EP/G027633/1, EP/G069581/1, EP/F005954/1, EP/I017771/1. The code is also used within groups at Cambridge, Warwick and Cranfield Universities, e.g. for simulations of a Boeing Fan in EPSRC project EP/I010440/1.

6 Acknowledgments

This project was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR - A Research Councils UK High End Computing Service - is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd. http://www.hector.ac.uk

For useful discussions thanks to the lead developer of BOFFS, Dr Richard Jefferson-Loveday, and the lead developer of NEAT Dr James Tyacke, who are both at the University of Cambridge.

References

[1] M. A. Leschziner, "Turbulence Modelling for Separated Flows with Anisotropy-Resolving Closures", *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, Vol. 358, pp. 3247-3277.
[2] C. Klostermeier, "Investigation into the Capability of Large Eddy Simulation for Turbomachinery Design", University of Cambridge, 2008.
[3] B. van Leer, Upwind-Difference Methods for Aerodynamic Problems Governed by the Euler Equations, Lectures in Applied Mathematics Vol. 22, pp. 327-336, 1985.
[4] P. G. Tucker, *Computation of Unsteady Internal Flows*, Kluwer Academic Publishers, Norwell, MA, 2001.