HPC Challenge 2009, Category A
Oct, 2009

A Hybrid Meshfree-Cartesian Scheme to Simulate Complex Moving Body Problems in Three-Dimensional Space

Yu Peng

Department of Mechanical Engineering

1. Background

Flows involving moving bodies/boundaries are frequently encountered in engineering applications and everyday living. Examples are flows in turbomachines, artificial heart valves and extrusion processes [1]. Further examples include flows associated with store separation problems during flight and self-propulsion problems in fish swimming and bird/insect flight etc. These are unsteady flows in which the fluid boundaries are constantly changing, either in a prescribed manner or in a manner determined by forces of interaction between the phases.

2. Objectives

The flows involved moving bodies have attracted considerable attention over the years, and important contributions have been made both in methods and applications [2]. Three main groups of numerical methods are currently available for dealing with this type of flows. They are the mesh-based finite element and finite volume methods, Cartesian grid methods and the overset or chimera grid methods. These existing approaches have their own inherent advantages and drawbacks. Such drawbacks often limit the further development of the approaches towards more complex applications. Besides the limitations on numerical methods, the complicate flow phenomena associated with the moving boundary are still far from fully understood.

Taking into consideration the advantages and limitations of the various approaches for modeling moving boundary flow problems, there clearly exists an opportunity to develop a fresh approach, in which one hopes to minimize some of the cumbersome and potentially error-prone features of existing schemes, and to strengthen numerical representation in the area of the boundary. To achieve this goal, a novel hybrid meshfree-Cartesian grid approach is adopted here [3-5]. In this approach, an enveloping cloud of meshfree nodes discretizes the moving body and its immediate fluid neighbourhood. The moving body/boundary and its enveloping meshfree nodal cloud are superposed on a background space of Cartesian grid points.

3. Significance and Impact of Research

The present method offers the following advantages by combining the meshfree and Cartesian approaches: (1) easy to build high-order scheme, (2) good efficiency as the bulk computation is based on standard 7-point finite difference scheme, (3) precise implementation of boundary conditions, (4) grid flexibility including the ability to accommodate complex geometry with good grid resolution and grid orthogonality at boundaries, (5) grid administration is relatively simple and (6) grid/mesh regeneration frequently used in mesh-based methods is avoided and data interpolation is kept at a very minimal level.

Besides the academic merits mentioned above, the present approach can be applied to study many moving boundary flow problems encountered in engineering applications and everyday living. One of interesting examples is the biological flier such as insects using flapping wing to generate high lift and thrust in fluids. Such flow phenomena induced by flapping wing have attracted a great deal of interest recently as scientists and engineers strive to understand the complex and often novel aerodynamics underlying the flight of insect. Significant contributions to better understanding of flying mechanism may be made by the application of the present approach. Such understandings may spark novel ideas on engineering applications such as flow control, thrust production etc. They may also help us to develop novel propulsive mechanisms that emulate the motion of flapping wing for unmanned aerial vehicles (UAVs). Other interesting applications include the blood flow in human heart, flows in turbomachines etc.

4. Numerical Aspects

The dynamics of three dimensional flow of the incompressible and viscous fluid is governed by the arbitrary Lagrangian-Eulerian (ALE) formulation of the three-dimensional Navier-Stokes equation [6]. The moving body is embedded and enveloped by a cloud of mesh-free nodes, which convects with the motion of the body against a background of Cartesian nodes. Spatial discretization is accomplished by the combination of a Generalized Finite Difference (GFD) method and conventional finite difference method (FDM) applied to the meshfree and Cartesian nodes respectively. The mixed Lagrangian-Eulerian form of the Navier-Stokes equations on convecting nodes is integrated by a second-order Crank-Nicolson based projection method. Error minimization in GFD is carried out by singular value decomposition (SVD).

For external flow with moving body, a sufficiently large computational domain is required to avoid boundary effect. Also, the fine mesh is applied to the immediate fluid neighbourhood of the body to resolve high shear gradient at boundaries. Thus, millions of computational nodes may be needed to discretize the whole domain and the required memory may be up to several gigabytes to store the variables on the computational nodes. The resultant linear system for the discretized equations is usually huge and the solution procedure is thus very slow. Moreover, to ensure a stable solution and fully capture the flow behaviour, the time step size is often small. Thus, it may take several weeks or even months for the simulations to finish running.

5. OpenMP vs MPI

The numerical simulation of moving boundary flow problems still poses a significant challenge to current computers in terms of computing memory and turnaround time requirements. Solving such problems using a parallel system of processors (i.e. parallel computing) to reduce the turnaround time is now becoming a norm among researchers addressing large-scale problems. The message passing interface (MPI) technique, which utilizes a distributed system of processors with local memory, is currently the most popular parallelization approach. Typically, in this approach, a large problem is divided into a set of computing blocks. The blocks may represent a partition of the physical domain, or some other partitions of the problem. Each block is served by a processor and its local memory. When blocks represent space partition, a common interface may be needed to communicate data between adjacent blocks.

However, for 3D moving boundary problems, the moving objects may travel from one computational block to another in the course of its motion. Such situations make domain decomposition and data communication at the interfaces very difficult. Thus, we adopted a more recently-developed parallel computing technique that is based on shared memory processors (SMP), i.e. OpenMP, in our codes to avoid the explicit domain decomposition and data communication involved for MPI-based programming.

OpenMP is an implementation of multithreading based on a directive-based approach [7]. It utilizes a form of fork-and-join parallelism: the master thread forks a specified number of slave threads at those points where parallel operations are required. The created threads run concurrently, with the runtime environment allocating threads to different processors. At the end of the parallel operation, the slave threads join back to the master thread. The primary benefit of using OpenMP is the relative ease of code parallelization. As OpenMP consists of a set of compiler directives, it allows the same base code to be used for development purpose on both single-processor and multiprocessor platforms and supports incremental parallelization.

6. Parallelization Strategy

6.1 Parallel *for* loops

If no dependence between one iteration of the loop and another, we can simple indicate the compiler that the iteration of a for loop may be executed in parallel:

```
#pragma omp parallel for
for(ijk=0;ijk<Point_Meshless;ijk++)
    U_Ale[ijk] = (X_Meshless[ijk]-X_Meshless_Old[ijk])/Dt;
```

For the doubly nested loops in which either loop could be executed in parallel, it is better to parallelize the *outer loop* so that the program only incurs the fork/join overhead once:

```
#pragma omp parallel for private(i)
for(j=0;j<IPoint;j++)
    for(i=0;i<JPoint;i++)
        A[i][j]= B[i][j]+ C[i][j];
```

The reduction clause is used to sore partial sums in private variables and then adding the partial sums to the shared variable after loop:

```
#pragma omp parallel for private(tmp2) reduction(+: tmp3)
    for(i=0;i<Point_Meshless;i++)
    {
        tmp2=FI[i];
        FI[i]=(U[i]-So[i]);
        tmp3+=(tmp2-FI[i])*(tmp2-FI[i]);
    }
```

A critical section can be denoted by putting the program "`#pragma omp critical`" in front of assignment statement (`tmp3+=`). Now only one thread at a time may execute code in the critical section and the assignment statement that reads and updates `tmp3` can be executed correctly.

For a bulky loop, the statements are encapsulated as a *function* to minimize scoping required and reduce error-prone:

```
#pragma omp parallel for shared (nb)
for(it=0;it<Point_Meshless;it++) Node_Search_A(it, nb);
```

6.2 Parallel *task*

An important new feature of OpenMP is that it allows parallelizing a linked list of task. For example, a linked *class* is parallelized as:

```
#pragma omp parallel
{
    #pragma omp single
    {
        Current_Meshless_Member=C2M_Object->C2M_Head_Ptr;
        while(Current_Meshless_Member != NULL)
        #pragma omp task
            Update_Velocity_B(Current_Meshless_Member);
        Current_Meshless_Member = Current_Meshless_Member ->Next_Member;
     }
 }
```

Alternatively, it can also be parallelized as follows:

```
Tmp_Member=C2M_Object->C2M_Head_Ptr;
#pragma omp parallel private (Current_Meshless_Member)
{
    #pragma omp critical
      {
          if(Tmp_Member==NULL)
              Current_Meshless_Member = NULL;
          else
             {
                 Current_Meshless_Member = Tmp_Member;
                 Tmp_Member = Tmp_Member->Next_Member;
             }
      }
      while(Current_Meshless_Member != NULL)
      {
            Update_Velocity_B(Current_Meshless_Member);
            #pragma omp critical
            {
                  if(Tmp_Member==NULL)
                      Current_Meshless_Member = NULL;
                  else
                  {
                        Current_Meshless_Member = Tmp_Member;
                        Tmp_Member = Tmp_Member->Next_Member;
```

```
                    }
                }
            }
        }
```

6.3 Functional parallelism

We can assign different threads to different portion of code:

```
#pragma omp sections
{
    #pragma omp section
    <code block1>
    #pragma omp section
    <code block2>
    #pragma omp section
    :
}
```

6.4 Use of OpenMP runtime library routines

In OpenMP every thread on a multiprocessor has a unique identification number. We can retrieve this number using the function *omp_get_thread_num*. The number of active threads can be obtained by function *omp_get_num _threads*. Thus, the *for* loop can be parallelized as:

```
#pragma omp parallel private(tid)
{
    tid=omp_get_thread_num();
     t= omp_get_num _threads;
    for(i=tid;i<Point_Meshless;i=i+t)
      <code block>
}
```

6.5 Summary

In general, in order to get good performance of a parallel code, it is necessary to parallelize a sufficiently large portion of the code, that is, high coverage. Besides this, we can increase the *grain size*, which is the number of computations performed between communication or synchronization steps, to improve the performance of a parallel program. Moreover, we can also use the *schedule* clause to specify how the iteration of a loop should be allocated to threads. Usually, *static schedules* have low overhead but may exhibit high load imbalance while *dynamic schedules* have higher overhead but can reduce load imbalance.

7. Other issues related to performance

In addition to the costs of invoking the parallel loop and executing the barrier, cache and synchronization effects can greatly increase the cost. The cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations. The cache size, structure, technique may affect the performance of the parallelization. Besides these, the locality of code may also have some effect. Arrays in C/C++ are stored in row-major order, means that the rows of the array are laid out one after the other in memory. Thus, for a two dimensional array A[m][n], the address of A[i][j] is just before the address of A[i][j+1] but is *n* elements away from the address of A[i+1][j]. Thus, it is better to arrange the loops as:

```
for(i=0;i<m;i++)
    for(j=0;j<n;j++) A[i][j]=...
```

Successive references in time are adjacent in memory. There is no opportunity for the cache line to be evicted, and we are able to fully exploit locality. Many compilers, like Intel® icc, automatically interchange loops for the user, but sometimes more complicated and global transformation are needed to improve locality.

Barriers are used as a global point of synchronization. A *nowait* clause is used to minimize synchronization when the barrier at the end of *parallel for* or *parallel regions* is not necessary:

```
#pragma omp parallel for private (j) nowait
for(i=1;i<IPoint-1;i++)
   for(j=1;j<JPoint-1;j++)
        U[i][j]=(dx2*U[i][j+1]-dx1*U[i][j+2])/(dx2-dx1);
#pragma omp parallel for private (k) nowait
for(i=1;i<IPoint-1;i++)
   for(k=1;k<KPoint-1;k++)
        W[i][k]=(dx2*W[i][k+1]-dx1*W[i][k+2])/(dx2-dx1);
```

One can also enhance the performance of the code by applying the variety of optimizations used by the compiler. Intel® C++ Compiler 11.0 is used to compile the present code. The opinion -ipo is applied to enable Interprocedural Optimizations (IPO). IPO benefits the code performance primarily because it enables inlining. The preliminary test indicated that by using -ipo opinion, the time for the simulations to finish running is reduced around 6%. It is worth noting that -O3 opinion has a negligible effect on the code performance.

8. Speedup & Efficiency

In this section, the performance and efficiency of the present OpenMP code are examined. The test case concerns the simulation of flow induced by a sphere rotating at a constant angular velocity about its z-axis in an unbounded incompressible viscous fluid domain. The background computational domain was discretized by 22,743,361 Cartesian nodes while the surface of the sphere and its vicinity were discretized by 37,505 meshfree nodes. The present code is written in C/C++ language and compiled with Intel® C++ Compiler 11.0. The simulations were performed on three computational nodes provided by Computer Centre, National University of Singapore: atlas3-c63 (installed with two Dual-Core Intel Xeon 5160 @ 3.00 GHz processors, total 4 CPU-cores), atlas4-c80 (installed with two Quad-Core Intel Xeon E5430 @ 2.66GHz processors, total 8 CPU-cores) and atlas4-c85 (installed with four Quad-Core Intel Xeon E7330 @ 2.40 GHz processors, total 16 CPU-cores). The memory usage of the present case is around 6 gigabytes.

Speedup factor (S_P) and efficiency (E_P) of the present OpenMP code are defined as $S_p = T_1/T_p$ and $E_p = T_1/(PT_p)$, respectively, where P is the number of CPU-cores, T_1 is the wall clock time of the sequential algorithm, and T_P is the wall clock time of the parallel algorithm with P CPU-cores. Table 1 summarizes the CPU times and wall clock times for running 50 time steps of the simulation by using different number of CPU-cores for all nodes. Figure 1 shows the parallel speedup factor and efficiency using different number of CPU-cores for all nodes.

Table 1. Comparison of CPU time and wall clock time for different number of CPU-cores

atlas3-c63	CPU-cores	1	2	3	4	/
	CPU Time (s)	12469	12772	13718	14313	/
	Wall Clock Time (s)	12479	6415	4609	3633	/
atlas4-c80	CPU-cores	1	2	4	6	8
	CPU Time (s)	9882	10293	11418	13163	14835
	Wall Clock Time (s)	9894	5183	2900	2233	1890
atlas4-c85	CPU-cores	1	4	8	12	16
	CPU Time (s)	15842	16601	18775	22545	25508
	Wall Clock Time (s)	15855	4209	2387	1930	1619

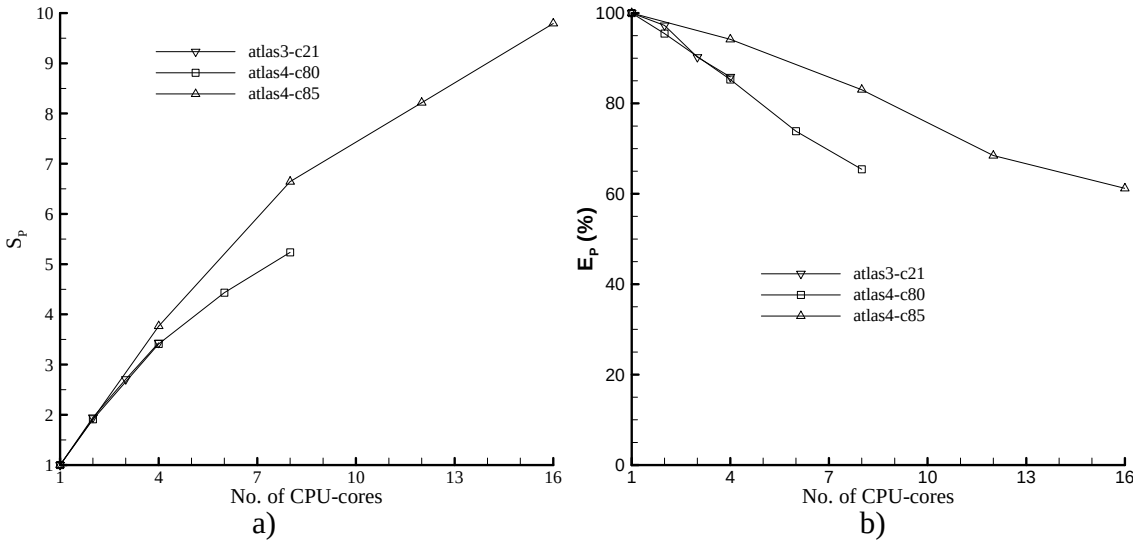
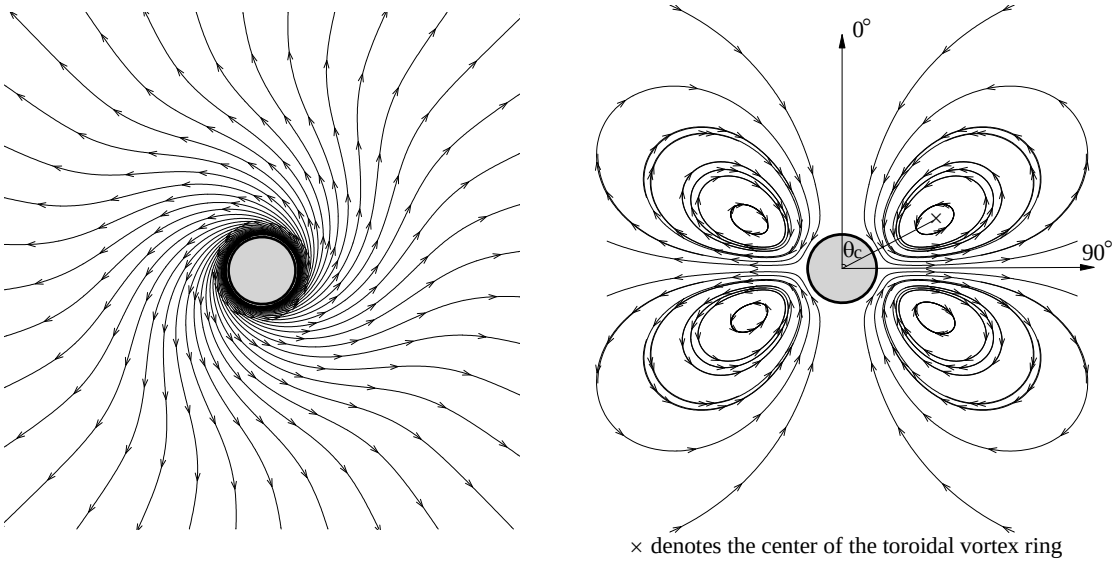


Figure 1 a) Speedup factor and b) efficiency of the present OpenMP parallelization.

Table 1 indicates that, with an increase in the number of CPU-cores, the computational turnaround time is significantly reduced although the total CPU time is increased. The increase in total CPU time is due the OpenMP overhead. Table 1 also shows that when the same number of CPU cores is used, the computational time on atlas4-c80 is the shortest while that on atlas4-c85 is longest. Figure 1a demonstrates that the speedup factor rises more slowly as the number of CPU-cores P is increased; S_P = 3.4 for P = 4 on atlas4-c63, S_P = 3.4 for P = 4 and 5.2 for P = 8 on atlas4-c80, S_P = 3.8 for P = 4, 6.6 for P = 8, and 9.8 for P = 16 on atlas4-c85. Figure 1b depicts parallel efficiency with P; which is 85.8% for P = 4 on atlas4-c63, 85.3% for P = 4 and 65.4% for P = 8 on atlas4-c80 while is 94.2% for P = 4, 83% for P = 8 and 61.2% for P = 16 on atlas4-c85. Figure 1b also shows that the parallel efficiency decreases with an increase in P. The results indicate that although the simulations on the atlas4-c63 and atlas4-c80 run faster than those on atlas4-c85, the simulations on atlas4-c85 can achieve better speedup and efficiency, which may be due to other factor, such as a nodes' bus, cache etc. This suggests that if the number of CPU cores keeps increasing, the simulations on atlas4-c85 eventually run fastest.

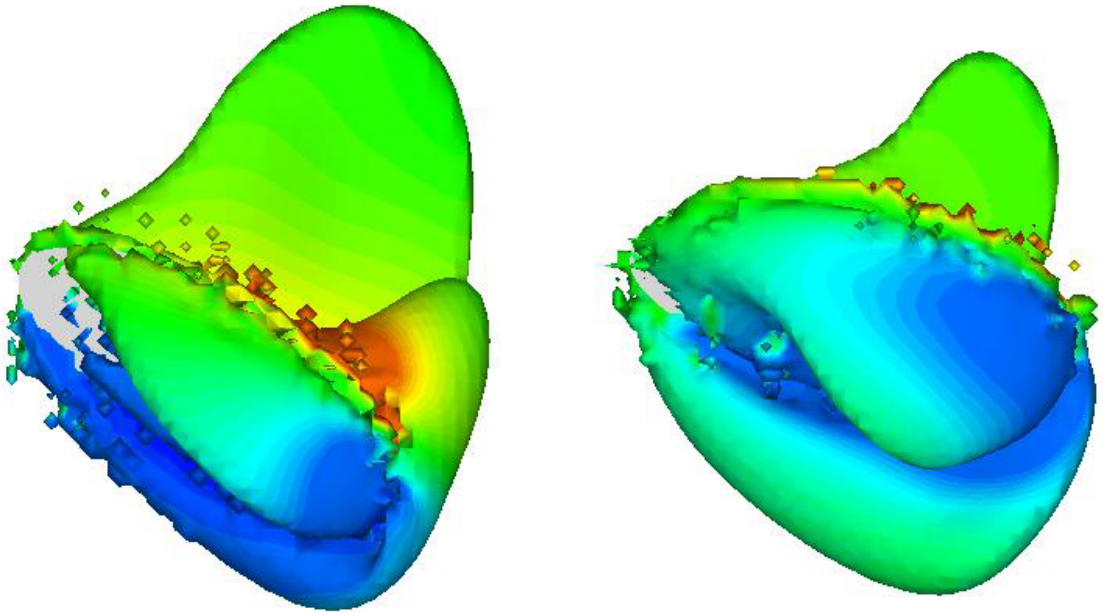
9. Numerical Examples

In this section, two flow visualizations obtained by using the present code are presented. The first case is the flow induced by a rotating sphere in an infinite domain. The memory usage of the present simulation is around 6 gigabytes as mentioned above. It takes around 46 hours for the simulation to finish on an altas4 node (8 CPU-cores). The streamtraces of the flow field in the *x*-*y* and *y*-*z* central planes at Re = 20 is shown in Figure 2.



a) b)

Figure 2 Streamtraces of steady-state flow fields around a rotating sphere in an unbounded domain at Re = 20: (a) xy central plane, (b) yz central plane.



a) b)

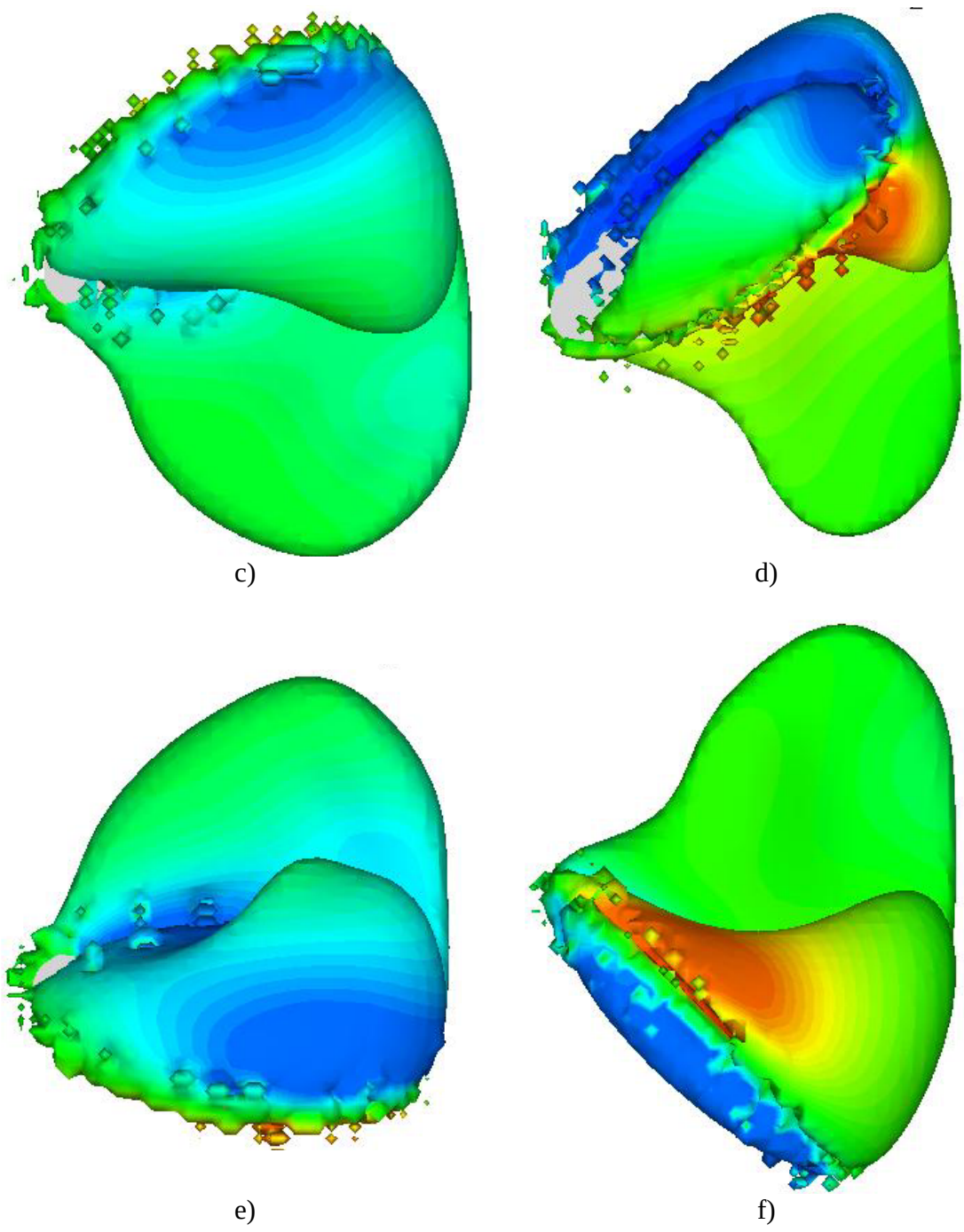


Figure 3 Vortices fields of one complete flapping cycle in *xy* central plane

The second case is the flow induced by an elliptic flat wing which simulates an ideal insect wing. The memory usage for this case is around 2 gigabytes. It takes around 38 hours for the simulation to finish on an altas4 node (8 CPU-cores). Figure 3 present the vortices fields of one complete flapping cycle at Re = 80.

10.Conclusion

The present project aims to develop a code to simulate 3D moving boundary problems, which combines the advantages of the existing approaches. To achieve this goal, a SVD-GFD based hybrid meshfree-Cartesian grid scheme is applied here. The present code has been successfully implemented to simulate several moving boundary problems, which demonstrate the effectiveness and potential of the present hybrid meshfree-Cartesian grid scheme for solving complex moving boundary problems in 3D.

The present code is speeded up by utilizing OpenMP technique. More than 95% portion of the code has been parallelized. By systemically considering coverage, granularity, load balancing, locality and synchronization effects, the present code achieves speedup of around 10 for a computational node with 16 CPU-cores, which a corresponding parallel efficiency of 61.2%. By using *-ipo* opinion provides by Intel® C++ Compiler 11.0, the computational time is further reduced around 6%. These results indicated that the simulations which take two months to finish on a CPU-core now can complete within one week on 16 CPU-cores. It is worth noting that the system configuration also has a significant effect on parallel speedup and efficiency. When using 8 CPU-cores, the efficiency of atlas4-c85 is much higher than those of atlas4-c80 and atlas4-c63.

Reference

[1]. H. Aono, F. Liang and H. Liu, Near- and far-field aerodynamics in insect hovering flight: an integrated computational study. *Journal of Experimental Biology* 2008; 211: 239-257.

[2]. S. Xu and Z. J. Wang, An immersed interface method for simulating the interaction of a fluid with moving boundaries. *Journal of Computational Physics* 2006; 216: 454-493.

[3]. Chew CS, Yeo KS, Shu C. A generalized finite-difference (GFD) ALE scheme for incompressible flows around moving solid bodies on hybrid meshfree - Cartesian grids. *Journal of Computational Physics* 2006; 218: 510-548.

[4]. Ang SJ, Yeo KS, Chew CS, Shu C. A singular-value decomposition (SVD)-based generalized finite difference (GFD) for close-interaction moving boundary flow problems. *International Journal for Numerical Methods in Engineering* 2008; 76: 1892-1929.

[5]. Wang XY, Yeo KS, Chew CS, Khoo BC. A SVD-GFD scheme for computing 3D incompressible viscous fluid flows. *Computers & Fluids* 2008; 37: 733-746.

[6]. Hirt CW, Amsden AA, Cook JL. An arbitrary Lagrangian-Eulerian computing method for all flow speeds. *Journal of Computational Physics* 1974; 14: 227-253.

[7]. Chandra R, Menon R, Dagum L, Kohr D, Maydan D, McDonald J. Parallel Programming in OpenMP. Morgan Kaufmann Publishers, San Francisco 2001.