Hybridized Parallelization of NGA

Hang Chu†, Lingnan Liu†, Jilong Wu†, Mohammed Sameed Shafi†, Sheng Wang‡
†School of Electrical and Computer Engineering, Cornell University
‡Sibley School of Mechanical and Aerospace Engineering, Cornell University
hc772,ll656,jw859,ms2788,sw767@cornell.edu

Abstract

In this report, OpenMP is applied in one of the state of art multiphase fluid mechanics solver, NGA. It is reported to have at least 15% decreasing in the running time in Von Karman vortex street test case.

1 Introduction

NGA is a high-order, fully conservative CFD code, which is tailed for turbulent flow computation. It enables the prediction of turbulent multiphase reacting flows from first principles using large-scale computing resources. NGA consists of a range of multi-physics modules integrated around a variable density, low Mach number Navier-Stokes solver. NGA has been used in numerous DNS and LES studies including liquid atomization [1,2], electrohydrodynamics [3], spray dynamics, spray combustion[4], biomass gasification [5], premixed, partially-premixed, and non-premixed turbulent jets [6,7], and combustion in technical devices, such as large-scale furnaces [8], internal combustion engines, and aircraft engine afterburners.

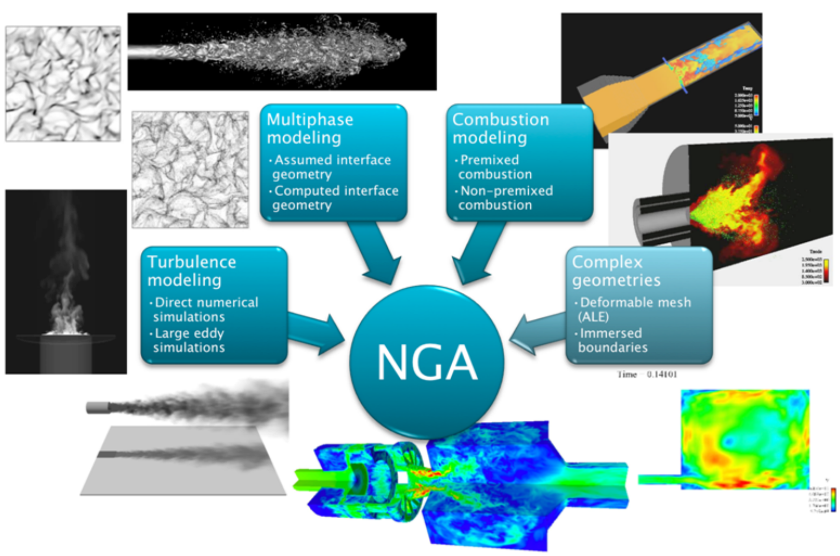

Figure 1: Applications of the NGA code.

NGA has been developed using MPI at the birth of this code. In order to make the code easier to be distributed, very few developers have tried to optimize the code. One possible way to have significant speed up the code without big effort and compensate of clearness is to use OpenMP, which enables a shared memory-multiprocessing process. In this paper, OpenMP is applied to some of the most time-consuming functions in NGA. At least 15% reduction in running time when simulating the Von Karman Vortex street problem.

In this report, the idea of hybrid parallelization in simple test case, 1D wave problem, is tested in Section 2. Von Karman Vortex street and profiling of the code is described in Section 3. Details of application of OpenMP on NGA and the performance improvement are reported in Section 4. Future work following this project is illustrated in Section 5.

2 Simple test of hybrid MPI-OpenMP

There are many reasons for one to use hybrid MPI-OpenMP parallelization. MPI deals well in passing messages, but suffers from significant overheads in a memory sharing framework. On the other hand, OpenMP has a shared memory architecture, but has problems in dealing with multiple nodes. In many applications, multiple nodes are often needed to handle the heavy computation, in the same time variables might need to be shared within a node. Hybridized MPI-OpenMP parallelization provides a good solution. In the hybridized framework, MPI is used for inter-node communication, OpenMP is used for intra-node variable sharing.

As the first part of our project, we begin with a simple testing experiment to get familiar with mixed MPI-OpenMP programming. We made changes to the *Wave1d* implementation from previous homework: in the function *sim_advance* we added useless but time consuming operations to simulate a process where more computation is needed to update a single cell.

We first compared two cases: two thread pure MPI, two node MPI each has two thread OpenMP, and two node MPI each has four thread OpenMP. The time is shown in Table 1.

Table 1: First comparison.

# of cells	30000	40000	50000	60000
MPI 2	4.12s	5.43s	6.76s	8.34s
MPI-OpenMP 2x2	4.13s	5.50s	6.81s	8.08s
MPI-OpenMP 4x2	2.05s	2.72s	3.40s	4.06s

From Table 1 it can be seen that: 1.2x2 hybrid code runs as fast as pure MPI. In the hybrid code, inter-node MPI communication is more expensive, also cell updating is faster as more threads do the job. The two effects cancel each other. 2. 4x2 hybrid code runs faster than both 2x2 hybrid code and pure MPI. If the number of cells becomes significantly larger, or the computation for updating each cell increases significantly, and assume that we have sufficient computation power at hand, then MPI-OpenMP will become a better choice.

We also conducted another comparison: pure MPI on two nodes each has eight threads, MPI on two nodes and each has eight threads running OpenMP. The time is shown in Table 2.

Table 2: Second comparison.

# of cells	10000	100000	200000
MPI 8x2	0.47s	3.48s	6.98s
MPI-OpenMP 8x2	0.46s	3.45s	6.82s

From Table 1 it can be seen that: if using the same number of threads and single cell updating doesnt involve heavy variable sharing, then pure MPI has almost the same scalability as a hybridized framework. In the original NGA code, only MPI is used as the situation for the major computation process is consistent with this experiment.

We would like to thank our dear instructor Prof. David Bindel for providing the *mpihsub* script.

3 Von Karman vortex street/code profiling

Von Karman vortex street is a repeating pattern of swirling vortices, which is caused by unsteady separation of flow around blunt bodies. Figure 2 shows a Von Karman vortex street caused by air flow over the island of Jan Mayen in the Greenland Sea. With the improvement of computational method, it is possible to simulate the Von Karman vortex street using everyday laptop. The simulation of Von Karman vortex street is shown in Figure 3 with Reynolds number equals to 100.

One handy timing tool in NGA is the *mpi_wtime* function. The running time is divided to four major parts, i.e., IB(immersed boundary), combustion, velocity, and pressure calculation. The time cost in percentage of IB(immersed boundary), combustion, velocity, and pressure calculation is plotted in Figure 5. Pressure calculation consumes about 70 percent of the whole running time, which means it is the most expensive part of the simulation. As a result, the start point is to find the time consuming functions in the pressure calculation. In all the profiling tools on C4, Vtune is chosen to profile the serial code. We choose Vtune as it has been used in the previous projects. It turns out that only the serial code is profiled. The result for the simple profile is shown in Figure 4. The profiling result simply shows the time spent in each function, rather than the cumulative time in each function. After digging in the code, we found a more detailed profiling result shown in Table 3.

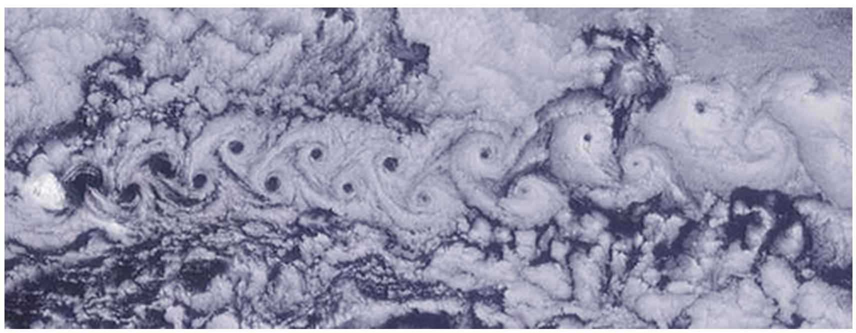

Figure 2: A Von Karman vortex street caused by the airflow over the island of Jan Mayen in the Greenland Sea [9].

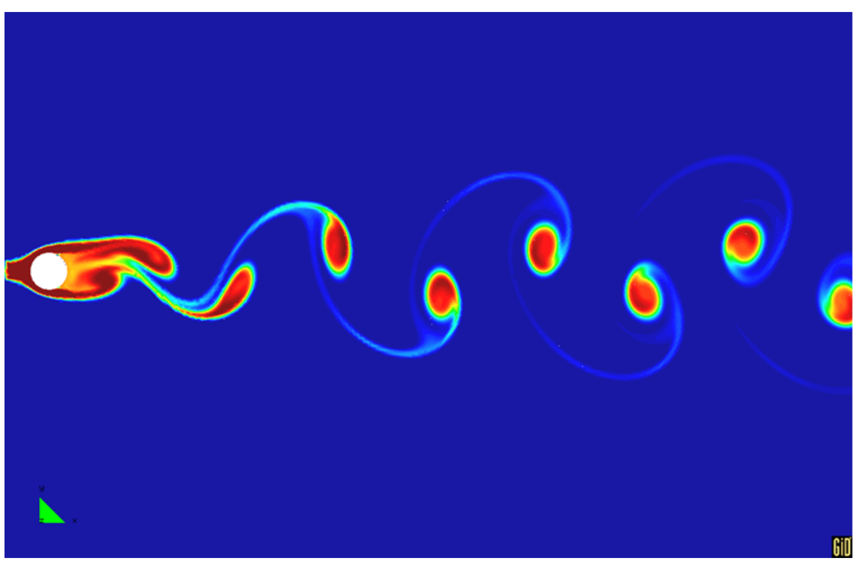

Figure 3: Simulation of Von Karman vortex street, $R_E = 100$.

The profile result turns out that the preconditioning part is most time consuming part of the code.

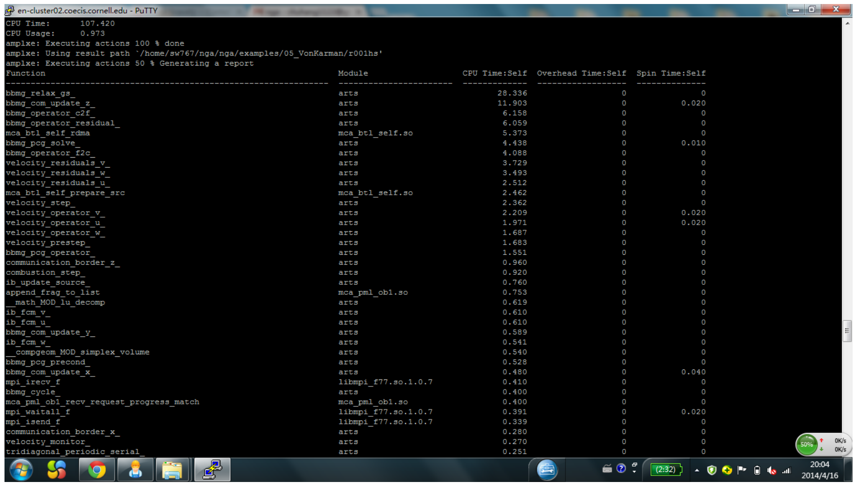

Figure 4: Profiling result of the serial code using Vtune.

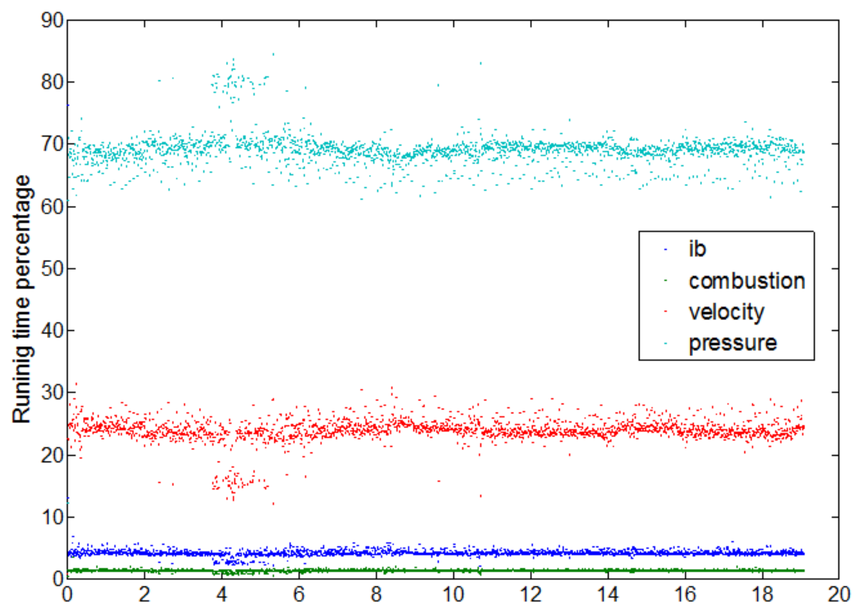

Figure 5: Time cost in percentage for IB, combustion, velocity and pressure calculation.

Table 3: A improved profiling result.

function	time
bbmg_pcg_solve	63.22s
bbmg_relax_gs	33.81s
bbmg_com_update_z	10.80s
bbmg_operator_residual	7.93s
bbmg_operator2f	6.36s
velocity_residual_v	4.47s
velocity_residual_w	4.47s

4 Hybridized parallelization

Based on the analysis in the previous section, *bbmg_operator_residual* and *velocity_residuals_w*, *velocity_residuals_v*, *velocity_residuals_u* are chosen to use OpenMP.

Some fragments of the code of applying OpenMP in *bbmg_operator_residual* is shown below.

```
!$OMP PARALLEL SHARED(lvl,n,k) PRIVATE(i,j)
! Compute residual
!$OMP DO SCHEDULE(DYNAMIC,20)
do k=lvl(n)%kmin_,lvl(n)%kmax_
  do j=lvl(n)%jmin_,lvl(n)%jmax_
    do i=lvl(n)%imin_,lvl(n)%imax_
      lvl(n)%r(i,j,k) = lvl(n)%f(i,j,k) &
        -sum(lvl(n)%lap(i,j,k,:,:,:)*lvl(n)%v(i-1:i+1,j-1:j+1,k-1:k+1))
    end do
  end do
end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Code of applying OpenMP in *velocity_residuals_w* is shown below. The same code can be used in *velocity_residuals_u* and *velocity_residuals_v* with very few modifications.

```
!$OMP PARALLEL SHARED(rhoUi,rhoVi,rhoWi,interp_Jw_zm,rhoU,rhoV,rhoW,W,interp_cyl_v_ym,
interp_cyl_w_zm,Fcyl,interp_Ju_z,interp_Jv_z,stc1,stc2) PRIVATE(ii,jj,kk,i,j,k)
!$OMP DO SCHEDULE(DYNAMIC,32)
! Convective part
do kk=kmin_-stc1,kmax_+stc2
  do jj=jmin_-stc1,jmax_+stc2
    do ii=imin_-stc1,imax_+stc2
      i = ii-1; j = jj-1; k = kk-1;
      rhoWi(i,j,k) = sum(interp_Jw_zm(i,j,:)*rhoW(i,j,k-stc1:k+stc2))
      Fcyl(i,j,k) = &
        - sum(interp_cyl_v_ym(i,j,:)*rhoV(i,j-stc1:j+stc2,k)) * &
        sum(interp_cyl_w_zm(i,j,:)*W(i,j,k-stc1:k+stc2))
      i = ii; j = jj; k = kk;
      rhoUi(i,j,k) = sum(interp_Ju_z(i,j,:)*rhoU(i,j,k-stc2:k+stc1))
      rhoVi(i,j,k) = sum(interp_Jv_z(i,j,:)*rhoV(i,j,k-stc2:k+stc1))
    end do
  end do
end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL

!$OMP PARALLEL SHARED(FX,FY,FZ,VISC,grad_w_z,grad_u_z,grad_w_x,grad_v_z,grad_w_y,
U,V,W,divv_u,divv_v,divv_w,ymi,interpv_cyl_v_ym,interpv_cyl_w_y,interp_sc_xz,
interp_sc_yz,stv1,stv2,st1,st2) PRIVATE(ii,jj,kk,i,j,k)
!$OMP DO SCHEDULE(DYNAMIC,32)
! Viscous part
do kk=kmin_-stv1,kmax_+stv2
  do jj=jmin_-stv1,jmax_+stv2
    do ii=imin_-stv1,imax_+stv2
      i = ii-1; j = jj-1; k = kk-1;
      FZ(i,j,k) = &
        + 2.0_WP*VISC(i,j,k)*( &
        + sum(grad_w_z(i,j,k,:)*W(i,j,k-stv1:k+stv2)) &
        - 1.0_WP/3.0_WP*( sum(divv_u(i,j,k,:)*U(i-stv1:i+stv2,j,k)) &
        + sum(divv_v(i,j,k,:)*V(i,j-stv1:j+stv2,k)) &
        + sum(divv_w(i,j,k,:)*W(i,j,k-stv1:k+stv2))) &
        + ymi(j)*sum(interpv_cyl_v_ym(i,j,:)*V(i,j-stv1:j+stv2,k)))
      i = ii; j = jj; k = kk;
      FX(i,j,k) = &
        + sum(interp_sc_xz(i,j,:,:)*VISC(i-st2:i+st1,j,k-st2:k+st1)) * &
        ( sum(grad_u_z(i,j,k,:)*U(i,j,k-stv2:k+stv1)) &
```

```
      + sum(grad_w_x(i,j,k,:)*W(i-stv2:i+stv1,j,k)) )
    FY(i,j,k) = &
      + sum(interp_sc_yz(i,j,:,:)*VISC(i,j-st2:j+st1,k-st2:k+st1)) * &
      ( sum(grad_v_z(i,j,k,:)*V(i,j,k-stv2:k+stv1)) &
      + sum(grad_w_y(i,j,k,:)*W(i,j-stv2:j+stv1,k)) &
      - yi(j)*sum(interpv_cyl_w_y(i,j,:)*W(i,j-stv2:j+stv1,k)) )
   end do
  end do
end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL

!$OMP PARALLEL SHARED(interp_sc_z,RHOmid,grad_Pz,P,divv_zx,divv_zy,divv_zz,
FX,FY,FZ,interp_zz,interp_zy,interp_zx,divc_zz,divc_zx,divc_zy,rhoWi,rhoUi,rhoVi,W,ymi,
interp_cyl_F_z,interpv_cyl_F_ym,Fcyl,ResW,Wold,rhoWold,dt_uvw,srcWmid,st1,st2,
stc1,stc2,stv1,stv2) PRIVATE(i,j,k,RHOi,rhs,st)
!$OMP DO SCHEDULE(DYNAMIC,32)
! Residual
do k=kmin_,kmax_
  do j=jmin_,jmax_
   do i=imin_,imax_
    RHOi = sum(interp_sc_z(i,j,:)*RHOmid(i,j,k-st2:k+st1))
    ! Pressure + Viscous terms
    rhs =-sum(grad_Pz(i,j,k,:)*P (i,j,k-stc2:k+stc1)) &
      +sum(divv_zx(i,j,k,:)*FX(i-stv1:i+stv2,j,k)) &
      +sum(divv_zy(i,j,k,:)*FY(i,j-stv1:j+stv2,k)) &
      +sum(divv_zz(i,j,k,:)*FZ(i,j,k-stv2:k+stv1))
    ! Convective term
    do st=-stc2,stc1
      n = interp_zz(i,j,st)
      rhs = rhs - divc_zz(i,j,k,st) * rhoWi(i,j,k+st) * &
        0.5_WP*(W(i,j,k+st+n+1)+W(i,j,k+st-n))
    end do
    do st=-stc1,stc2
      n = interp_zx(i,j,st)
      rhs = rhs - divc_zx(i,j,k,st) * rhoUi(i+st,j,k) * &
        0.5_WP*(W(i+st+n-1,j,k)+W(i+st-n,j,k))
      n = interp_zy(i,j,st)
      rhs = rhs - divc_zy(i,j,k,st) * rhoVi(i,j+st,k) * &
        0.5_WP*(W(i,j+st+n-1,k)+W(i,j+st-n,k))
    end do
    ! Cylindrical term - Convective
    rhs = rhs + ymi(j)*sum(interp_cyl_F_z(i,j,:)*Fcyl(i,j,k-stc2:k+stc1))
    ! Cylindrical term - Viscous
    rhs = rhs + ymi(j)*sum(interpv_cyl_F_ym(i,j,:)*FY(i,j-stv1:j+stv2,k))
    ! Full residual
    ResW(i,j,k) = -2.0_WP*W(i,j,k)+Wold(i,j,k) &
      + ( rhoWold(i,j,k) + dt_uvw*rhs + srcWmid(i,j,k) )  RHOi
   end do
  end do
end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Table 4 lists the running time for the first ten steps of code with OpenMP, OpenMP in *bbmg_operator_residual*, OpenMp in both *bbmg_operator_residual* and velocity residual. The test is operated on pseudo 2D grid with grid size of $512 * 512 * 6$. After the simulation had reached a region with small running time variation, which is after the first three steps, the running time is averaged. Code without OpenMP, the running time for each step is 11.23 seconds. The average running time with OpenMP in the *bbmg_operator_residual* is 8.82 seconds, which is 22 percent less than the previous timing. With more OpenMP in the code is not necessary to reduce improve the performance of the code. For example, OpenMP is applied in both *bbmg_operator_residual* and velocity residual in the last column, only 17 percent performance improvement is achieved, which is less than the 22 percent.

Table 4: Running time for the first ten steps of code with OpenMP, OpenMP in *bbmg_operator_residual*, OpenMp in both *bbmg_operator_residual* and velocity residual.

time step	no OpenMP	OpenMP in *bbmg_operator_residual*	OpenMP in bbmg and v
0	29.78s	39.79s	29.02s
1	32.65s	19.16s	23.66s
2	13.79s	9.96s	13.40s
3	11.31s	9.21s	10.66s
4	11.17s	8.87s	10.56s
5	11.40s	8.93s	10.56s
6	11.79s	8.53s	10.87s
7	11.23s	9.92s	10.48s
8	10.98s	8.32s	10.41s
9	10.95s	8.57s	10.49s
10	11.01s	8.18s	10.50s
average(3-10)	11.23s	8.82s	10.57s
improvement		22%	17%

Figure 6 shows timing results using different number of cores. From Figure 6 it can be seen the code scales well.

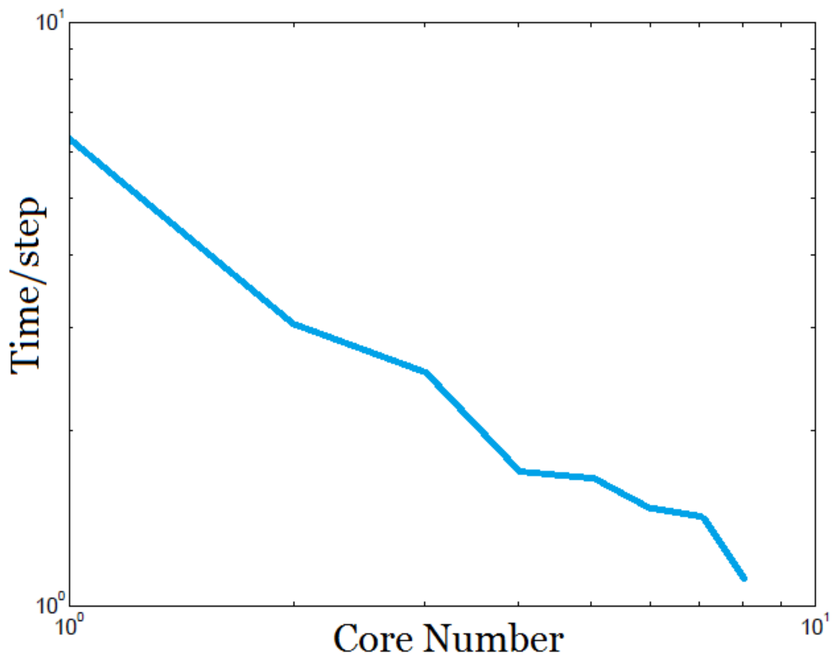

Figure 6: Average time spent per step using different numbers of cores (in seconds).

5 Future work

One future improvement is to use Jacobi method in the *bbmg_relax* function, rather than Gauss-Seidel method. Although Gauss-Seidel method needs half of the iteration steps in Jacobi method, the computation for each element can not be done in parallel. We expect to have much more improvement to use Jacobi method and OpenMP in *bbmg_relax*. We actually have our Jacobi version implemented and got correct results using it, however it is much slower than we expected (about 100 times slower than the original Gauss-Seidel method). Given the limited time we are not able to finish optimizing this Jacobi implementation. We will keep working on this as our future work.

The performance evaluation is performed on MacBook Air 2011 with 4 threads. The ultimate goal for this project is to run on supercomputers with multi nodes and each node with multi threads. Therefore, another future work would be figuring out how to use MPI among nodes and OpenMP on one node.

Von Karman vortex street is an interesting test case to start with, but it has been solved numerical for more than forty years. More state of the art and more interesting problems can be solved by NGA. OpenMP would be applied to more test cases to speed up the core in the future.

6 Reference

[1] O. Desjardins, V. Moureau, and H. Pitsch. An accurate conservative level set/ghost method for simulating primary atomization. *J. Comput. Phys.*, 227(18):83958416, 2008.

[2] O. Desjardins and H. Pitsch. A spectrally refined interface approach for simulating multiphase flows. *J. Comput. Phys.*, 228(5):1658-1677, 2009.

[3] B. P. Van Poppel, O. Desjardins, and J. W. Daily. A ghost fluid, level set methodology for simulating electrohydrodynamic atomization of liquid fuels. *J. Comp. Phys.*, 229:7977-7996, 2010.

[4] O. Desjardins and H. Pitsch. Modeling effect of spray evaporation on turbulent combustion. *10th International Congress on Liquid Atomization and Spray Systems*, Kyoto, Japan, 2006.

[5] P. Pepiot, M. W. Jarvis, M. R. Nimlos, and G. Blanquart. Chemical kinetic modeling of tar formation during biomass gasification. *In 2010 Spring Meeting*, Boulder, CO, 2010.

[6] E. Knudsen and H. Pitsch. A dynamic model for the turbulent burning velocity for large eddy simulation of premixed combustion. *Combust. Flame*, 154:740-760, 2008.

[7] E. Knudsen and H. Pitsch. A general flamelet transformation useful for distinguishing between premixed and non-premixed modes of combustion. *Combust. Flame*, 156:678-696, 2009.

[8] L. Wang and H. Pitsch. Prediction of pollutant emissions from industrial furnaces using large eddy simulation. *5th US Combustion Meeting*, San Diego, CA, 2007.

[9] http://forum.netweather.tv/blog/126/entry-1634-von-karman-vortices/