Fall 2017 ME759 Final Project Report
University of Wisconsin-Madison

Parallelizing Advection Equation using OpenMP, MPI and CUDA

Arpit Agarwal
Raunak Bardia
Chia-Wei Kuo

December 21, 2017

Abstract

In this project, we would try to parallelize the two-phase flow C++ solver made by our group in 2015. This solver could well reproduce the 2D pure-advection problem with a accurate fourth-order accuracy. However, there are some drawbacks associated with the use of the code. One is that the code is developed based on sequential calculation perspective. Therefore, the code would be of limited use when it comes a high mesh resolution calculation. Besides, the code is not optimized either, which would make the user hard to access. To solve these problems, we would first optimize the serial code by further dividing the code into several files, including a main program and header files that declare all variables and functions. We also try a different optimization choice in the `Makefile`. Moving from the serial optimization is the code parallelization. We would attempt the three popular approaches, i.e. OpenMP, MPI, and CUDA, and compare their performance against the original serial code. Based on the scaling analysis, it is found all of the three parallelization approaches could substantially speedup the code, in which CUDA has the largest improvement while MPI has the least enhancement, and OpenMP is ranked between them. The successful implementation of code parallelization could be extended to account for momentum equation solver or in a more realistic but complex 3D simulation in the future.

Contents

1. Introduction

Sharp capture of liquid-gas interface is the important topic in two-phase flow simulations. With a precise depiction of interface movement, the transient flow dynamics, such as primary breakup or phase change, could be better understood through numerical approach. In this perspective, a new two-phase flow 1000-line C++ code was developed by our group In 2015, mainly by Raunak Bardia (one of the team member). The code is capable of locating the liquid-gas interface to a accurate fourth order accuracy. The numerical algorithm is based on the Gradient Augmented Level Set (GALS) [1], which is one of the improved version of of the classical Level Set approach by solving one more equation representing the gradient of the level set function.

The code is currently aimed at pure-advection problem [2] only. This means the motion of the liquid-gas interface is advected purely based on the prescribed velocity field v(x,y,t), and the momentum equation solution is bypassed. Besides, it is applicable for 2D calculation only. The extension of this code for dealing with a complete flow solver or in 3D simulation is beyond the current scope, and this may be initiated in the future.

Under the conditions above, our code performs well in attaining high accuracy. However, some limits still exist and need to be concerned. The code we have developed is for sequential calculation only, and it is not parallelized. Besides, we did not modularize the code, which means all functions are directly written in the main code. These two points would limit the code usage when it comes to the high mesh resolution calculation such as 512x512, where 512 is the grid number in the main coordinates (x and y).

To improve the code usage, we would first work on serial-optimization. Following this is the code parallelization, which would be main focus of this final project. We will try to implement all three methods of parallelization we learned in this class, including (1) multi-threaded - OpenMP, (2) GPU - Cuda, and (3) multi-node - MPI. To simplify the problem, we would focus on the the square domain only.

The report would be organized as follows. Fistly we would briefly describe the methods to be used, including serial optimization and parallelization. Then we would state how we implement those into our code. After the implementation, several simulations would be run for doing scaling analysis, and they would be compared with the original performance to see how much improvement we could get. Finally is the conclusion and future work, where we would present the possible direction for further code modification.

2. Methods

Parallelizing the code should be based on the “good” serial code, i.e. the code is supposed to exhibit good computational performance. With a better serial code, we could get higher improvement for code parallelization. Thus, the first task is on the serial code optimization. We would modulate all functions used to make the code succinct. This means we would divide the original single c++ file into several smaller ones, including a main program and a group of header files. Inside the main program, it would only execute the functions by calling the corresponding header files, and this is the

end of serial code optimization. Then, code parallelization would be implemented. We would attempt all three methods we learned from this class, i.e. OpenMP, MPI, and CUDA. These could help us better understand the characteristics of each method. As suggested by the class, generally, CUDA is good for fine grain, while OpenMP is favored for coarse grain. MPI is otherwise beneficial for the calculations that require a vast amount of large memory. Because there is no complex procedure in our code, where we mainly use a simple `if` and `switch` clause, we expect CUDA should be more effective than OpenMP in enhancing the code performance. Regarding MPI, currently we are not familiar with how to dynamically communicate each rank for achieving two-way data sharing, which is essential for the task belonging to the CFD realm. Therefore, we anticipate the MPI may not be comparable to CUDA as well.

3. Implementation

In this section, we would illustrate how we implement the methods mentioned in Section-2.

3.1 Serial code optimization

The folder containing the original code is shown in Figure 1 (top), where `GALS_Advection.cpp` is the main program which contains around 1000 lines. We would divide `GALS_Advection.cpp` into several small ones, including a main program and several header files, as shown in Figure 1 (bottom).

```
GALS_Advection  GALS_Advection.cpp  Hermite.h  Initializer_vortex.h  Makefile
```

```
AdvectionPointCalcs.h  GALS_Advection         Makefile               VortexVelocity.h
Allocation.h           GALS_Advection.cpp     Setting.h
Constants.h            Hermite.h              TimeSteppingMethods.h
defineMPI.h            InitializeLevelSet.h   VariableDefinitions.h
```

Figure 1: (Top) the snapshot of original serial code; (bottom) the snapshot of the code after cleanup

Here we mainly modulate all functions into the header file, `AdvectionPointCalcs.h`. Inside this header file, all main functions are declared. The other improvement we made is on the optimization choice in the Makefile. We change it to O3 from O, as shown in Figure 2.

```
# Optimization and architecture
OPT   := -O
```

```
# Optimization and architecture
OPT   := -O3
```

Figure 2: (Top) the original choice of optimization in the `Makefile`; (bottom) the new choice

3.2 Code parallelization

Now we turn our focus to parallelize the serial code. OpenMP, MPI, CUDA would be respectively implemented. Further, the combination of MPI and OpenMP is also attempted.

3.2.1 OpenMP

We use 32 threads based on the `Euler` server. OpenMP is used for all functions declared in the `AdvectionPointCalcs.h`. For each parallelized function, we use `schedule(static)` and `collapse` clauses, as shown in Figure 3.

```
# pragma omp parallel
{
# pragma omp for schedule(static) collapse(2)
our code...
our code...
our code...
}
```

Figure 3: The snippet of OpenMP implementation

3.2.2 MPI

We divided the tasks for calculating the two functions in `AdvectionPointCalcs.h` into two ranks. For rank-1, the two functions are calculated for only one half the grid cells, and the calculation for the other half grid cells is executed in rank-2. After rank-2 finishes its tasks, the data is sent back to rank-1 by blocking type communication. When rank-1 receives the data, it would continue and finish the calculation by calling the other functions, as shown in Figure 4.

```
// node-1
if(ctx.rank() == 0){
     //----- MPI part ------
     function to be parallelized...
     MPI_Recv(&...);
     //----- done for MPI task ----

     // serial calculation
     serial function...
 }  // end of calculation and node-1

 // node-2
 else {
     function to be parallelized...
     MPI_Send(&...);
 } // end of node-2
```

Figure 4: The snippet of MPI implementation

3.2.3 CUDA

From the basics of fine-grain parallelism the concept of CUDA parallelization was based on assigning each grid node to a single thread of the GPU. The CUDA program required a complete overhaul of the vectorized data types used in the serial program as that type was not supported by the GPU. The variables that store the level set information for each grid point were converted to a one-dimensional pointer just like the matrix examples we did in our assignments on CUDA.
The numerical algorithm that was followed by each thread was as follows:

- Calculate the advection point from which the level set data will be updated for the next time step
- Calculate the interpolated value of level set at that advection point using the nearby points - This required access to spatially local global memory by each thread.
- Evaluate the gradients of level set using the interpolated value and doing a time integration on that

Although, it was a one dimensional pointer that stored the N x N values at the grid points, the grid was still treated in two dimensions. The grid was divided into Nb x Nb blocks, each with a tile size of Nt x Nt threads, where Nt <= 32. The corresponding node index for each thread was obtained as shown in Figure 5a, and the corresponding position in the 1D pointer was obtained as shown in Figure 5b.

```
unsigned int bx = blockIdx.x;
unsigned int by = blockIdx.y;

unsigned int tx = threadIdx.x;
unsigned int ty = threadIdx.y;

unsigned int index_x = bx * TileSize + tx;
unsigned int index_y = by * TileSize + ty;
```

Figure 5a: Thread Configuration of Grid

```
unsigned int indexToWrite = index_y * nx + index_x;
```

Figure 5b: Array Location of the Node

3.2.4 Combination of MPI and OpenMP

As a further extension, we try to combine MPI and OpenMP. The procedure is basically the same as Section 3.2.2, but with all function being parallelized at the same time using OpenMP.

4. Results and discussions

4.1 Verification

To verify the solution obtained from these codes we ran a Vortex flow test case. The system is initialized with some level set values at each grid point and the level set is subjected to a vortex flow for a time Tperiod. The flow reverses after half the time and ideally we should expect the level set to come back to its original values at the end of time period.

We calculate the maximum error between the initial and final level set values obtained from the advection algorithm. The results shown below are for the results obtained from the CUDA code. As expected, Figure 6 shows that the error values are converging as the grid size decreases and the error decreases by nearly a cubic power of the grid size, which is expected from the numerical analysis of this scheme [1]. The pictorial representation of the case for different grid sizes at the maximum deformation is shown in Figure 7.

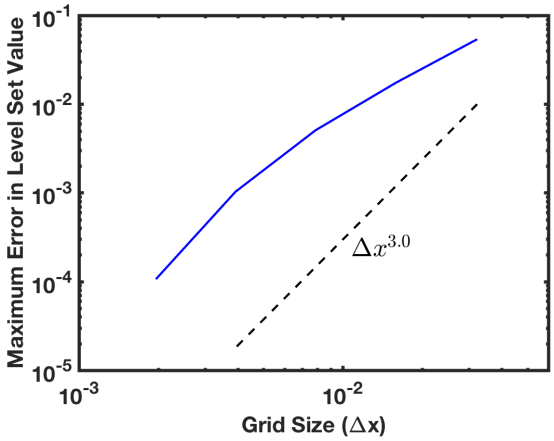


Figure 6: Level set error convergence for vortex velocity field

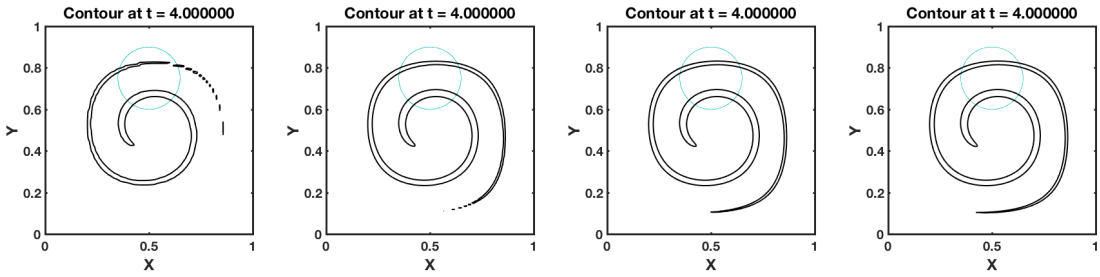


Figure 7: Grid Sizes: (a) 64x64 (b) 128x128 (c) 256x256 (d) 512x512. (a) to (d) is counted from the left to the right.

4.2 Scaling result

After verifying the parallelized code, we would run several simulations respectively using OpenMP, MPI, CUDA, and the combination of MPI and OpenMP. The computational performance is compared against the original serial code as well as the optimized serial code.

4.2.1 Overall comparison

Figure 8 shows the overall comparison, where N is the total problem size (32x32, 64x64, 128x128, and 256x256). Several points are drawn based on this result.

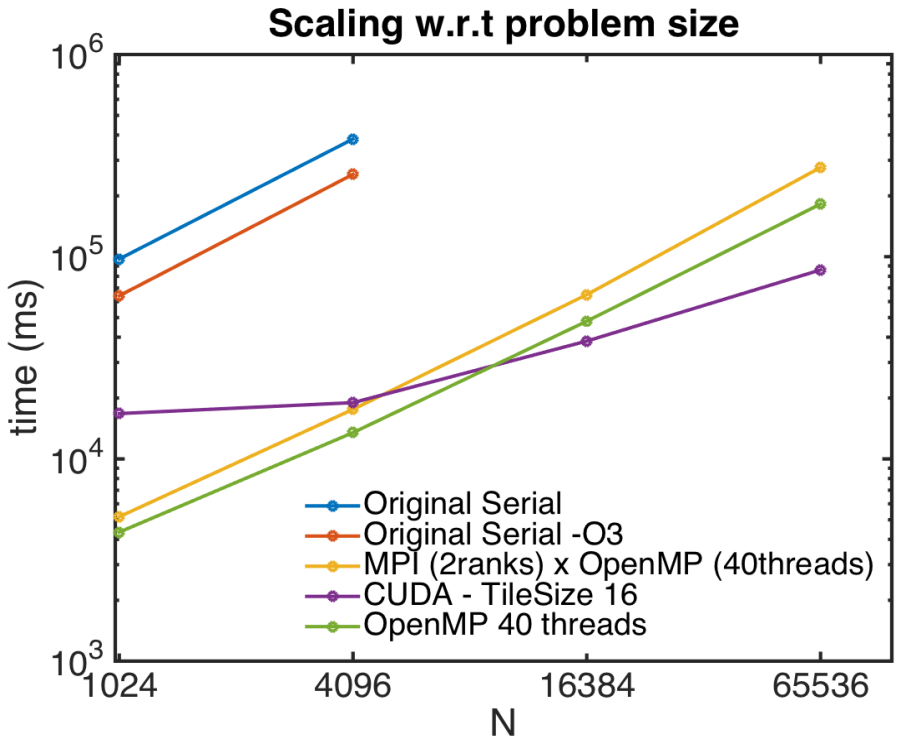


Figure 8: The performance comparison of each approaches illustrated in Section 3 against the original serial code

- Speedups:
  - OpenMP (40 threads) vs Original Serial : 22.4 times faster
  - CUDA vs OpenMP (40 threads): 2.11 times faster
  - Overall speedup (CUDA vs orig serial): 47.5 times faster
- CUDA performs poorly for smaller problem sizes - overhead of moving data is relatively high for smaller cases
- CUDA performs much faster for higher problem sizes - almost an order of magnitude faster than the fastest OpenMP implementation
- MPI with OpenMP (2 ranks x 40 threads) performs slower than a pure OpenMP implementation (40 threads). Communication overhead in MPI is probably killing the performance.
- Note that there are only two data points for the serial code - this is because the code is prohibitively slow and we could not run the larger cases with it.

4.2.2 OpenMP scaling

Figure 9 shows the OpenMP Scaling. It indicates that

- A near linear speedup is obtained when the number of threads used for the OpenMP implementation is increased from 2 to 4 and the speed up begins to saturate with further increase in the number of threads.
- The tests conducted up to 32 threads continued to show a declining trend in the runtime.

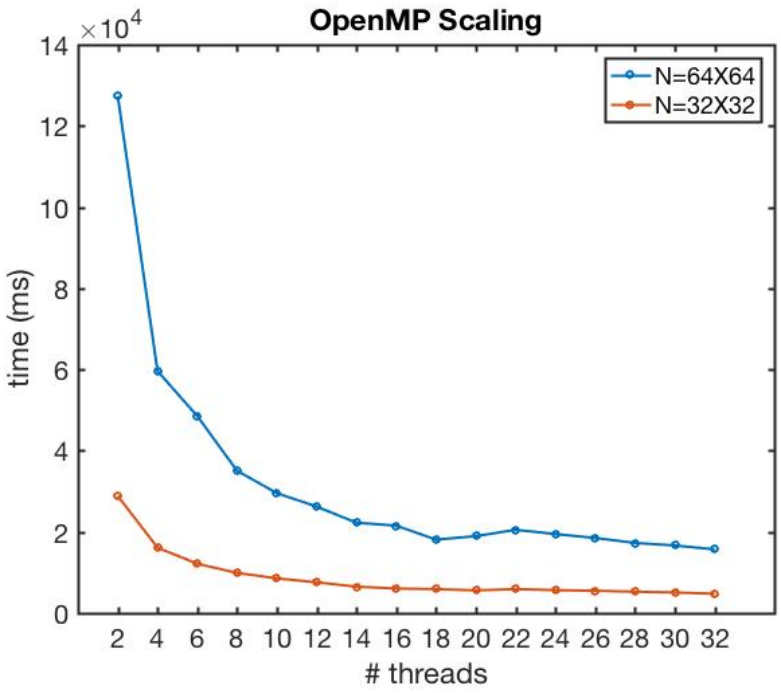


Figure 9: Scaling analysis of the OpenMP

4.2.3 CUDA performance

Figure 10 shows the effect on run time by the use for different number of threads in a single CUDA block. The tile size in the following discussion refers to the number of threads in a single direction in a block.

- Tile size 32 (1024 threads per block) performs worse than tile sizes of 16 and 8
- We suspect this is an occupancy issue
    - Each thread requires a fixed no. of registers
    - No shared memory usage, therefore no. of registers is the bottleneck
    - 1024 threads per block leads to lower occupancy and therefore poorer performance
- Tile sizes of 16 (256 threads per block) and 8 (64 threads per block) both show an improved performance for the same problem size, which indicates that we do not hit the registers limit in these cases.

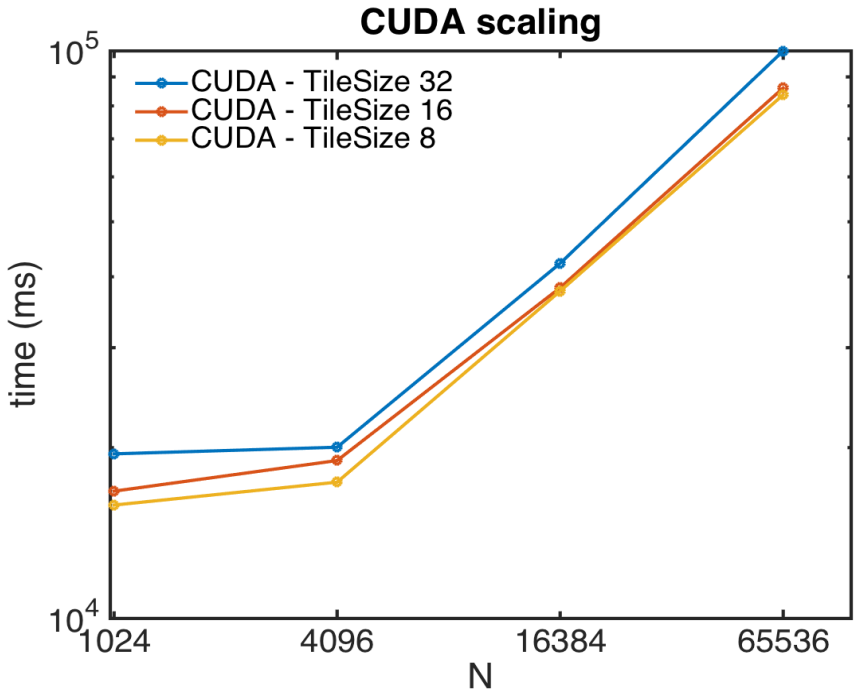


Figure 10: Scaling analysis of CUDA

4.2.4 MPI combining OpenMP scaling

Figure 11 shows the performance metric for the MPI combining OpenMP implementation with different number of OpenMP threads. A near linear speedup is obtained when the number of threads used for the OpenMP implementation is increased from 2 to 4.

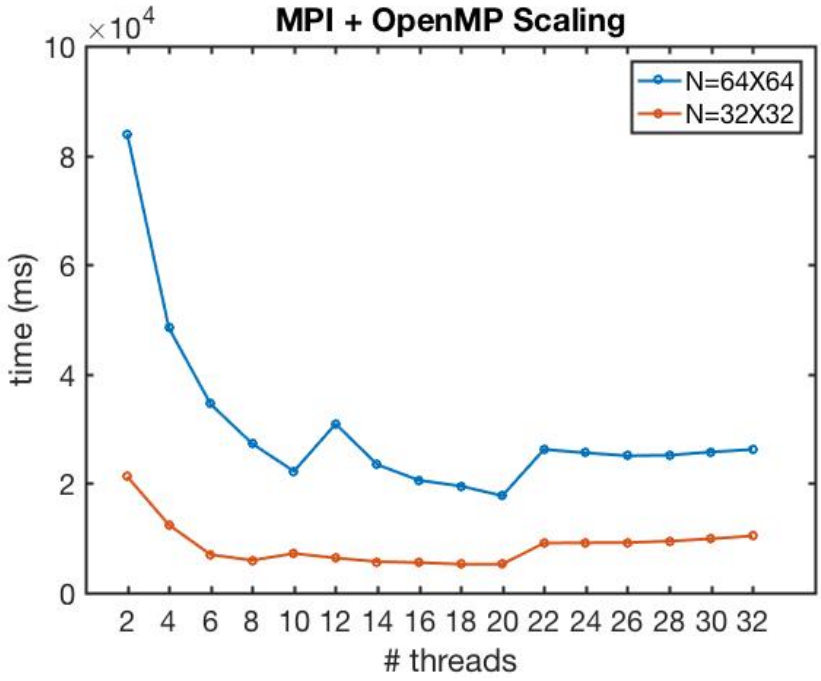


Figure 11: Scaling analysis of MPI combining OpenMP

5. Conclusions and future works

The code parallelization is working well with substantial performance improvement. Among the three parallelization approaches, CUDA exhibits the best performance, while MPI has the lowest effect. OpenMP is the one ranked between CUDA and MPI.

The CUDA implementation was very direct and each node was treated as a separate thread for processing, which is in line with the fine grain parallelism ideology of CUDA. The following areas of improvement were identified

- Use of shared memory - Each node requires its adjacent node values to create an interpolant in this numerical scheme. Currently, each node reaches out to the global memory for this information.
- Use of vectors - The current storage of data was done using 1D pointers. However, it may be beneficial to use vectors instead.

The OpenMP implementation was the most straightforward. Improvements in the OpenMP use can be obtained by:

- The current implementation parallelizes the for loop without much insight into the movement of data.
- Some OpenMP threads may require previou time data of other nodes for creating the interpolant. This may lead to some NUMA issues, which have not been investigated here.

The reason why MPI could not deliver a good enhancement is twofold.

- Our performance analysis for all codes were done on `Euler`. Only two ranks were available for a single job, which limited the use of MPI.
- More importantly, the entire code has not been parallelized using MPI. Only the first two functions are solved on two separate ranks and the data, thus generated is transferred back to the master rank for a serial processing of the following functions.
- Parallelization of the other functions required a dynamic communication between the two ranks, which we were not able to achieve even after several trials. The issue lies with the formation of the interpolant which requires the information from the other side of a processor boundary.

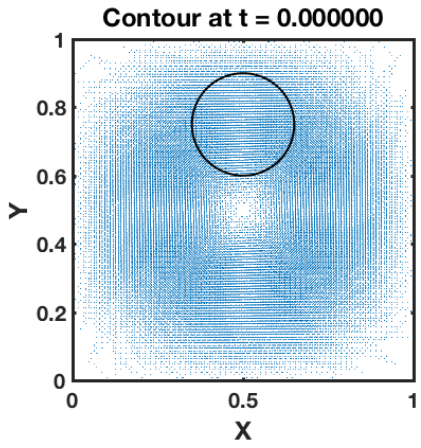
A complete and correct implementation of the MPI code is at the forefront of future work as it will give a better idea for the speed-ups obtained when compared to CUDA. As GPU's are limited by the amount of global memory, realistic 3D simulations will require the use of MPI as a performance enhancing implementation of the code.

References

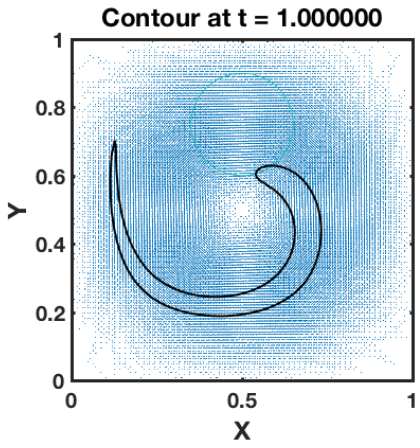
[1] J. Nave, R. Rosales, and B. Seibold, "A gradient-augmented level set method with an optimally local, coherent advection scheme." Journal of Computational Physics, Vol. 229, pp. 3802–3827, 2010.

[2] W. Rider and D. Kothe, "Reconstructing volume tracking." Journal of Computational Physics, Vol. 141, pp. 112-152, 1998.

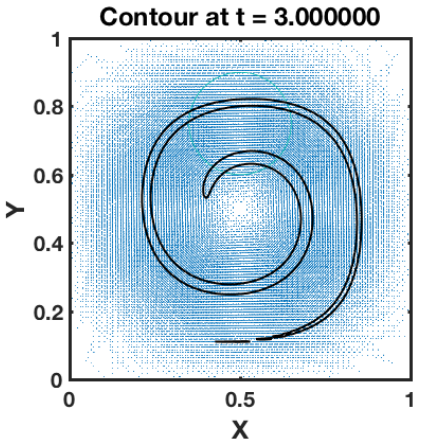
VORTEX VELOCITY ADVECTION CASE (All timing tests are done on this case)



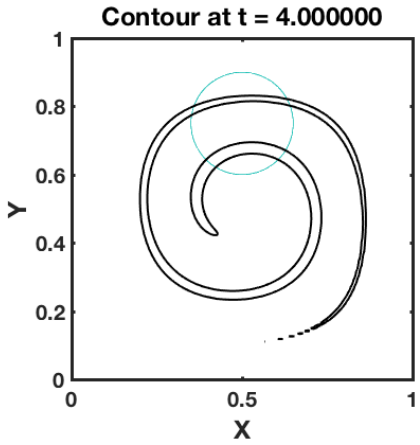
Initial Circle at t = 0



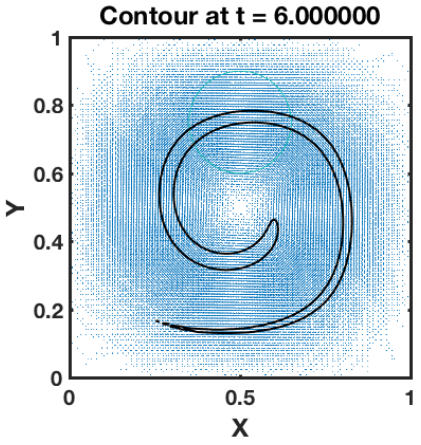
Deformation at t= 1



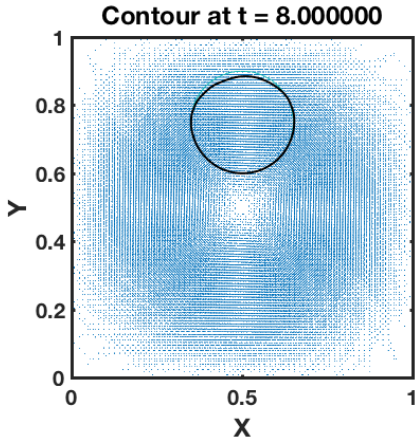
Deformation at t = 3



Maximum Deformation

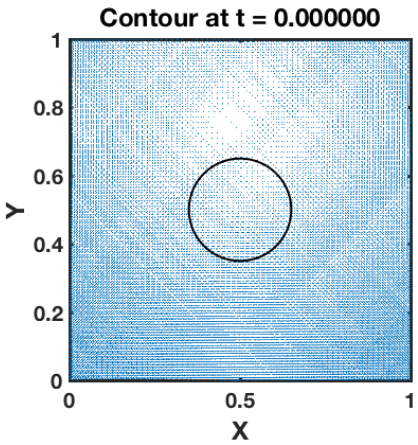


Reversed Flow. Deformation at t = 6

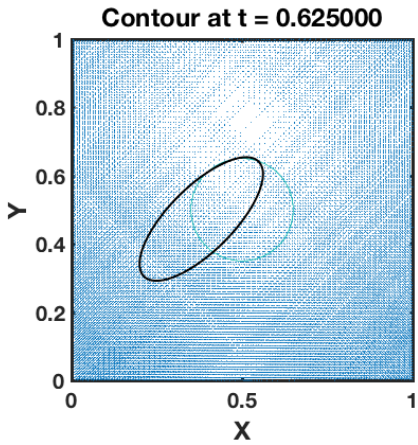


Back to original position with some deformation

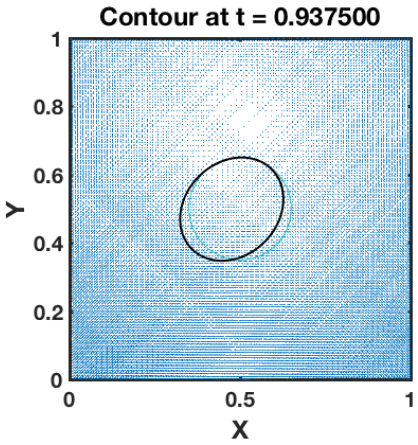
SYMMETRIC VELOCITY ADVECTION CASE



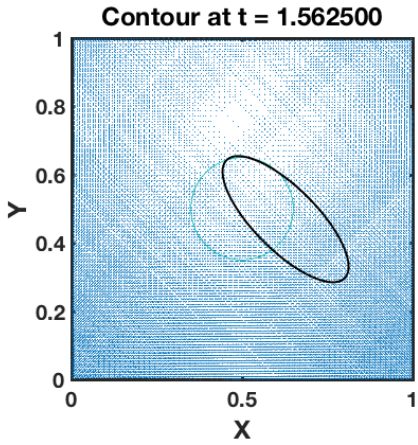
Initial Circle at t = 0



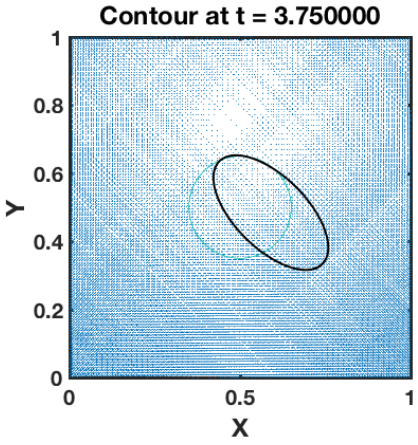
One sided deformation



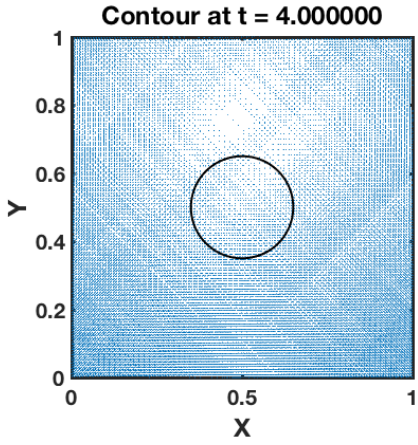
Reversing the direction of flow



Deformation in other direction



Final time period



Back to original position with some deformation