Final Project Report

Mohamad Ibrahim Cheikh

Introduction

SYSTERMS modeled by ordinary or partial differential equations often tend to be constrained by the speed and performance of processor they run on. However in the last two decades with the advancement of multiprocessing hardware and software technologies in high-performance computing (HPC), these constraints are getting smaller. Although HPC dimished the old constraints, it brought its own set of challenges, in particular challenges in software programing. Some widely used parallel programming paradigms are based on message passing, such as MPI [4], which is suitable both on distributed memory (DM) and distributed-shared memory (DSM) systems architectures. Other popular models are based the parallel compiler directives, such as OpenMP [3] and HPF [9], where the OpenMP takes advantage of the shared memory parallelism, and the HPF exploits the data parallelism.

This report hopes to implement a parallelized version of one of the most common partial differential equations, the Burgers' equation. The equation is named after Johannes Martinus Burgers, and has applications in various areas of applied mathematics, such as fluid dynamics[1], nonlinear acoustics [7], molecular gas dynamics [11], and traffic flow [6]. Although parallelization of this partial differential equation is not a new thing, our approach will focus on exploiting the intel based library, the MPI and OpenMP libraries to get the peak performance. Our aim is to surpass the performance of OpenFOAM, which is a free open source code known for solving pde's in MPI environments.

Literature Review of PDE Parallelization Techniques

A number of methods for parallelizing partial differential equations have been proposed over the years, one of the most common approaches is the **domain decomposition**. Such method is based upon the idea that the domain of the problem can be represented as a union of two or more sub-domains, where each of them is a restriction of the original problem, and takes on a particularly convenient form [8]. The advantage of reformulating a large discrete problem as a collection of smaller problem, is to enable the small discreet problems to be solved independently, and thus exploit the idea of parallelism. A typical case of domain decomposition is shown in Figure 1.

The important idea behind Figure 1 is that there are two levels of decomposition: The first level is the decomposition of the domain it self into smaller sub-domains (shaded in blue), and the second level is to establish communication between the neighboring domains (shaded in red). The efficiency of the domain decomposition method will depend on how much level 1, which is performing the computation, is taking time, in comparison to level 2 which is the communication time [5].

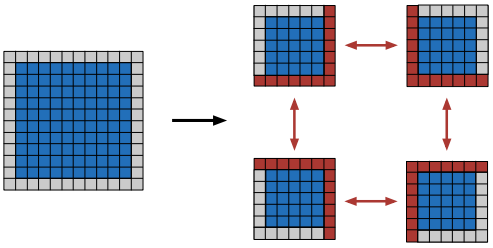


Fig. 1: Domain decomposition method

Another approach that has become a standard tool in all linear algebra codes is **vectorization** [10]. These methods achieve their speed by using an arithmetic unit that breaks a simple mathematical operation, such as a multiplication, into several subtasks, which are executed in an assembly line fashion on different operands. However the drawback of vector methods is the overhead associated with them that is called the *start up time*. Due to this overhead, vectorization is faster than regular scalar operations only when the length of the vector is sufficient to offset the cost of the start up time.

The final approach that is implemented to parallelize the performance of a set of partial differential equation solver is **coupled** scheme [2]. This approach enables the solver to divide the partial equations, and solve them separately. This approach can only be used if these equations don't depend on each other.

Burgers' Equation

Burgers' equation is used as a simplified model of turbulence described by the traditional Navier-Stokes equation. One of its advantages is that the exact solution is known, even though it's a nonlinear equation. It is often used as a benchmark for numerical methods, and applied in almost all the events involving the

movement of a flow. The two dimensional Burgers' equation is:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$
$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)$$

In the current numerical study, the equation was discretized like:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} + u_{i,j}^n\frac{u_{i,j}^n - u_{i-1,j}^n}{\Delta x} + v_{i,j}^n\frac{u_{i,j}^n - u_{i,j-1}^n}{\Delta y} = \nu\left(\frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2}\right)$$

$$\frac{v_{i,j}^{n+1} - v_{i,j}^n}{\Delta t} + u_{i,j}^n\frac{v_{i,j}^n - v_{i-1,j}^n}{\Delta x} + v_{i,j}^n\frac{v_{i,j}^n - v_{i,j-1}^n}{\Delta y} = \nu\left(\frac{v_{i+1,j}^n - 2v_{i,j}^n + v_{i-1,j}^n}{\Delta x^2} + \frac{v_{i,j+1}^n - 2v_{i,j}^n + v_{i,j-1}^n}{\Delta y^2}\right)$$

To make the numerical implementation of the above equation more clear, all the unknown components u and v which resemble the future time step are rearranged on the left side of the equation while the components at the current time step are taken to the right side as follows:

$$\overbrace{u_{i,j}^{n+1}}^{unknown} = u_{i,j}^n - \frac{\Delta t}{\Delta x}u_{i,j}^n\left(u_{i,j}^n - u_{i-1,j}^n\right) - \frac{\Delta t}{\Delta y}v_{i,j}^n\left(u_{i,j}^n - u_{i,j-1}^n\right) + \underbrace{\frac{\nu\Delta t}{\Delta x^2}\left(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n\right) + \frac{\nu\Delta t}{\Delta y^2}\left(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n\right)}_{known}$$

$$\overbrace{v_{i,j}^{n+1}}^{unknown} = v_{i,j}^n - \frac{\Delta t}{\Delta x}u_{i,j}^n\left(v_{i,j}^n - v_{i-1,j}^n\right) - \frac{\Delta t}{\Delta y}v_{i,j}^n\left(v_{i,j}^n - v_{i,j-1}^n\right) + \underbrace{\frac{\nu\Delta t}{\Delta x^2}\left(v_{i+1,j}^n - 2v_{i,j}^n + v_{i-1,j}^n\right) + \frac{\nu\Delta t}{\Delta y^2}\left(v_{i,j+1}^n - 2v_{i,j}^n + v_{i,j-1}^n\right)}_{known}$$

For our problem, the periodic boundary conditions are utilized. That means when a point gets to the edge of the domain, it would wrap around back and re-enter the domain from the opposite side. The periodic boundary conditions for the two dimensional domain are:

$$\phi_{0,j} = \phi_{n_x,j}$$
$$\phi_{i,0} = \phi_{0,n_y}$$

where ϕ represent either $v_{i,j}$ or $u_{i,j}$, and n_x and n_y represent the number of grid points in the x and y directions respectively.

Code Implementation

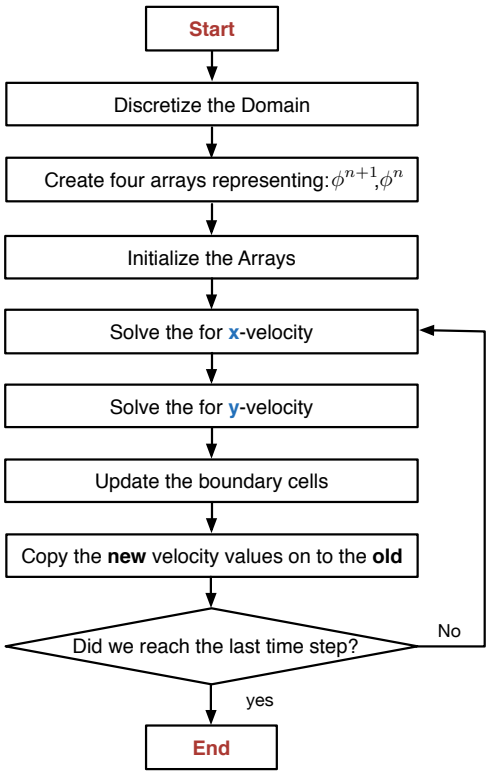


Fig. 2: Program implementation of the python code that solved the Burger's equation

The 2D Burgers' Equation solver that was given to us was implemented on Python script. Below are the steps the code uses to solve the Burgers' equation (Figure 2 represents a diagram of these steps):

- The code would start by discretizing the spatial grid according to Δx and Δy.
- Then the code will create four arrays that should represent the new and old, x & y velocities. Each array will have its own ghost and inner cells.
- The next step would be to initialize the domain. For the current case we chose the below equation to initialize the domain:

$$u_{i,j} = sin(j\Delta x)cos(i\Delta y) + 2$$
$$v_{i,j} = sin(j\Delta x)cos(i\Delta y) + 2$$

  where the minimum value is 2 and the maximum value is 3.
- When everything has been setup, the code will enter a loop process to solve the Burgers' equation. Inside the time loop, the code will do the following steps:
  - Solve the Burgers' equation by looping over the internal points of the domain. During the

looping process the code will solve for the x-velocity and then the y-velocity.
- After the code finishes solving the Burgers' equation, it updates the boundary conditions of the domain.
- Finally it copies the data of the new velocity on to the old velocity.

Optimization of the Serial Code Performance

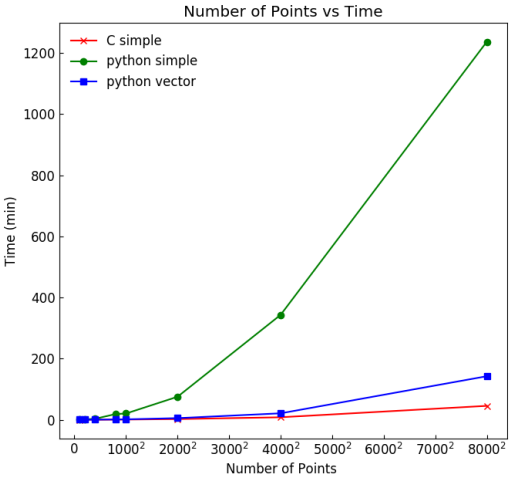
The first task we did was convert the python script given to us into a vectorizable python format, through implementing the `numpy.py` library. The vectorization of the code in python increased the speed by about **x8.6** for an 8000x8000 mesh (The maximum we could go to in serial). However this was not enough and so we converted it into the c language (non-vectorized) and got a speed up of about **x27.5** times faster than the original python code and **x3.2** times faster then the vectorized python code for an 8000x8000 mesh. The results of the performance comparison done on the three codes is shown in figure 3. The results show that in case of memory utilization, the difference between the codes is constant, indicating that its probably due to the libraries being loaded by python. As for the performance the c code showed that for larger meshes its computational time much lower than both python vectorizable and non-vectorizable versions.

Big O

After finishing our performance analysis on the three codes, we wanted to find out the Big O notation of the program, to predict its performance for a higher number of points. Not surprisingly we found out that the performance is linear with the total number of points $O(n)$. To confirm that we plotted in figure 4 the expected performance from $O(n)$, and the real performance. Starting from point 100^2 to 4000^2, the expected curve was on top of the real data.

However for the 8000^2 we saw a jump in the real computational over the expected trend of the $O(n)$. This is not a big surprise. The reason behind this jump is because in the 8000x8000 case, there are 4 matrices of size 8000^2 that were created but could not be stored on the caches, and had to go to the RAM. This caused the reading and writing of the cpu time to be much slower than usual. A rough calculation would prove that since we have an 8000x8000 mesh. Then in total each matrix will be of size $8000^2 \times 8$bytes=5.12×10^8bytes=512Mb, and since we have 4 of these matrices than they exceed the size of the L1, L2, and L3 cache for any machine I know.

(a) Computational Time



(b) Memory

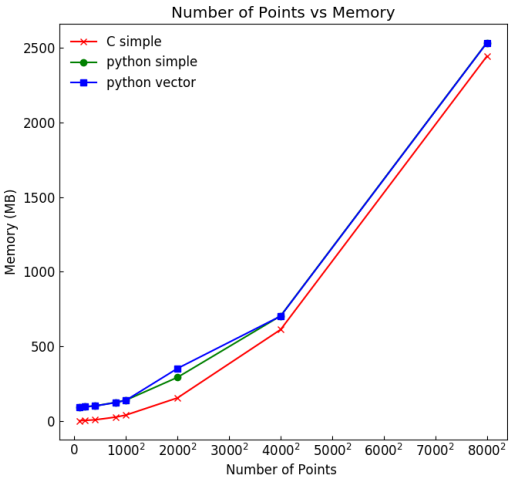


Fig. 3: (a) Computational time versus number of grid point for the serial version of the code, (b) Memory used vs number of grid point for the serial version of the code

Profiling

The next step was profiling the code performance to find out which case is taking the most computational time. Figure 5 shows the results of profiling done on the c code. As expected most of the time is going into solving the equation (yellow), while some time is being spent on copying the data (green). As the mesh size increases from 200x200 to 8000x8000 we notice that the copying time increases from 5% to 12%. Before we parallelize the code, we hope to lower the copying time, since it is such a trivial task that has nothing to do with the computation.

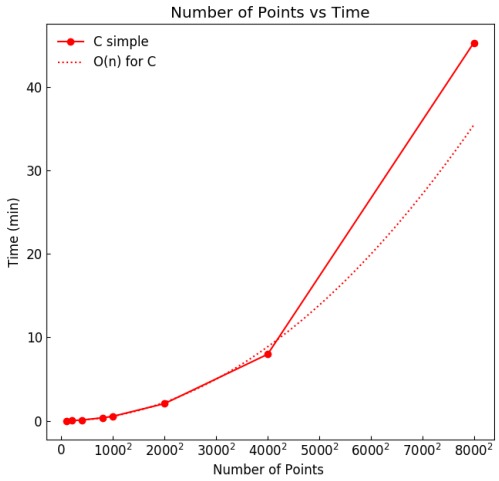


Fig. 4: Comparison between the expected Big O and real results

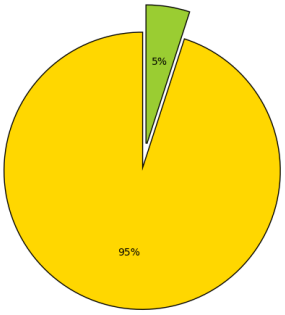
Updating the Serial Code

As stated previously one problem that the serialized ”c” version of the code was experiencing is the increase in time it takes the code to copy the new matrices (u^{n+1},v^{n+1}) onto the old matrices (u^n,v^n) as the matrix get larger, as shown in the profiling results of figure 5.

To fix this problem we removed the copy function and instead used the new matrices (u^{n+1},v^{n+1}) to solve the Burgers equation and their solution would be the updated-new matrices (u^{n+2},v^{n+2}). But instead of storing matrices (u^{n+2},v^{n+2}) in a new array, we stored them in the old matrices (u^n,v^n). So in general instead of copying (u^{n+1},v^{n+1}) in to (u^n,v^n), we solved for (u^{n+1},v^{n+1}) to get (u^{n+2},v^{n+2}), and we stored this in (u^n,v^n). By doing this step we removed the copy function and replaced it with an additional time step that involve solving the equation.

One problem with this approach is that the time increment changed. Previously, the code was moving one time step per one time loop; now the code is moving two time steps per one time loop since it is solving the Burgers Equation twice. Ultimately, this is not a problem since the code is solving more than 10,000 time steps. An example of the difference between the original serial code and the updated serial code is shown below :

(a) Small case 200x200



(b) Large case 8000x8000

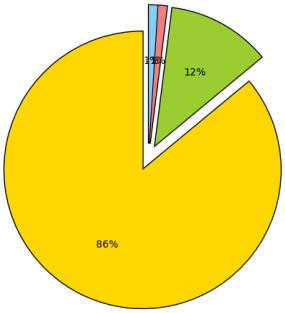


Fig. 5: Charts showing the profile data for (a) a small case 200x200, (b) a large case 8000x8000. (Yellow = Computational time, Green = Copying Arrays, Blue = Update Boundary, and Red = Writing output)

Listing 1: Original Serial Code

```
//Time Loop at an increment of n++
for(n=1;n<nt;n++)
{
  solve_equation();
  // Use "un" to get "un+1"
  // and use "vn" to get "vn+1"

  update_boundary();
  // Update boundary of
  // "un+1" and "vn+1"

  copy_arrays();
  // copy "un+1" and "vn+1"
  // to "un" and "vn"

  if(n % auto_save == 0)
  write_output(n);
}
```

Listing 2: Updated Serial Code

```
//Time Loop at an increment of n+2
for(n=1;n<nt;n+=2)
{
  solve_equation();
  // Use "un" to get "un+1"
  // and use "vn" to get "vn+1"

  update_boundary();
  // Update boundary of
  // "un+1" and "vn+1"

  solve_equation2();
  // Use "un+1" to get "un+2"="un"
  // and "vn+1" to get "vn+2"="vn"

  update_boundary2();
  // Update boundary of
  // "un" and "vn"

  if(n % auto_save == 0)
  write_output(n);
}
```

The difference in performance between the original serial code and the updated serial code is shown in figure 7. The figure shows that for small problems ($< 1000 \times 1000$), the computational time difference between the two codes is not worth the effort. However for larger problems, for example $12,000 \times 12,000$ the figure shows that the updated serial code is **x1.35** faster than the original code. This is a good improvement in serial performance speed, because in general some cases would take 5 days to finish on the original code, however with this improvement it would take around 3.7 days.

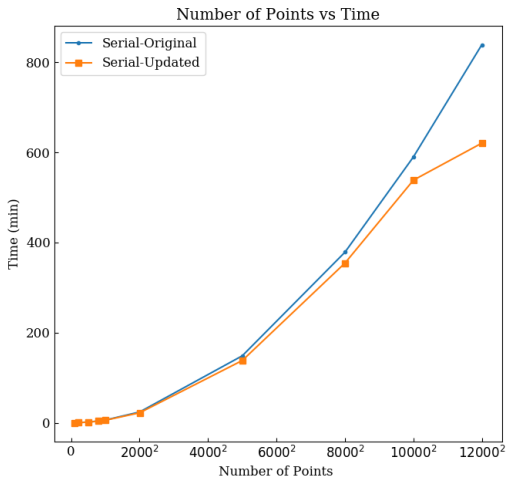


Fig. 6: Computational time versus number of grid point for the original code and updated code.

Parallelization of Burgers' Equation

Thus far, all of our work on the Burger's equation code lead to improvement in the serialized performance of the code. This section will detail the approaches taken to parallelize the Burgers equation. We will present three methods implemented in this report for parallelization

1- Coupled Approach:

The first approach refers to a method were the partial differential equations are divided over the threads. Instead of solving the equation one at a time, the coupled approach would solve the two equations together (provided that the two equations have no dependency on each other).

For the current case, the 2D Burgers equation, there are two equations that the code is solving at every time step; An equation for the **x-velocity** ($u^{n+1} = f^u(u^n, v^n)$), and an equation for the **y-velocity** ($v^{n+1} = f^v(u^n, v^n)$). Thus implementing the coupled approach requires the machine to provide the code with 2 cores (since there is only 2 equations).

If MPI was chosen as the parallelization module, then the amount of communication between the two cores would be very high especially for large problems. This is because in solving the equations the solver requires the knowledge of both u^n and v^n. Each core would only possess one of the two matrices, so for them to acquire the other matrix they have to receive it from the other core, and for example if the matrix was 8000×8000 which is approximately 512 Mb, then the communication time to send this matrix would exceed the time to solve it. For this reason we resorted to **OpenMP**, because it utilizes a shared memory space, and wont get affected by the communication time.

The parallelization of the serial code using the couple approach is very easy; all that is required is to assign which thread to solve which equation. Since each thread is solving a different equation each thread will write the solution on a different matrix. The problem of threads writing on the same memory is avoided. The figure below shows an example of the difference between the serial and parallelized version of the code:

Listing 3: Updated Serial Code

```
// Loop over the x points
for (i = 1; i < ny -1; i ++)
{
  // Loop over the y points
  for (j = 1; j < nx - 1; j++)
  {
    // Solve for the x-velocity
    un[i][j] = solve_un(u[i][j],v[i][j])

    // Solve for the y-velocity
```

```
    vn[i][j] = solve_vn(u[i][j],v[i][j])
  }
}
```

Listing 4: OpenMP-Coupled Approach

```
#pragma omp parallel
#pragma omp private(i,j,rank)
if(rank == 0) // OpenMP thread = 0
{
// Loop over the x & y points
for (i = 1; i < ny -1; i ++)
  for (j = 1; j < nx - 1; j++)
      // Solve for the x-velocity
     un[i][j] = solve_un(u[i][j],v[i][j
         ])
}
else // OpenMP thread = 1
{// Loop over the  x & y points
for (i = 1; i < ny -1; i ++)
  for (j = 1; j < nx - 1; j++)
     // Solve for the y-velocity
     vn[i][j] = solve_vn(u[i][j],v[i][j
         ])
}
```

The difference in performance between the original serial code, the updated serial code, and the OpenMP-Coupled is shown in figure 7. The figure shows that for larger problems, for example $10,000 \times 10,000$ the OpenMP coupled approach was **x1.85** faster than the updated serial code. So if a problem would take five days to finish on the serial code, it would take around 2.7 days using the OpenMP coupled approach.

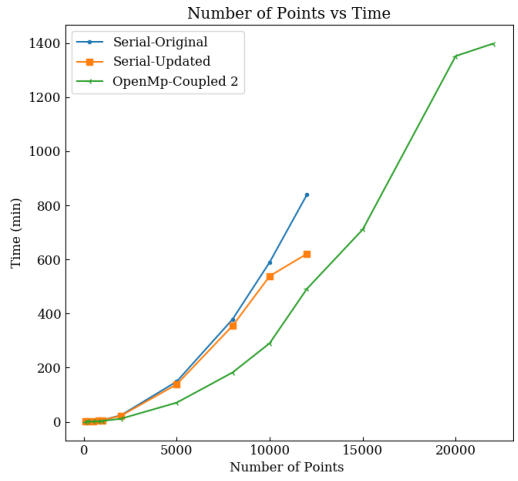


Fig. 7: Computational time versus number of grid point for the serial codes vs the OpenMP coupled approach.

Disadvantage: The disadvantage behind this approach is that the maximum number of cores is related to the number of equations, so for the 2D Burgers equation the maximum cores that can be used 2.

2- Domain Decomposition

This section details the **domain decomposition** approach. As stated previously this approach is based on the idea that the domain of the problem can be represented as a union of two or more sub-domains, where each of them is a restriction of the original problem, and takes on a particularly convenient form [8]. The advantage of reformulating a large discrete problem as a collection of smaller problem, is to enable the small discreet problems to be solved independently, and thus exploit the idea of parallelism. The parallel programming paradigms that will be implemented for this approach are **MPI** and **OpenMP**[4], each with its own advantage and disadvantage.

In order to implement the domain decomposition approach using OpenMP, what is required is to divide one of the spatial for loops according to the number of threads. For example if two threads are given, and the x-loop goes from the point with increment 0 to 1000, then each thread will loop over 500 point, where the first starts from the point with increment 0, and the second starts from the point with increment 500.

However implementing the domain decomposition approach using MPI proved to be much more difficult than expected due to the fact that the MPI paradigram doesn't allow shared memory across the MPI-threads. For this reason each MPI-thread will need to have its own matrices, but the good thing about these matrices is that their size is much smaller than the original serial matrix, in fact their size is equal to $1/number\_of\_mpi\_threads\times$ original size. The other difficult thing about this approach is the communication between the cores. Each core needs to send information about its boundary internal cells to the other cores and receive from them information about their boundary cells. Figure 8 shows the difference between the two approaches.

The results of the simulation are shown in the figure 9. For the OpenMP approach, figure 9a shows that for a large case like $10,000 \times 10,000$ the 2, 4,8, and 16 cores were **x1.91**, **x2.53**, **x6.48** and **x18.307** faster than the Serial case. As for the MPI approach it showed for 2, 4, 8, 16 and 32 cores to be **x1.94**, **x3.80**, **x11.26**, **x17.64** and **x23.93** faster than the Serial case. The results show a **superlinear** speedup in the OpenMP 16 Cores, and the MPI 8 and 16 cores. This indicates that there are problems with serial code for large cases, one example is inefficient cache (the L1,L2, and L3 caches are being filled up causing the cpu to call data from the RAM which is a lot slower).

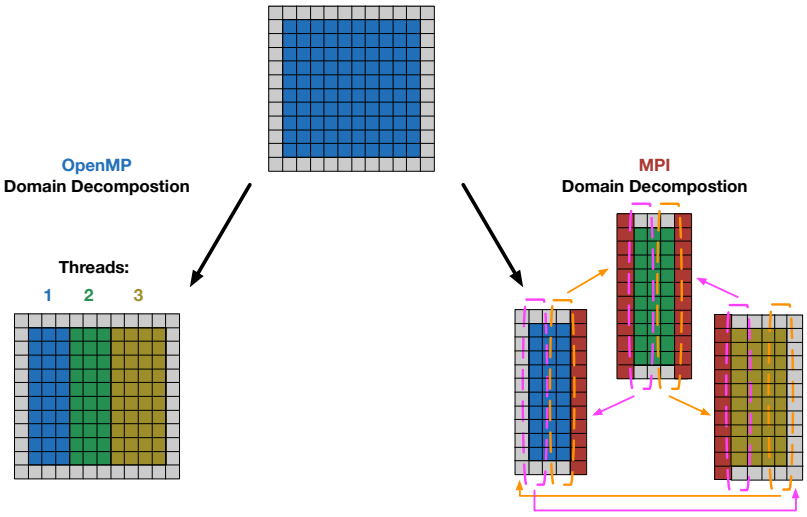
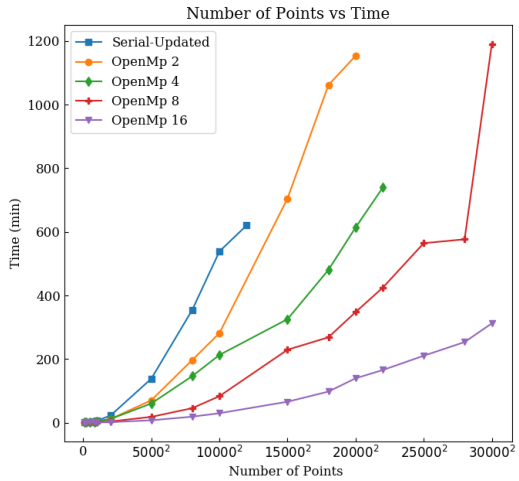


Fig. 8: Sketch of the domain decomposition approach done on OpenMP and MPI

(a) OpenMP



(b) MPI

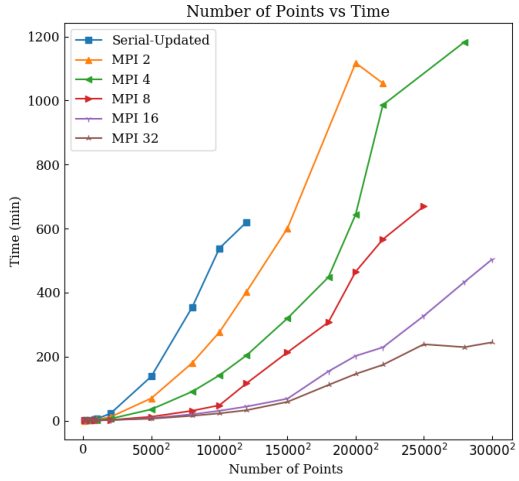


Fig. 9: Computational time versus number of grid point for the serial version of the code (a) OpenMP, (b) MPI

3- Both Approaches

In the final approach we combined both the coupled approach with the domain decomposition approach, and implemented them both using the **OpenMP** parallel module. So in this approach we decomposed the domain into (N/2) subsections where N represent the number of threads, and then assigned for the even threads to solve equation 1, and the odd threads to solve equation 2. By doing so, we hoped to get the best of both approaches. The performance of this approach is shown in figure 10. The figure shows for $10,000 \times 10,000$ the 2, 4,8, and 16 cores were **x1.85**, **x3.09**, **x6.69** and **x14.35** faster than the Serial case.

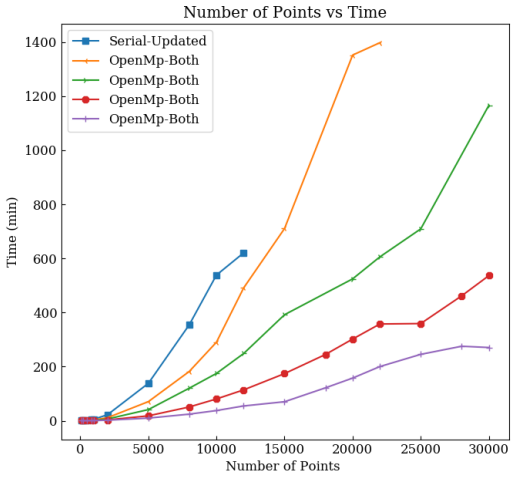


Fig. 10: Computational time versus number of grid point for the serial codes vs the OpenMP coupled approach.

Comparison with OpenFOAM

In this section, we compare the performance of our parallelized codes, with the free, open source CFD software called "OpenFOAM". Before the comparison is shown, we will discuss the difference between the two codes. We will start by describing how each code creates it own computational domain. Our code is created for 2D problems, thus it builds the domain by creating a 2D array. Moreover, our code assumes uniform discretization in the spatial domain, this enables it to link the location of the cell element in the 2D array with its respective index. OpenFOAM on the other hand is built for large, non-uniform, 3D problems, thus need a set of arrays that explain their domain. OpenFOAM has a total of four arrays just to explain where each point is located. As for solving the burgers equation, our code simply takes the value of velocities of the point and its neighbors and insert them into the burger equation to get the answer. OpenFOAM on the other hand, has to interpolate the values from the cell center to the face center, then insert the face center values in to the burger equation to get the answer.

In summary, our code is optimized to solve a special case, while OpenFOAM is programmed to run all sort of complex cases. Nevertheless, we will still compare the performance of our code to the OpenFOAM code. The parallelization technique implemented in OpenFOAM is similar to the MPI domain discretization implemented in our case. Figure 11 compares the performance of our 3 codes vs OpenFOAM. It is clear from the figure that our three codes where much faster than OpenFOAM solver for all number of cores, the difference in performance was around from **x7** times in favor of our solvers.

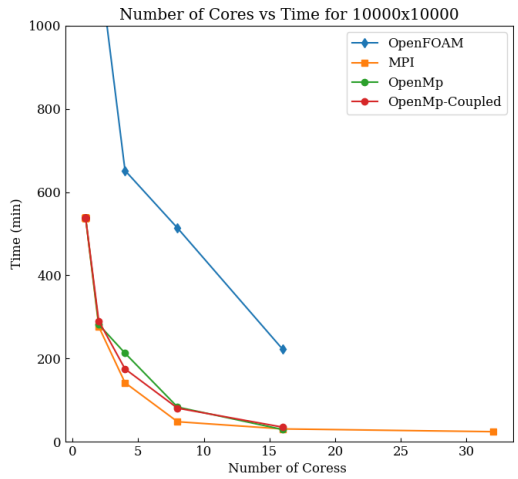


Fig. 11: Computational time versus number cores for the 10000x10000 case, of our codes vs the OpenFOAM.

Summary

A summary of all the results is shown in table 2. The table uses the data from the Serial-Updated case as a reference for the computational speed up. The reason Serial-Updated was chosen is because it represents the case upon which all the parallelized models where applied. In some cases there was no data for the Serial-Update model for this reason the reference became the Coupled-OpenMP-2cores, and if it didn't have any data then the reference became Domain-Decomposition-OpenMP-8cores.

One interesting observation from the table is that for small cases (cases that fit on the L1 & L2 cache) parallelization harms the performance. This could be seen for case 101^2 in which parallelization of more than 2 cores dropped the performance to **x0.11** for 32 cores, especially for domain decomposition alone. However the real interesting results lies in the cases that showed super linear speedup, which occurred for cases with more than 8 cores. For example domain decomposition with OpenMP 16 core was able to achieve a speed up of about **x19.24**, while the MPI 16 core reached a speedup of **x18.15**. The cases with cores = 32 and 64 were not able to achieve a superlinear-speed up probably due to the fact that the some of the machines were being loaded, but the more probable reason is because the communication time was increased in comparison to the computational time.

	Serial		Coupled	Domain Decomposition										Both			Domain Decomposition				
Type	Original	Updated	OpenMP	OpenMP				MPI						OpenMP			OpenFOAM				
# Cores	1	1	2	2	4	8	16	2	4	8	16	32	64	4	8	16	1	2	4	8	16
# Points	Speed up in Computational time in comparison to the Blue numbers																				
101^2	0.93	**1**	1.95	1.74	**0.22**	4.47	8.62	1.76	1.34	**0.28**	**0.28**	**0.11**	**0.43**	3.34	2.70	5.29	**0.28**	**0.41**	**0.02**	**0.10**	**0.01**
201^2	0.91	**1**	1.95	1.92	**0.57**	6.92	12.39	1.85	3.15	**0.82**	**0.86**	**0.20**	1.15	3.71	6.80	10.13	-	-	-	-	-
501^2	0.92	**1**	1.97	1.97	1.26	7.62	**17.79**	1.90	3.81	8.50	12.98	1.35	16.24	3.92	7.51	12.67	-	-	-	-	-
801^2	0.93	**1**	1.99	1.98	1.51	7.79	**17.08**	1.93	3.71	**9.21**	6.31	5.68	1.92	3.94	7.67	8.66	-	-	-	-	-
1001^2	0.92	**1**	1.98	1.98	1.57	4.22	**18.54**	1.89	3.85	**9.45**	10.59	4.52	3.42	3.95	7.95	10.50	**0.29**	**0.56**	**0.71**	**0.85**	1.19
2001^2	0.92	**1**	1.98	1.98	1.95	6.37	**18.96**	1.96	3.92	**10.90**	15.87	10.93	10.79	3.49	7.69	11.76	-	-	-	-	-
5001^2	0.93	**1**	1.96	1.98	2.30	7.73	**19.24**	1.98	3.94	**11.52**	**18.05**	23.79	26.05	3.36	7.69	14.42	**0.20**	**0.46**	**0.62**	1.24	1.77
8001^2	0.94	**1**	1.95	1.96	2.43	7.83	**19.24**	1.97	3.91	**11.57**	**18.15**	23.73	26.66	2.95	6.98	14.61	-	-	-	-	-
10001^2	0.91	**1**	1.86	1.80	2.53	6.49	**19.21**	1.95	3.80	**11.27**	**17.64**	23.93	27.75	3.08	6.70	14.35	**0.14**	**0.46**	**0.83**	1.05	2.42
12001^2	0.74	**1**	1.26	1.26	1.91	2.94	**18.31**	1.54	3.04	5.31	14.15	19.12	-	2.50	5.44	11.37	-	-	-	-	-
15001^2	-	-	**1**	1.01	1.48	3.10	**13.07**	1.18	2.22	3.33	**10.39**	12.14	-	1.81	4.08	**10.13**	-	-	-	-	-
18001^2	-	-	**1**	1.27	1.73	3.97	**10.84**	-	2.38	3.46	6.92	9.58	-	-	**4.35**	**8.76**	-	-	-	-	-
20001^2	-	-	**1**	1.21	1.83	3.89	**10.93**	1.21	2.10	2.91	6.69	9.28	-	2.58	**4.48**	**8.58**	-	-	-	-	-
22001^2	-	-	**1**	-	-	3.29	**9.71**	1.33	-	2.47	6.11	8.02	-	2.31	3.91	6.98	-	-	-	-	-
25001^2	-	-	-	-	-	**1**	**2.68**	-	-	-	1.73	2.37	-	-	1.57	2.30	-	-	-	-	-
28001^2	-	-	-	-	-	**1**	**2.28**	-	-	-	1.33	2.51	-	-	1.25	2.09	-	-	-	-	-
30001^2	-	-	-	-	-	**1**	**3.82**	-	-	-	**2.37**	4.48	-	-	**2.22**	**4.40**	-	-	-	-	-

* Reference Computational Time
* Speed Down (Not Good)
* Super Linear Speed Up

References

[1] J. Baker, A. Armaou, and P. D. Christofides. Nonlinear control of incompressible fluid flow: Application to burgers' equation and 2d channel flow. *Journal of Mathematical Analysis and Applications*, 252(1):230–255, 2000.

[2] M. Balogh, A. Parente, and C. Benocci. Rans simulation of abl flow over complex terrains applying an enhanced k-ε model and wall function formulation: Implementation and comparison for fluent and openfoam. *Journal of wind engineering and industrial aerodynamics*, 104:360–368, 2012.

[3] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[5] W. D. Gropp. Parallel computing and domain decomposition. In *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA*, 1992.

[6] R. Haberman. *Mathematical models: mechanical vibrations, population dynamics, and traffic flow*. SIAM, 1998.

[7] M. F. Hamilton, D. T. Blackstock, et al. *Nonlinear acoustics*, volume 1. Academic press San Diego, 1998.

[8] D. E. Keyes and W. D. Gropp. A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. *SIAM Journal on Scientific and Statistical Computing*, 8(2):s166–s202, 1987.

[9] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr, M. E. Zosel, D. E. Ulberg, A. J. Mallinckrodt, S. McKay, et al. The high performance fortran handbook. *Computers in Physics*, 8(4):428–428, 1994.

[10] J. M. Ortega, R. G. Voigt, S. for Industrial, and A. Mathematics. *Solution of partial differential equations on vector and parallel computers*. Society for Industrial and Applied Mathematics, 1985.

[11] T. Passot and E. Vázquez-Semadeni. Density probability distribution in one-dimensional polytropic gas dynamics. *Physical Review E*, 58(4):4501, 1998.