ECE563 - Programming Parallel Machines - Project

Kameswararao Anupindi, Niranjan Ghaisas
School of Mechanical Engineering, Purdue University

May 4, 2012

Contents

List of Figures

List of Tables

1 Statement of the Project

This project is concerned with studying the parallel performance of two numerical approaches used for simulation of fluid flows. The first approach is a traditional finite-difference based computational fluid dynamics (CFD) solver name WenoHydro, which solves the incompressible Navier-Stokes equations [3]. The second approach is based on the Lattice Boltzmann method (LBM), which solves the Boltzmann transport equations for fluid flow at mesoscopic length scales [1, 2, 5]. Both the WenoHydro and the FlowLBM solvers are written in Fortran90. The codes are parallelized using domain decomposition with calls to subroutines of the Message Passing Interface (MPI) libraries for exchanging data among processors. The parallel performance of the codes is evaluated by conducting numerical experiments on the Rossman cluster available through Purdue RCAC. RCAC servers use PBS script files for managing the job submission process on the cluster, and the same is used for the present simulations.

Various metrics can be defined for evaluating the performance of a parallel code. Three metrics which will be considered in this report are given below:

- Speedup: $\psi = \frac{T_1}{T_P}$
- Efficiency: $E = \frac{\psi}{P}$
- Karp-Flatt metric: $e = \frac{1/\psi - 1/P}{1-1/P}$

In the above definitions, P denotes the number of processors, and T_P denotes the time required for execution of the code on P processors. The serial time T_1 can be computed either as the time required for execution of the serial version of the code (T_s), or as the time required for execution of the parallel version of the code ($T_{P=1}$). The first, and most intuitive, metric is the speedup, ψ, which is the ratio of serial execution time to parallel execution time. The second metric, defined as the ratio of speedup to the number of processor employed, is the efficiency of parallel execution. The Karp-Flatt metric measures the fraction of the time spent by processors in a parallel execution for performing serial computation.

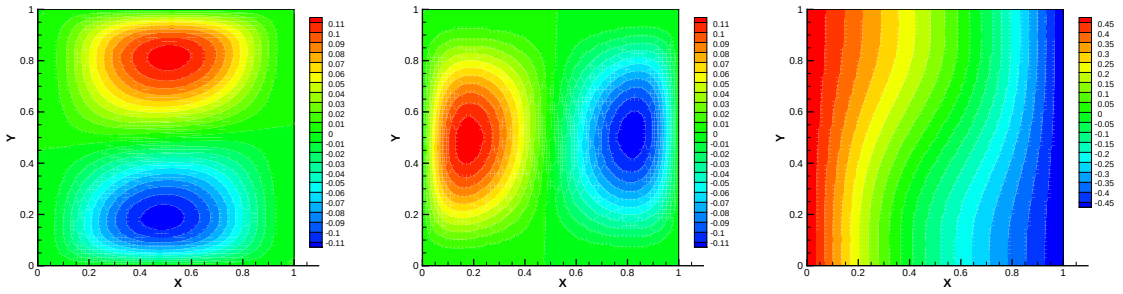
The next section presents details about the WenoHydro algorithm and its evaluation. The FlowLBM solver and its parallel performance are then presented, followed by a summary and conclusion.

2 WenoHydro

The parallel performance of a computational fluid dynamics (CFD) code named WenoHydro is evaluated in this section. WenoHydro simulates the flow of an incompressible fluid with a buoyant force caused by density differences. A brief description of the code, the test problem considered and sample fluid dynamic results are given in the next subsection. This is followed by details about the baseline parallel algorithm. Three further modifications to the baseline algorithm are then described. Parallel execution characteristics of the baseline and modified algorithms are reported and analyzed in Section 2.3.

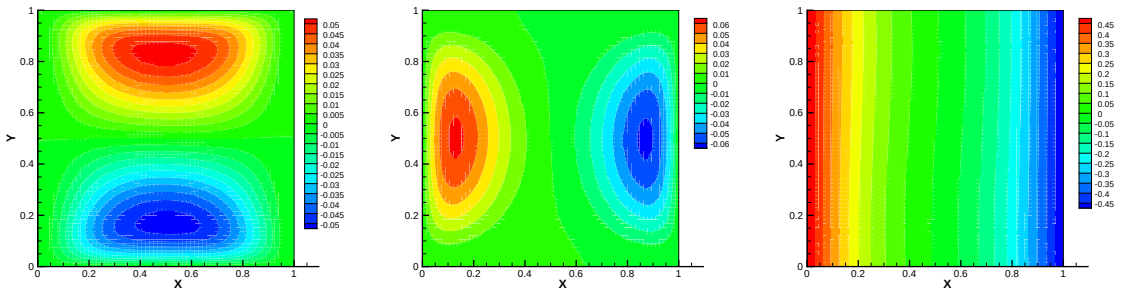
2.1 Code and Sample Results

The WenoHydro code is a CFD code written in Fortran90, employing high-order numerical methods for convection, diffusion and time stepping. The OpenMP version of the code has been validated previously in two and three dimensions for simulating laminar and turbulent flows [3]. An MPI version of the code has been written recently and the aim of this project is to analyze and improve its parallel performance. Two dimensional flow driven by heating the left wall and cooling the right wall of a square cavity is used as a test case. The differential heating of the walls sets up steady convection patterns in the cavity. Starting from a cavity filled with fluid with uniform temperature at rest at non-dimensional time $t = 0$, the steady state solution is attained at large non-dimensional times, $t > 10$. The horizontal velocity, u, vertical velocity, v, and temperature contours at steady state for one particular strength of the differential heating are shown in Figure 1. In order to evaluate the speedup, efficiency, Karp-Flatt and iso-efficiency metrics, the code is executed from $t = 0$ to $t = 1$. Contours of u, v and T at this intermediate time instant of non-dimensional $t = 1$ are shown in Figure 2. All simulations described below, using different algorithms, and on different numbers of processors, yield results identical to those in Figure 2.



(a) u (b) v (c) T

Figure 1: Steady state solution at non-dimensional time $t = 10$ of the flow in a two-dimensional differentially heated square cavity.



(a) u (b) v (c) T

Figure 2: Intermediate solution at non-dimensional time $t = 1$ of the flow in a two-dimensional differentially heated square cavity. All simulations reproduce these results.

2.2 Baseline and Modified Algorithms

The WenoHydro code solves the incompressible Navier-Stokes equations using a 5^{th}-order scheme for the convective terms, a 4^{th}-order scheme for the viscous terms and a 3^{rd}-order explicit Runge-Kutta scheme for time stepping [3]. Solution of incompressible Navier-Stokes equations also involves solving a Poisson equation for the pressure for every stage of the time stepping scheme. For solving the pressure Poisson equation, the WenoHydro code makes use of the Schur decomposition technique, which is an inherently parallel variant of the exact (non-iterative) LU decomposition method [4].

The overall algorithm for advancing the solution from time step n to time step $n+1$ involves three predictor stages and four corrector stages. Each predictor step involves computing derivatives for the convective and viscous terms, while each corrector step involves setting up and solving a Poisson problem for the pressure. The algorithm can be briefly expressed as:

- **Predictor 1:** Compute derivatives for the convective and viscous terms based on the available velocity field, u_i^n. Use these to compute intermediate solutions u_i^* and u_i^{**}.
- **Corrector 1:** Set up and solve a Poisson problem for pressure and correct u_i^*.
- **Corrector 2:** Set up and solve a Poisson problem for pressure and correct u_i^{**}.
- **Predictor 2:** Compute derivatives for the convective and viscous terms based on the intermediate velocity field, u_i^{**}. Use these to compute intermediate solution u_i^{***}.
- **Corrector 3:** Set up and solve a Poisson problem for pressure and correct u_i^{***}.
- **Predictor 3:** Compute derivatives for the convective and viscous terms based on the intermediate velocity field, u_i^{***}. Use these to compute solution at the next time step u_i^{n+1}.

- **Corrector 4:** Set up and solve a Poisson problem for pressure and correct u_i^{n+1}.

Arrays for the velocity, temperature and pressure fields are distributed across processors arranged in a two-dimensional grid (2D decomposition). Each processor performs computation on its parts of the global arrays. Each processor also sends and receives data to and from other processors in order to facilitate this computation. Advancing the solution by one time step (through three predictor and four corrector stages) requires both local communication between adjacent left, right, top and bottom processors, and global communication between all processors. The local communication involved is:

1. Three layers (rows or columns) of velocities and temperature before every predictor stage for computing intermediate velocities,
2. One layer of velocities before every corrector stage for setting up the Poisson problem,
3. One layer of pressure values after every corrector stage for correcting the velocities.

One global communication episode with one call to MPI_AllReduce is required during every corrector stage for solving the pressure Poisson problem.

The global communication in the total communication described above is part of the parallel Poisson solver, and cannot be improved upon. The local communication operations however, can be implemented in a number of ways, such as by changing the manner of accessing data, by splitting up messages into smaller chunks resulting in larger number of smaller messages, and increasing the computation on individual processors so as to altogether eliminate one or more communication steps. A baseline implementation and three different versions based on these ideas are described below.

- **Version 1**
  The baseline version of the implementation carries out all the three communication operations described above. Before each communication, the data to be sent is copied into buffers reserved exclusively for sending messages. The data is sent from 'send' buffers to exclusively reserved 'receive' buffers, and finally copied out from the 'receive' buffers into proper velocity, temperature and pressure arrays. In this version, all the data to be sent at a time is packed into one 'send' buffer.

- **Version 2**
  In the first modification, the 'send' and 'receive' buffers are eliminated, and data is accessed directly from and placed into appropriate locations in the velocity, temperature and pressure arrays using MPI derived data types. This modification leads to a reduction in the amount of required memory. In addition, it is expected that the time required to fill the 'send' buffer before the actual communication step, and the time required to copy out data from the 'receive' buffer after the communication step would be reduced.

- **Version 3**
  In the first modification, constructing the derived data types to be used for directly accessing data to be communicated is complicated by the fact that the block lengths and the distances between blocks are not uniform. A simpler way of accessing data elements involves constructing MPI derived data types for rows and columns first, and using these to build derived data types for accessing 2D sections of the array. However, this modification requires that messages to be sent be split up into three sub-messages, one each for the two velocity components and the temperature. This modification, with a simpler means of accessing data resulting in sends and receives of larger number of smaller messages, is contained in Version 3.

- **Version 4**
  In the final version, computation on each processor is increased so that each processor computes one additional layer of the computational grid in each direction, in addition to its share of the domain. This computation enables eliminating one local communication step, namely the communication of one layer of velocities before every corrector stage for setting up the Poisson problem. However, this also leads to the need for communication of two (instead of one) layers of pressure values for correcting the velocities. Thus, overall, the amount of data communicated is not reduced. However, one local communication step (Step 2) is eliminated, while the size of one local communication step (Step 3) is increased.

The above four versions have been implemented and are evaluated in the next subsection.

Table 1: Serial times in seconds using two alternatives

	Serial	Parallel, $P = 1$
$N = 64^2$	245.97	620.45
$N = 128^2$	974.35	4003.7

2.3 Parallel performance

The parallel performance of the above algorithms is evaluated in this section using speedup, efficiency and Karp-Flatt metrics. Execution times of the four versions described above have been recorded using calls to the MPI_WTIME subroutine. The serial execution time can be determined by either executing the serial code, or by executing the parallel code on a single processor, with appropriate conditional statements which ensure that no spurious communication occurs on the single processor. The effect of these alternatives will be demonstrated. Finally, an iso-efficiency analysis of the algorithm will be presented.

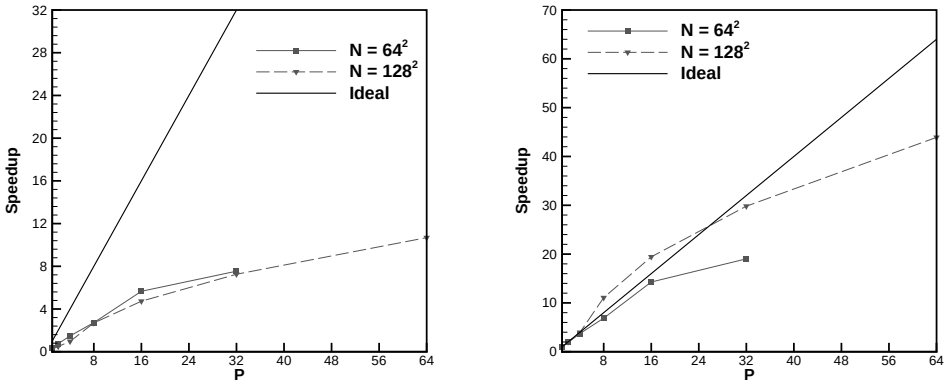
2.3.1 Effect of serial time

As mentioned above, the serial execution time can be determined using either the serial version of the code, or using the parallel version of the code executed on a single processor. Execution of the parallel code adds overhead related to communication and parallel execution, even when executed on a single processor. The overhead can be reduced by making use of conditional 'IF-ELSE' statements, which ensure that communication is carried out only if the number of processors is larger than one. Serial execution times computed using the serial version of the code and computed using single processor execution of the parallel code are tabulated in Table 1.

It can be seen from the table that the serial execution times using the two alternatives are drastically different. It should be recalled that solution of the incompressible Navier-Stokes equations requires solution of a Poisson problem for the pressure. The Poisson solver employed in the parallel version of the code is the inherently parallel Schur-decomposition method. On the other hand, the Poisson solver used in the serial version of the code is a multigrid solver. Thus, although the rest of the algorithms in the serial and parallel codes are identical, the Poisson solvers employed are completely different. This explains the fact that serial times obtained using the serial code and the serial times obtained using the single processor execution of the parallel code are significantly different. Figure 3 shows the speedups obtained using the two definitions for the serial time. As can be seen from the figure, the definition of serial time has a significant impact on the speedups in this case, and underscores the importance of choosing a correct definition for the serial time. For further analysis, we choose to consider the time taken for the single processor execution of the parallel code as the serial time. This choice has been made because the serial code employs a different algorithm for the Poisson solver, and thus, is not really relevant for the present purpose.

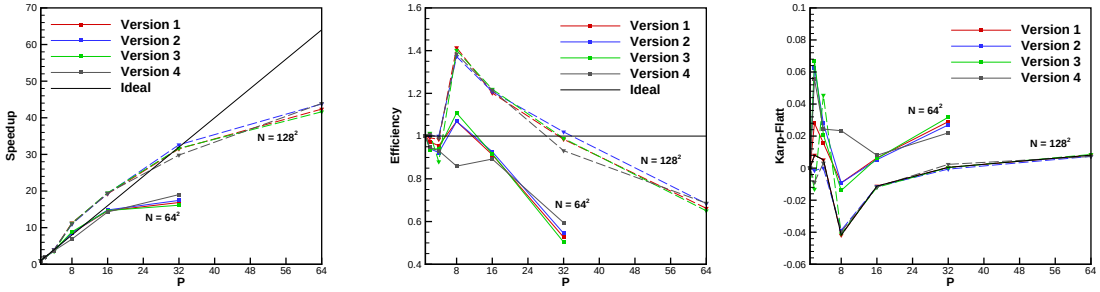
2.3.2 Speedup, Efficiency and Karp-Flatt metrics

Figure 4 shows the evolution of the speedup, efficiency and Karp-Flatt metrics with number of processors from $P = 1$ to $P = 64$ in multiples of 2, for two problem sizes of $N = 64^2$ and $N = 128^2$. It can be seen from Figure 4a that the codes show a super-linear speedup on up to 8 processors for the smaller $N = 64^2$ problem, and up to 16 processors for the larger $N = 128^2$ problem. These better-than-expected speedups result in efficiencies greater than unity in Figure 4b, and also yield negative Karp-Flatt metrics in Figure 4c. The speedup drops below the ideal value for the smaller sized problem on 16 and 32 processors. Similarly, the speedup is below its ideal value for the larger sized problem on 32 and 64 processors. The efficiencies, which are above unity for smaller number of processors reduce on increasing the number of processors. The efficiency reduces to around 60% for $N = 64^2$ sized problem on 32 processors, and to around 65% for $N = 128^2$ sized problem on 64 processors. The Karp-Flatt metric (e), which is nothing but the serial fraction of the parallel computation, is smaller for the larger size problem. Further, comparing the rate of growth of e for the two problem sizes between processors $P = 16$ and $P = 32$, it can be seen that e grows slightly faster for the smaller problem than for the larger problem. However, the metric e increases with increasing number of processors for both the problem sizes. This indicates that as the serial fraction increases with increasing number of processors, the speedup will eventually saturate and these problems will not scale indefinitely.



(a) Serial (b) Parallel, $P = 1$

Figure 3: Speedup of the parallel WenoHydro code based on two definitions of serial times. (a) Serial code (b) Parallel code, executed on single processor



(a) Speedup (b) Efficiency (c) Karp-Flatt

Figure 4: Parallel performance metrics for MPI parallel WenoHydro code on processors $P = 1$, 2, 4, 8, 16, 32 and 64, for two problem sizes, $N = 64^2$ and $N = 128^2$. (a) Speedup (b) Efficiency (c) Karp-Flatt metric.

2.3.3 Comparison of four versions

One striking feature, apparent from Figure 4, is that the four versions described in the previous section yield almost identical speedups for both the smaller problem and the larger problem. In order to investigate why reducing data access, eliminating operations associated with copying data into and out of buffers, and reducing communication does not result in reduction of the overall computation time, individual stages in one time step of the CFD algorithm have been tabulated in Table 2, along with averages of times required for each individual stage. This data has been obtained from Version 1 of the parallel code, and has been averaged over 1000 time steps. It can be seen that on an average, one complete time step takes 0.166 s, out of which only about 0.8% time is spent in local communication. The bulk of the time is spent in the computation of derivatives of velocities, and in the pressure Poisson solution. Thus, improvements to local communication would not lead to large improvements in performance, and this is reflected in the results shown in Figure 4.

2.3.4 Iso-efficiency analysis

In order to copmute an iso-efficiency function, estimates for the serial time and the parallel times are needed. For the 2D WenoHydro CFD code considered here, the time required for all computations in a serial execution is of the order $O(N^2)$. It can be noted that in a 3D code, the computations would be order $O(N^3)$. Thus,

$$T_1 = c_1 N^2$$

Table 2: Timing data for sub-parts of one Runge-Kutta time step of the WenoHydro code.

Stage	Operation	Time in seconds
Predictor 1	Total	0.022
	Communicate velocities	3.6×10^{-4}
Corrector 1	Total	0.025
	Communicate velocities and pressure	6.0×10^{-5}
Corrector 2	Total	0.025
	Communicate velocities and pressure	6.0×10^{-5}
Predictor 2	Total	0.022
	Communicate velocities	3.6×10^{-4}
Corrector 3	Total	0.025
	Communicate velocities and pressure	6.0×10^{-5}
Predictor 3	Total	0.022
	Communicate velocities	3.6×10^{-4}
Corrector 4	Total	0.025
	Communicate velocities and pressure	6.0×10^{-5}
Total time for RK3 step $= 3\,(0.022) + 4\,(0.025) = 0.166$ s		
Total communication time $= 3\left(3.6 \times 10^{-4}\right) + 4\left(6 \times 10^{-5}\right) = 0.00132$ s		

where c_1 is a constant. For a parallel execution on P processors, the computation on each individual processor reduces by a factor of P. This is accompanied by an increase of the execution time due to communication. As described earlier, all communication operations can be classified as either local or global communication operations. The local communication operations are independent of number of processors. With c_3 denoting the time taken for one communication step with neighbours, the time required for local communication with left, right, top and bottom neighbours is order $O(4c_3)$. Finally, global communication is involved in solving the pressure Poisson problem, and involves one MPI_AllReduce call for every corrector stage. Thus, the time for global communication is of the order $O(4 \log P)$, since each time step involves four corrector stages. Putting it all together, the parallel computation time is

$$T_P = c_2 N^2/P + 4c_3 + c_4 \log P$$

The overhead time is given as

$$\begin{aligned} T_o &= PT_P - T_1 \\ &= P\left(c_2 N^2/P + 4c_3 + c_4 \log P\right) - c_1 N^2 \\ T_o &\sim P \log P \end{aligned}$$

With $W = KT_o \sim P \log P$, the theoretical iso-efficiency analysis indicates that the work should increase at the rate of $P \log P$ with the number of processors P.

The iso-efficiency characteristics observed in our numerical experiments is now discussed. Figure 4b shows that for all versions, the efficiency of $N = 128^2$, $P = 32$ computations is slightly larger than the efficiency of $N = 64^2$, $P = 16$ computations. In other words, increasing the number of processors by a factor of 2, and simultaneously increasing the problem size by a factor of 4 results in a higher efficiency. This indicates that the iso-efficiency function is better than $O(P^2)$. The same can be concluded by considering data points corresponding to $N = 64^2$, $P = 32$ and $N = 128^2$, $P = 64$ computations. Figure 4b also shows that the efficiency for the $N = 128^2$, $P = 64$ computation is smaller than the efficiency of the $N = 64^2$, $P = 16$ computation. This indicates that the iso-efficiency function is not better than $O(P)$. Thus, our results show that the iso-efficiency function lies between $O(P)$ and $O(P^2)$, consistent with the theoretically predicted $O(P \log P)$. An analysis with more data points on the efficiency vs P plot can potentially show the exact $P \log P$ scaling.

3 FlowLBM

FlowLBM solver employs lattice Boltzmann method (LBM) [1, 2, 5] for the simulation of fluid flow. In this method the Boltzmann equations are solved in a mesoscopic limit which enables the simulation of incompressible fluid flow.

Each iteration of the solver includes the following steps:

- Collision step of the lattice points (can be executed in parallel)
- Apply boundary conditions (specific to each processor depending on whether the processor boundary coincides with the domain boundaries or not)
- Streaming/advection step (exchanges data between processors for the ghost/overlap regions)
- Computation of macroscopic quantities such as density, and velocities (can be executed in parallel)

3.1 The Simulation Method

LBM is a numerical method for solving the Boltzmann equation (1), where $f_\alpha(\mathbf{x}, t)$ is a set of distribution functions, which represents the probability of finding a particular particle at a position $\mathbf{x}$ at time t with a velocity $\mathbf{c}_\alpha$.

$$f_\alpha(\mathbf{x} + \mathbf{c}_\alpha \delta t, t + \delta t) - f_\alpha(\mathbf{x}, t) = \Omega_\alpha \tag{1}$$

The collision operator Ω_α models the collision of particles and controls the rate of approach of distribution functions to an equilibrium state given by f_α^{eq}. Bhatnagar, Gross, and Krook [2] proposed the BGK dynamics, where the collision operator Ω_α is approximated as a single-relaxation-time (SRT) model as given by equation. (2).

$$\Omega_\alpha = \frac{1}{\tau}\left[f_\alpha(\mathbf{x}, t) - f_\alpha^{eq}(\mathbf{x}, t)\right] \tag{2}$$

Where τ is the relaxation time related to the viscosity (ν) as follows:

$$\tau = \frac{3}{\delta t}\nu + \frac{1}{2} \tag{3}$$

The equilibrium distribution function $f_\alpha^{(eq)}$ is obtained by discretizing the Boltzmann distribution. In the present work the form given in equation (4) is used which involves terms in velocity up to second order,

$$f_\alpha^{eq}(\rho, \mathbf{u}) = w_\alpha\ \rho\ \left[1 + \frac{\mathbf{c}_\alpha.\mathbf{u}}{c_s^2} + \frac{(\mathbf{c}_\alpha.\mathbf{u})^2}{2c_s^4} - \frac{|\mathbf{u}|^2}{2c_s^2}\right] \tag{4}$$

with the lattice speed of sound $c_s = \frac{1}{\sqrt{3}}$ for the D2Q9 model and the lattice weights are given as follows:

$$w_{k=9} = 4/9 \tag{5}$$
$$w_{k=1,2,3,4} = 1/9 \tag{6}$$
$$w_{k=5,6,7,8} = 1/36 \tag{7}$$

and the lattice is defined by the following vectors as shown in equation (8), for $i = 1, 9$.

$$c_i = \begin{bmatrix} 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 & 0 \end{bmatrix} \tag{8}$$

The convention to define a lattice type is $DnQm$ where Dn denotes the dimension of the lattice with n set to 2 for 2D and 3 for 3D and Qm defines the number of lattice sites. For example $D2Q9$ lattice used in the present simulations is a 2D lattice and has 9 lattice sites or degrees of freedom.

The macroscopic observable quantities are given by the moments of the particle distribution function as follows. The integral of zeroth moment of distribution function gives the density and the first moment gives the momentum as described in equations (9,10).

$$\rho(\mathbf{x}, t) = \sum_{\alpha} f_{\alpha}(\mathbf{x}, t) \tag{9}$$

$$\rho(\mathbf{x}, t)\mathbf{u}(\mathbf{x}, t) = \sum_{\alpha} \mathbf{c}_{\alpha} f_{\alpha}(\mathbf{x}, t) \tag{10}$$

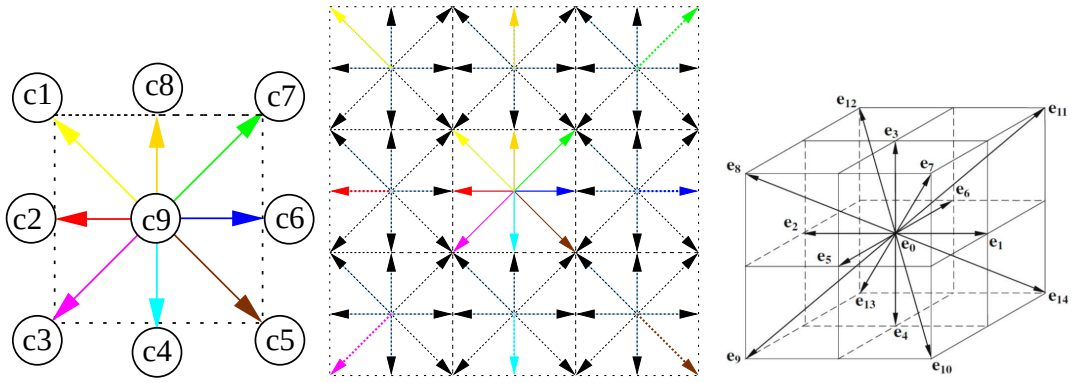
With this introduction to the numerical method we present the general algorithm to solve the LBM on a finite difference grid with a lattice attached to each of the grid point. The Algorithm to simulate using LBM consists of three main steps as described below.

3.1.1 Algorithm

The overall simulation algorithm is composed of the following three steps:

1. Collision
2. Streaming
3. Boundary conditions

In the collision step the particle distribution function f_{α} are relaxed towards their respective equilibrium value $f_{\alpha}^{(eq)}$ according to equation (2). In the streaming step each f_{α} is *streamed*, or *advected* to the respective lattice site dictated by its lattice vector $\mathbf{c}_{\alpha}$. The D2Q9 lattice is shown in Figure (5a) with the corresponding lattice vectors. The streaming operation is shown in Figure (5b). With this we can explain how the streaming/advection occurs which essentially represented on the left hand side of the equation (1). For example if we assume that the lattice in the center with all solid arrows in Figure (5b) is the location where streaming has to be performed, then the distribution function corresponding to each of the arrows/lattice sites are advected/streamed to the adjacent lattice as per the color coding and the final position where the distribution function is assigned is represented by the broken arrows. Similarly the rest of the arrows shown in black will also receive their values from their neighbors and so on. Care should be taken while doing streaming operation such that we do not stream same value all along. For e.g. if we use a copying mechanism then for copying the distribution function values for $C6$ on x-axis we should start at the x-max location first and march towards x-min location. If we do it other way round then the value of $C6$ at x-min gets copied to all the grid points because the distribution function at the next site gets over-written before they could be copied. The present simulations are three-dimensional and hence a 3D lattice needs to be made use of. The 3D lattice used in present simulations is $D3Q15$ and is shown plotted in figure 5(c).



(a) D2Q9 Lattice (b) Streaming operation (c) D3Q15 Lattice

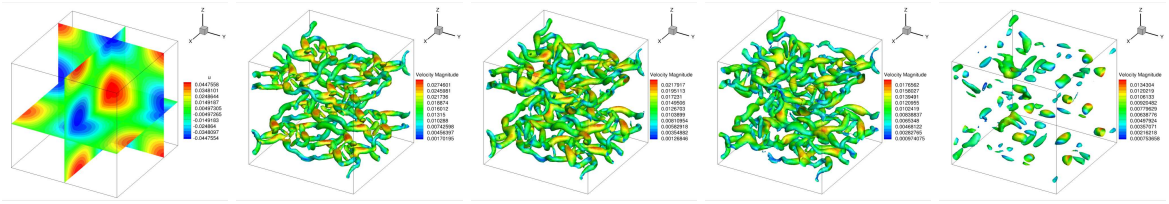
Figure 5: Schematic showing the lattice structure and streaming operation

3.1.2 Boundary conditions

In the present simulations periodic boundary conditions are used on all boundaries. These are the simplest type of boundary conditions. The periodic boundary condition implementation sets the outgoing distribution functions as incoming distribution functions on the other matching periodic boundary.

3.1.3 Simulation Results of FlowLBM solver

The Taylor-Green vortex (TGV) field which possess an exact solution to the Navier-Stokes equations is used as a test case in the present simulations. The regular velocity field dictated by the TGV initialization evolves into small scale turbulence and eventually dissipates depending on the viscosity. The results obtained from the present simulation are shown in figure 6. The initial velocity field u is depicted in frame (a) of figure 6. Frames (b) - (e) show the iso-surfaces of vorticity magnitude colored by velocity magnitude at the indicated times. As we can see how the regular velocity field in frame (a) evolves into small scale structures (spaghetti like structures) that are grouped together based on same value of vorticity ($\omega = \nabla \times \mathbf{u}$.).



(a) Initial (b) t = 1000 (c) t = 1500 (d) t = 2000 (e) t = 3000

Figure 6: Evolution of Taylor-Green vortex

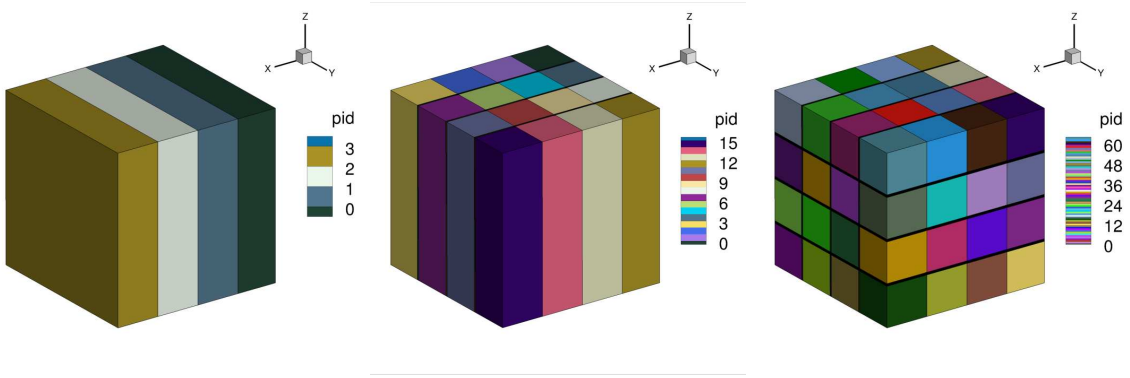
3.2 Parallel Algorithm & Performance

Here we describe the parallel aspects of the general algorithm presented in section 3.1.1.

- The initialization step: Each processor will fill initialize its own data with the lattice speeds, weights, and geometric and collision extents. This step also initializes the velocity according to the Taylor-Green vortex equation.
- The collision step: Each processor will word individually on their own data and will compute the collision terms shown on the right hand side of the equation 2.
- Application of periodic boundary conditions: Depending on the processor decomposition this step changes. For e.g. in a 3D processor decomposition the data will be exchanged in all principal directions and hence there is no need to explicitly assign the values for the periodic boundary condition. Where as in a 1D($x - direction$) decomposition this step needs to fill the periodic boundary values in the remaining two directions (y, z - directions)
- Exchange of data: In this depending on the type of decomposition we will exchange data in *xdir*, *ydir*, *zdir*. The data exchange involves communicating one layer of ghost cells for the 3D grid depending on the decomposition type and a total of 15 values needs to be exchanged per grid point, because we are using $D3Q15$ as the lattice here. New MPI types were defined based on a combination of VECTOR and CONTIGUOUS types and these newly committed types were used for data exchange directly without a need for additional buffers when exchanging data. This is the main communication step in the present algorithm.
- Streaming step: In this step, each processor will advect its own particle distribution function to its neighbors as described in section 3.1.1.
- Computation of primitive variables: In this step each processor will compute primitive variables (ρ, u, v, w) on its own data.
- File I/O: In this step all processors sends data to the host processor and it then writes the data to a single file so that this can be post processed. This step is not included in the present parallel timing computations.

3.2.1 Processor Decomposition

The processor decomposition is shown in figure 7. Three different kinds of decompositions have been implemented in the FlowLBM code as part of this project. The one-dimensional (slab decomposition), two-dimensional (pencil decomposition), and three-dimensional (box decomposition) are shown in figure 7(a)-(c) respectively. The colors indicate various processors in the figure 7. The number of processors are $4, 16$ and 64 in the decomposition shown in figures 7(a)-(c). Special care needs to be taken when the decompositions are performed in 2D and 3D. Because in those decompositions cross communication/diagonal communication shows up. To avoid this an existing strategy from the literature is used. Instead of sending the data to a diagonally opposite compute node directly, the same was achieved in two separate communications in 2D decomposition and three separate communications in the 3D decomposition.



(a) 1D (b) 2D (c) 3D

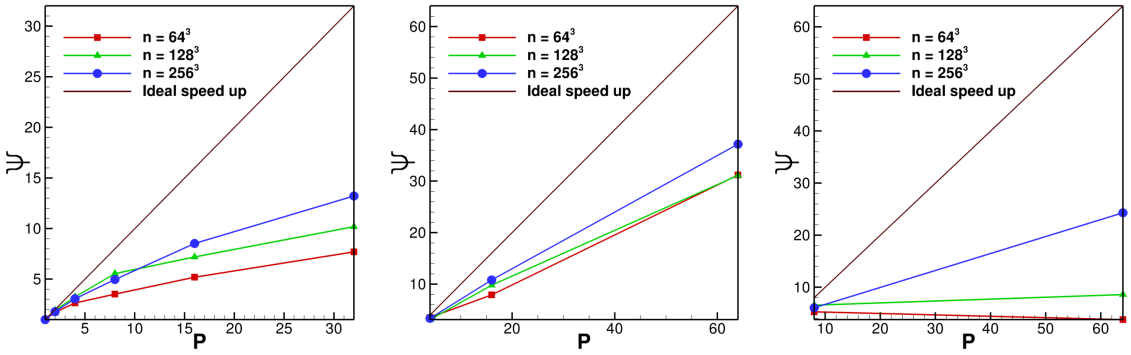
Figure 7: Schematic of processor decomposition

3.2.2 Speedup & Iso-efficiency

The speedup achieved (ψ) is shown plotted as a function of number of processors (p) for various problem sizes in Figure 8. Frames (a) - (c) of Figure 8 show the three different decompositions used, namely 1D, 2D, and 3D respectively as discussed earlier. Please note that the simulations in the 1D decomposition were only performed till 32 processors and hence, the plot in Frame (a) has a maximum value of 32 on both axes where as in the other Frames (b) and (c) the axes extend till 64. In all the decompositions, the speedup is increasing as the number of processors is increased, whereas the rate of increase of speedup ($\frac{\partial \psi}{\partial p}$) is decreasing as the number of processors is increased. This can be clearly seen in Frame (a) where the curves flatten out towards the end of the x-axis i.e. towards $p = 32$. In the $3D$ decomposition since there are only two sampling points, it is not possible to comment on the rate of decrease/increase of speedup with respect to number of processors. Among the three decompositions considered, the $2D$ decomposition seems to be giving better speedup for the problem. For example, for the highest problem size of 256^3 considered here, the speedup for 1D, 2D, and 3D decompositions is 13, 36, and 22 respectively. Although 3D decomposition is supposed to have a smaller surface/volume ratio, 2D decomposition is showing better speedup. This is because of lesser communication needed for 2D decomposition than for 3D decomposition.

Next we computed the efficiency for each of the cases considered and tabulated them in Tables 3 through 5. This gives an idea of how the efficiency of the program changes as the number of processors and the work size is increased, thus enabling the user to run at optimum efficiency at specified number of processors. From Table 3 we can see that the for $p = 4$ and a problem size of $n = 64$ in each direction we have an efficiency of 66. If we increase the number of processors to $p = 8$ and the problem size to $n = 256$ we can see that the efficiency decreases only slightly to 62. Similarly, we can see that the efficiency is close to 44 and 41 for problem sizes $(p = 8, n = 64)$, $(p = 16, n = 128)$, and $(p = 32, n = 256)$.

The $2D$ decomposition shows a similar iso-efficiency pattern. As we can see from Table 4, the efficiency is same almost constant for $(p = 16, n = 64)$ and for $(p = 64, n = 64, 128)$. Also a similar observation can be made for the pairs $(p = 16, n = 128)$ and $(p = 64, n = 256)$. lines for the 2D decomposition considered here. For $3D$ decomposition, the efficiency drops very quickly, even as the problem size is increased, as can be seen from Table 5.



(a) 1D (b) 2D (c) 3D

Figure 8: Parallel metrics for 1D, 2D, and 3D decompositions

	p=1	p=2	p=4	p=8	p=16	p=32
n=64	100	85.37	66.21	44.00	32.45	24.10
n=128	100	96.10	81.04	69.34	45.00	31.82
n=256	100	89.51	76.60	62.04	53.30	41.30

Table 3: Efficiency as a function of problem size (n) and number of processors(p) in 1D decomposition

From theoretical considerations, the serial computation time is of the order of n^3. With t_c denoting the time taken for communication of order n^2 elements with neighbouring processors, the communication time is of the order of $2t_c$, $4t_c$, and $8t_c$ in $1D, 2D$, and $3D$ decompositions respectively. Therefore, we have a serial time of $T_1 = n^3$, and a parallel time of $T_P = n^3/p + 2t_c$ for $1D$ decomposition. The overhead time, T_o is computed as

$$\begin{aligned} T_o &= pT_p - T_1 \\ &= p(n^3/p + 2t_c) - n^3 \end{aligned}$$

Hence, theoretically, the overhead time and work W should scale as p as the number of processors is increased. However, from the tabulated results obtained from the computations we see that in the $1D$ case doubling the number of processors increases the work size 8 times (scales as p^3), where as in the $2D$ decomposition case quadrupling the number of processors has increased the work size by 8 times to maintain the same efficiency. So, the results show that work should scale as $W = K \times p^3$ in the $1D$ decomposition case, where as it should scale as $W = K \times p^{3/2}$ in the $2D$ decomposition case. So, it seems that the $2D$ decomposition iso-efficiency curve seems to be better approximating the theoretical one.

	p=4	p=16	p=64
n=64	88.66	49.43	48.74
n=128	77.89	61.25	48.58
n=256	84.42	67.53	58.08

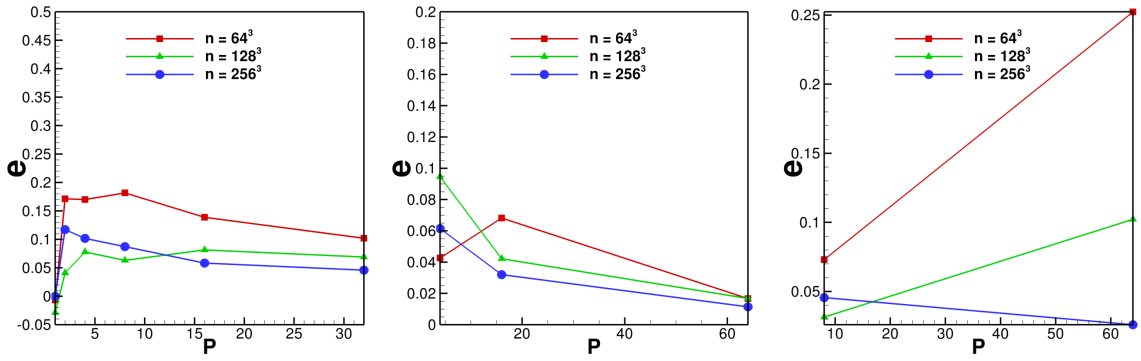
Table 4: Efficiency as a function of problem size (n) and number of processors(p) in 2D decomposition

	p=8	p=64
n=64	66.17	05.91
n=128	81.93	13.43
n=256	75.80	37.96

Table 5: Efficiency as a function of problem size (n) and number of processors(p) in 3D decomposition

3.2.3 Karp - Flatt metric

The Karp-Flatt metric was calculated for all the parallel configurations simulated and is shown plotted in figure 9. This metric is seen to initially in the $1D$ decomposition as shown in figure 9(a) and then decrease there after. A monotonic decrease is observed for the 128^3 and 256^3 cases as shown plotted in frame (b) where as an initial increase and decrease after wards trend is observed for the 64^3 case. For the $3D$ decomposition, the 256^3 case shows a decrease as the number of processors is increased.



(a) 1D (b) 2D (c) 3D

Figure 9: Karp - Flatt metrics for 1D, 2D, and 3D decompositions

4 Summary & Conclusions

Two numerical codes for computational fluid dynamics, WenoHydro and FlowLBM, have been parallelized and evaluated in this project. The codes have been parallelized using the domain decomposition framework on distributed memory systems, with MPI libraries used for message passing. Four versions of the WenoHydro code, differing based on the amount and patterns communication and computation involved, have been implemented and evaluated. Three kinds of processor grid arrangements have been studied for the FlowLBM code. Parallel performance studies carried out indicate that the definition of the serial time has a drastic impact on all metrics. The WenoHydro code shows very small speedup based on serial time, while it scales super-linearly when metrics are computed based on the time for single processor execution of the parallel code. In general, speedup increases with the number of processors, and efficiency reduces with number of processors. Studies also indicate a dependence on problem size and the layout of processor grids. Finally, iso-efficiency considerations show that for the WenoHydro code, work should scale as $P \log P$, while for the FlowLBM, work should scale as P. These theoretical predictions are reproduced by our numerical experiments for 2D decomposition.

References

[1] Shiyi Chen and Gary D Doolen. Lattice Boltzmann Method. (Kadanoff 1986), 1998.

[2] Li-shi Luo, Manfred Krafczyk, and Wei Shyy. Lattice Boltzmann Method for Computational Fluid Dynamics. *Fluid Dynamics*, (0):1–10, 2010.

[3] Dinesh A. Shetty, Travis Fisher, Aditya R. Chunekar, and Steven H. Frankel. High-order incompressible large-eddy simulation of fully inhomogeneous turbulent flows. *Journal of Computational Physics*, 229:8802–8822, 2010.

[4] F. X. Trias, M. Soria, C. D. Perez-Segerra, and A. Oliva. A direct fourier-schur decomposition for efficient solution of high-order poisson equations on loosely coupled parallel computers. *Numerical Linear Algebra Applications*, 13:303–326, 2006.

[5] Pieter Van Leemput, Martin Rheinländer, and Michael Junk. Smooth initialization of lattice Boltzmann schemes. *Computers & Mathematics with Applications*, 58(5):867–882, September 2009.