Arabian Journal for Science and Engineering (2020) 45:8981–8997
https://doi.org/10.1007/s13369-020-04640-1

RESEARCH ARTICLE-MECHANICAL ENGINEERING

Parallelization of Numerical Conjugate Heat Transfer Analysis in Parallel Plate Channel Using OpenMP

Asif Afzal[1] · **Zahid Ansari**[2] · **M. K. Ramis**[1]

Received: 2 July 2019 / Accepted: 14 May 2020 / Published online: 28 May 2020

Abstract
Conjugate heat transfer and fluid flow is a common phenomenon occurring in parallel plate channels. Finite volume method (FVM) formulation-based semi-implicit pressure linked equations algorithm is a common technique to solve the Navier–Stokes equation for fluid flow simulation in such phenomena, which is computationally expensive. In this article, an indigenous FVM code is developed for numerical analysis of conjugate heat transfer and fluid flow, considering different problems. The computational time spent by the code is found to be around 90% of total execution time in solving the pressure (P) correction equation. The remaining time is spent on U, V velocity, and temperature (T) functions, which use tri-diagonal matrix algorithm. To carry out the numerical analysis faster, the developed FVM code is parallelized using OpenMP paradigm. All the functions of the code (U, V, T, and P) are parallelized using OpenMP, and the parallel performance is analyzed for different fluid flow, grid size, and boundary conditions. Using nested and without nested OpenMP parallelization, analysis is done on different computing machines having different configurations. From the complete analysis, it is observed that flow Reynolds number (Re) has a significant impact on the sequential execution time of the FVM code but has a negligible role in effecting speedup and parallel efficiency. OpenMP parallelization of the FVM code provides a maximum speedup of up to 1.5 for considered conditions.

Keywords Parallelization · FVM code · OpenMP · Fluid flow · Speedup · Parallel efficiency

List of Symbols

Ar	Aspect ratio of battery cell
L	Length of battery cell
k	Thermal conductivity
l_o	Length of extra outlet fluid domain
l_i	Length of extra fluid domain
L_o	Dimensionless length of extra outlet fluid domain
L_i	Dimensionless length of extra inlet fluid domain
q'''	Volumetric heat generation
$\bar{q}$	Non-dimensional heat flux
$\bar{S}_q$	Dimensionless volumetric heat generation
Pr	Prandtl number
Re	Reynolds number
T	Temperature
T_o	Maximum allowable temperature of battery cell
$\bar{T}$	Non-dimensional temperature
u	Velocity along the axial direction
U	Non-dimensional velocity along the axial direction
u_∞	Free stream velocity
v	Velocity along the transverse direction
Q_r	Heat removed from surface (non-dimensional)
V	Non-dimensional velocity along the transverse direction
w	Half-width
$\bar{W}$	Non-dimensional width
x	Axial direction
X	Non-dimensional axial direction
y	Transverse direction
Y	Non-dimensional transverse direction

Greek Symbols

α	Thermal diffusivity of fluid
ν	Kinematic viscosity of fluid
ρ	Density of fluid

✉ Asif Afzal
asif.afzal86@gmail.com

✉ M. K. Ramis
ramismk@pace.edu.in

[1] Department of Mechanical Engineering, P. A. College of Engineering (Affiliated to Visvesvaraya Technological University, Belagavi), Mangaluru, India

[2] Department of Computer Science and Engineering, P. A. College of Engineering (Affiliated to Visvesvaraya Technological University, Belagavi), Mangaluru, India

ζ_{cc}	Conduction–convection parameter
μ	Dynamic viscosity

Subscripts

c	Center
f	Fluid domain
s	Solid domain (battery cell)
∞	Free stream
m	Mean

1 Introduction

Dealing with any physical problem by individual means of heat transfer is standard practice. But there are certain areas in which the given physical problem can be addressed by combining two modes of heat transfer, i.e., coupling conduction–convection or radiation mode, and such type of heat transfer process is called as conjugate heat transfer. This kind of heat transfer phenomena finds applications in practical and industrial uses. Consider the heat transfer analysis of electronic devices in which the conductions in the solid body are coupled with convection in the fluid body [1]. Few more areas to mention where this coupled heat transfer analysis is required include the heat extraction by coolants from nuclear fuel elements, heat exchangers, minichannels, and modern hybrid electric vehicles in which the lithium-ion (Li-ion) battery needs to be cooled, etc. On another hand, one of the most common idea to provide proper cooling and effective heat transfer to/from the surface of hot bodies is the use of a parallel plate channel. In parallel plate channels, the fluid is forced to flow past the parallelly placed hot solid bodies from which heat is extracted continuously. Hence, the phenomena of conjugate heat transfer in a parallel plate channel occur. The technique of parallelly placing fins/plates has a wide range of applications in innumerous areas, too large to mention [2].

The analysis of heat and fluid flow behavior in coupled heat transfer and parallel plate channels is commonly carried out using computational fluid dynamics (CFD) analysis. In CFD analysis, the prominent methods used for numerical prediction are finite difference method (FDM), finite volume method (FVM), or finite element method. The FDM and FVM methods are the most commonly used methods by scientists for conjugate analysis in parallel plate channels. However, FVM is comparatively more effectively used than FDM due to its several advantages in predictions of fluid flow behavior easily [3, 4]. Further, even though several methods are available to numerically obtain the solution of governing partial differential equations (PDEs) representing fluid flow conditions, but semi-implicit pressure linked equation (SIMPLE) algorithm is one of the most common employed algorithm developed by Patankar and Splanding [5]. One of the major issue associated with SIMPLE algorithm is that it is computationally expensive due to its slow convergence. The inner pressure corrections involved in this algorithm generally make the computations slow. Lot of research work is carried out to reduce the computational time of applications developed for numerical analysis using SIMPLE algorithm. Apart from developing modified versions of SIMPLE algorithm, parallelization of such applications/codes for fast numerical experimentations is also followed nowadays [6–9].

Recently, parallelization techniques of many CFD codes/software using different parallel computing tools are demonstrated for a particular application area and for in-house built codes. Gropp et al. [10] demonstrated parallelization of FUN3D code developed at NASA. FUN3D is developed for incompressible and compressible Navier–Stokes (NS) and Euler equations having a vertex-centered, tetrahedral unstructured grid. The subdomain parallelization of FUN3D code was successfully demonstrated using message passing interface (MPI) tool. Transient NS equation for incompressible flows in three dimensions for analysis of shear flows was solved by Passoni et al. [11] developing a code. Second-order finite difference central scheme in solid body, third-order procedure of Runge–Kutta in time along with explicit treatment of diffusive and convective terms, and for time marching fractional step method were used. The parallelization of the developed computational code was achieved using MPI applying their developed schemes (scheme A/B/C). 91% of parallel efficiency on doubling the processors and 60% parallel efficiency upon eightfold increase in processors was obtained.

Schulz et al. [12] used lattice Boltzmann method (LBM) for fluid flow analysis and established data structures to reduce the memory requirements. MPI parallelization for a grid size of 2×10^7 was used and achieved about 90% of parallel efficiency. Reproducing kernel particle method, also commonly known as mesh-free method, was used by Zhang et al. [13] to solve the NS equation for incompressible flow. MPI was used for domain decomposition-based parallelization to analyze three-dimensional flow over a cylinder, flow past a building. Seventy processors were used to obtain speedup of 35 [13]. Peigin and Epstein [14] used NES code, which primarily depends upon essentially non-oscillatory concept. MPI was used for multi-level parallelization of code NES for optimization. A cluster of 144 processors was used for implementing parallel execution, which provided around 95% parallel efficiency. Eyheramendy [15] used FEM analysis of lid-driven cavity problem developing a JAVA-based code. The cavity problem was solved for 50182 degrees of freedom (dofs) on a four-processor Compaq machine. For different Re, dofs, and number of threads the parallelization using multithreading feature of JAVA was accessed. By increasing the number of threads for different dofs, the parallel efficiency reduced. Jia and Sunden [16] used in-house developed CALC-MP three-dimensional multi-block code for solving NS and energy equation. Three different schemes 1/2/3 were developed for parallelization using different tools. The speedup obtained using three schemes was almost near on processors up to 8 and 16 in number.

Lehmkuhl et al. [17] presented the features of parallel unstructured CFD code ThermoFluids. This code is used for solving accurately and to have reliable results for industrial problems of turbulent flows. The parallelization of ThermoFluids was carried out using METIS software on a cluster of ten processors. Speedup of 8 is reported on maximum number of processors. Oktay et al. [18] used CFD code FAPeda to calculate aerodynamic pressure for a given angle of attack and speed on a wing shape. Using MUMPS library centered on multi-frontal approach, parallelization of FAPeda was achieved. The parallel speedup was worse using MUMPS; hence, the study was restricted to two-dimensional compressible flow. The CFD code GenIDLEST used for simulation of real-world problems was parallelized using OpenMP tool by Amritkar et al. [19]. This code is used for analysis of propulsion, biology-related problems to multi-physics flows. Further, Amritkar et al. [20] provided parallelization strategies using OpenMP for simulation of dense particulate system by discrete element method (DEM) with the same code GenIDLEST. OpenMP speedup was twice than the speedup of MPI on 25 cores.

Steijl and Barakos [21] applied quantum Fourier transform to solve Poisson equation to a vortex in cell method. MPI was used for parallelization of Poisson solver required for simulation of quantum circuits. Gorobets et al. [22] described parallelization of compressible NS equation for viscous turbulent fluid flow. OpenMP + MPI + OpenCL (open computing language) were used on a multi-level mode for parallelization across wide variety of hybrid architecture supercomputers. Compute unified device architecture (CUDA) tool was used by Lai et al. [23] to parallelize compressible NS equation on NVIDIAs GTX 1070 GPU. Two cases were demonstrated using the parallelized NS solver for flow over cylinder and double ellipsoid. CFD code ultraFluidX based on LBM was parallelized using CUDA by Niedermeier et al. [24]. Empty wind tunnel and wind tunnel with car were the test cases demonstrated by parallel execution of the code on multiple GPUs. The use of Green function for simulation analysis of complicated surfaces such as in modern ship design requires immense computational time. OpenMP was used to parallelize the Green function-based code applying 16 number of threads. For different discretization methods, the speedup up to 12 was achieved using 16 threads [25]. In the simulation analysis of bacterial biofilm model, fluid dynamics and solute simulation was together performed by Sheraton and Sloot [26]. For this two-model simulation analysis, FENICS software was used based on FEM. One to 16 processors were used for parallel performance analysis using METIS library. Wang et al. [27] developed in-house CFD code to solve NS equation for compressible viscous flow in three dimensions. For discretization in space, nonlinear weighted compact fourth-order FDM was used. Implicit/explicit scheme was used for time discretization, but the code is computationally expensive. Using MPI, OpenMP and offload programming model parallel execution on heterogeneous architecture of the supercomputer Tianhe-1A and Tianhe-2 was performed successfully.

It is a usual practice of CFD engineers to develop their own code for their specific application areas. The above literature works reported give an insight into parallelization of codes developed for simulation of fluid dynamics, design optimization of aerodynamics structure, compressible viscous flows, thermal and fluid flow analysis, etc. However, parallelization of SIMPLE algorithm-based code for conjugate heat transfer and fluid flow using OpenMP is not reported. In this work, an indigenous CFD code is developed for different applications, which includes fluid flow analysis over a flat plate, fins, or in a parallel plate channel considering conjugate condition at the interface of solid and fluid body. The analysis can be extended to prediction of either only thermal analysis or coupled/conjugate thermal and fluid analysis of plates with uniform or non-uniform heat generation. Conjugate heat transfer analysis of nuclear fuel elements, Li-ion battery cells, fins, etc., can be performed using this code. The in-house code solves NS equation in two dimensions considering incompressible flow in Cartesian coordinates. FVM formulation with staggered grid and SIMPLE algorithm to solve the NS equation is applied. But, due to pressure corrections required during each iteration, the computational cost is expensive. Hence, parallelization of the FVM code is carried out using OpenMP for different fluid flow conditions like internal flow, external flow, different Re, grid size, etc. Red and black successive over-relaxation (RBSOR) scheme is employed for parallelization of the pressure Poisson equation. As a demonstration, the thermal management problem of a Li-ion battery system is considered in this article. In the reminder of the article, numerical methodology, parallelization strategy, and obtained parallel performance on different computing machines are discussed.

2 Numerical Methodology

A battery module usually consists of battery cells that are densely packed to obtain higher power densities. For ease of operation and better thermal uniformity, the number of battery cells in each module is less. In this paper, a computationally efficient thermal model used for simulating the thermal behavior of modern electric vehicle battery cells generating uniform heat during charging and discharging operations at a steady state is simulated. A parallel channel with liquid coolant flow is employed to cool the battery cells during operation. The developed thermal model is then used to analyze the thermal behavior of the battery cell for various parameters, as shown in Fig. 1. Alongside, the computational domain is symmetrical along the vertical axis; therefore, to reduce the computational cost, only half of the domain through a flow passage, configuration is modeled. Figure 2 shows the simulated domain, which consists of two sub-domains, which are the Li-ion battery cells, a vertical parallel flow path channel and the coolant. The fluid flow inside the channel is commonly in laminar regime owing to the low velocity of the flow inside the channel.

The governing equation describing the heat transfer process when discharging/charging the Li-ion battery cell is given by

$$k_s \nabla^2 T + q''' = 0 \quad (1)$$

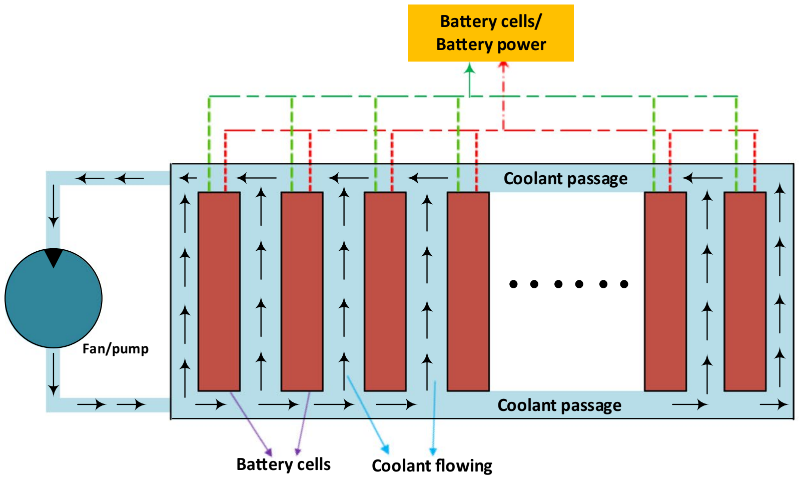
where q''' is volumetric heat generation term.

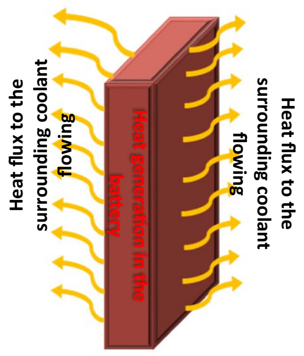
The governing equations for two-dimensional, steady, incompressible, laminar, forced convection flow in the fluid domain are continuity equation, x and y momentum equations and equation of energy, which are as follows:

$$\nabla u = 0 \quad (2)$$

$$(u\nabla u) = -\frac{1}{\rho}\nabla p + \mu \nabla^2 u \quad (3)$$

$$u\nabla T = \alpha \nabla^2 T \quad (4)$$



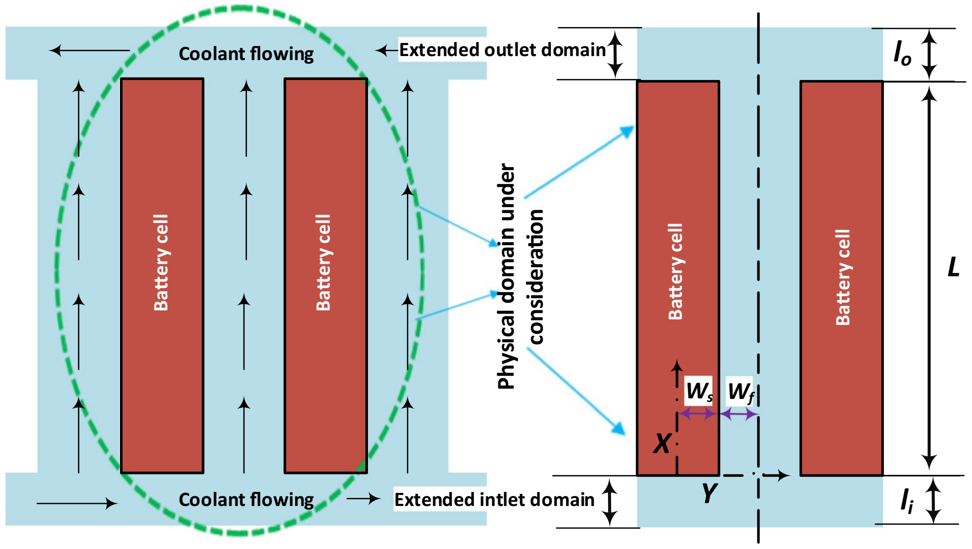

Heat flux to the surrounding coolant flowing
Heat generation in the battery
Heat flux to the surrounding coolant flowing

(a) Compact battery module with a stack of batteries and coolant passages

(b) Heat flux from the battery surface

Fig. 1 Schematic view of the arrangement of battery cells with air circulation fan and heat generation in batteries

Fig. 2 The symmetrical battery (prismatic cell) and coolant flow domain considered for computational analysis



The above equations are non-dimensionalized using the following set of normalizing parameters:

$$\bar{S}_q = \frac{q'''w_s^2}{k_s(T_o - T_\infty)}, C = 4\mathrm{Ar}^2, \bar{T} = \frac{T - T_\infty}{T_0 - T_\infty}, L_i = \frac{l_i}{L}, L_o = \frac{l_o}{L}$$
$$X = \frac{x}{L}, U = \frac{u}{u_\infty}, V = \frac{v}{u_\infty}, P = \frac{p}{\rho u_\infty^2}, A_r = \frac{L}{2w_s}$$
$$Y_s = \frac{y_s}{w_s}, Y_f = 1 + \frac{y_f}{L}, \overline{W}_f = \frac{w_f}{L}, \zeta_{cc} = \frac{k_f}{k_s}\left[\frac{w_s}{L}\right],$$
$$\mathrm{Re} = \frac{u_\infty L}{v}, \mathrm{Pr} = \frac{v}{\alpha} \tag{5}$$

The final set of non-dimensionalized governing equations turns out to be:

$$\frac{\partial^2 \bar{T}_s}{\partial X^2} + C\frac{\partial^2 \bar{T}_s}{\partial Y_s^2} + C\bar{S}_q = 0 \tag{6}$$

$$\nabla U = 0 \tag{7}$$

$$U\nabla U = -\nabla P + \frac{1}{\mathrm{Re}}\nabla^2 U \tag{8}$$

$$U\nabla \overline{T_f} = \frac{1}{\mathrm{Re\,Pr}}\nabla^2 T_f \tag{9}$$

2.1 Solution Strategy

The numerical solution of the conjugate problem consisting of energy and momentum equations is obtained by employing the staggered grid method of finite volume method (FVM). SIMPLE algorithm is used to solve the coupled momentum and continuity equation to obtain velocity and pressure components. Employing the tri-diagonal matrix algorithm (TDMA) for velocity and temperature equations, numerical solution is obtained. For pressure correction equation, successive over-relaxation (SOR) is used. The detailed numerical method, boundary conditions, solution strategy, and validation are discussed in [28], which are avoided here for the sake of briefness. The flowchart of the solution process is illustrated in Fig. 3, which is common for both internal and external flows. In the chart U^*, V^*, and P^* represent the velocity and pressure guessed values. The detailed explanation can be easily referred from work [5].

3 Parallelization Attempt Using OpenMP

Parallel architectures are in significant attention to offer immense computational power by utilizing multiple processing units. The progress in the growth of parallel processing is due to stagnation of central processing units (CPUs) clock speed. To benefit out of the present multi-core/processor architecture, the programs have to be developed for parallel execution [29–31]. In this research work, an effort is made to parallelize the developed FVM code for the present conjugate heat transfer problem. Parallelization of the in-house developed indigenous code written in C language is achieved on multi-cores (CPUs) of a single computing machine (CM). Parallel computing paradigm OpenMP (open multiprocessing) is employed for the parallelization of the FVM code. Parallelization of the FVM code is achieved using red and black successive over-relaxation (RBSOR) scheme.

For fluid flow conditions like internal flow, external flow, internal flow with outlet domain extended, and internal flow with inlet and outlet domain extended, computational speedup obtained is investigated in detail. Grid size of 42×82, 52×102, 62×122, and 72×142 for internal flow and grid size of 24×122 for inlet and outlet extended domain are adopted for parallel performance analysis. For external flow, the grid sizes chosen are 122×122, 162×162, 202×202, and 242×242 to understand the parallel speedup achieved. In the case of internal flow, the spacing between the parallel battery cells $\bar{W}_f = 0.1$ is kept constant. For both internal and external flows, $\mathrm{Re} = 250$, 750, 1250, and 1750 is considered. The other parameters are fixed to their base values for complete parallelization analysis. Parallel efficiency of the parallelized code is also investigated to understand the fraction of time for useful processor utilization.

3.1 The RBSOR Method

The computational time taken by the developed FVM code depending on parameters varies from approximately 30 min to 24 h. From the profile analysis of different functions used in the code, it is found that up to 91% of computational time is spent on pressure correction function. The remaining time is used by U and V velocity, temperature, and printing output results function. Hence, the major focus is made on parallelizing the pressure correction function using RBSOR scheme on CMs with different configurations. SOR method is employed for solving the pressure correction equation obtained using the SIMPLE algorithm technique. For the remaining functions, TDMA is used to solve the corresponding discretized equations. One of the commonly known scheme for parallelization of SOR is red and black SOR (RBSOR) scheme. In the following section, detailed description of working of RBSOR scheme is provided.

The SOR is an important iterative method to solve the system of linear equations. SOR is an expansion/improvement of Gauss–Seidel method that speedups convergence. It over-relaxes and combines the old values and current values by a factor greater than unity [32–35]. In this work, SOR is used to solve the pressure correction equation with

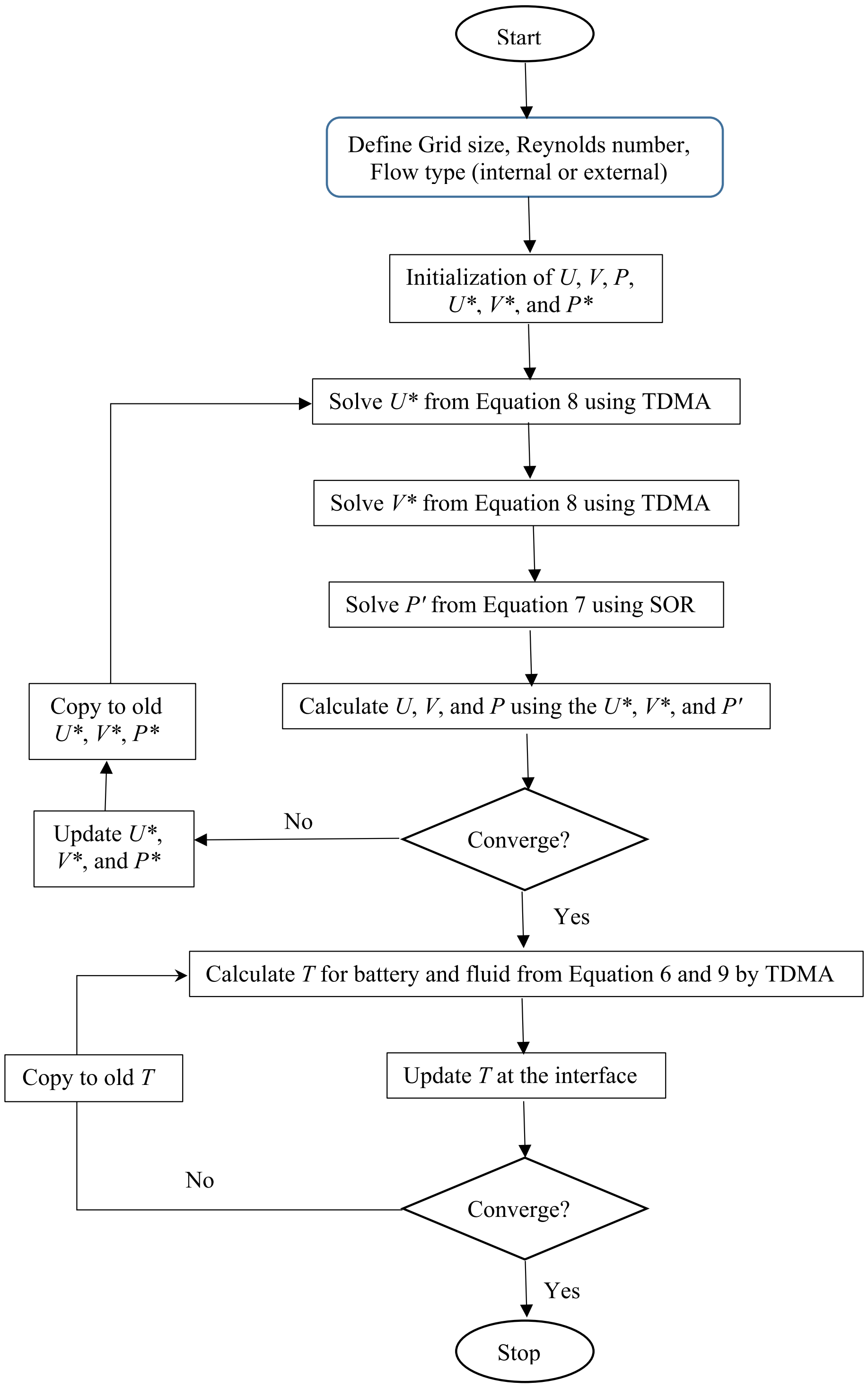

Start
Define Grid size, Reynolds number, Flow type (internal or external)
Initialization of U, V, P, U*, V*, and P*
Solve U* from Equation 8 using TDMA
Solve V* from Equation 8 using TDMA
Solve P′ from Equation 7 using SOR
Calculate U, V, and P using the U*, V*, and P′
Converge?
No
Update U*, V*, and P*
Copy to old U*, V*, P*
Yes
Calculate T for battery and fluid from Equation 6 and 9 by TDMA
Update T at the interface
Converge?
No
Copy to old T
Yes
Stop

Fig. 3 Flowchart of SIMPLE algorithm-based solution process

over-relaxation factor equal to 1.8. This pressure correction function consumes maximum computational time as mentioned earlier, due to inner iterations required for correcting the pressure.

The parallel implementation of SOR technique is not easy as it uses the values of neighboring cells/grid points of the current iteration as shown in Fig. 4. The gird point/cell shown in yellow color (number 15) requires the values of upper, lower, left, and right side cells (numbers 5, 25, 14, and 16, respectively) shown in blue color. Each grid point is serially executed one after the other taking the newly calculated values of neighboring points. In Fig. 4, this concept is shown serially from 1 to 100. Hence, this brings in sequential dependency and may lead to different results in parallelly

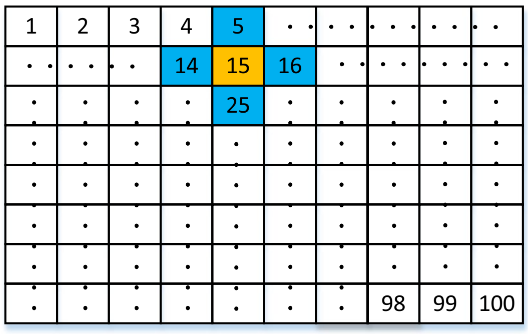


Fig. 4 Serial execution of grid points depending upon the four neighboring cells

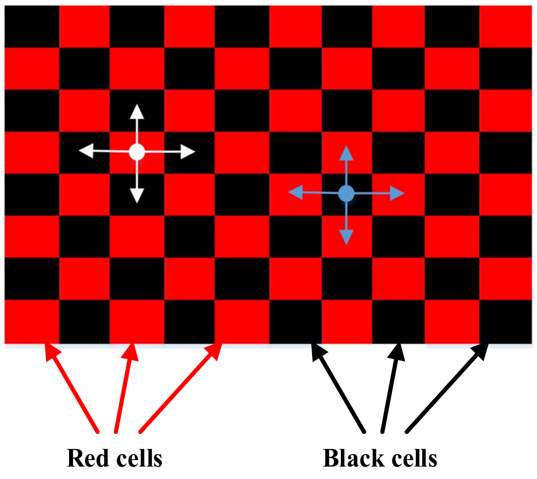


Fig. 5 RBSOR scheme for parallel implementation of SOR

executed SOR. To overcome this sequential dependency and parallelize the SOR algorithm, graph coloring methods are used. Using this coloring method, the single sweep of SOR can be broken into multiple sweeps which are suitable for parallel processing. The RBSOR scheme can be thought of as a compromise among Gauss–Seidel and Jacobi iteration. As shown in Fig. 5, the RBSOR solves by coloring in the checkerboard with alternative red/black grids. At first, all the red cells are computed simultaneously considering the neighboring black points. Then, black cells are computed using the updated red cells parallelly. The RBSOR scheme implementation is mentioned briefly in Algorithm 1.

3.2 OpenMP Paradigm

OpenMP is an application program interface (API) that can be used for explicitly writing multithreaded shared memory parallel applications. OpenMP primarily consists of a set of API components like compiler directives, environment variables, and runtime library routines. OpenMP is suitable for shared memory multi-core/processor in which the parallelism is achieved with the use of execution threads which exists within the source of a sole process [36–38]. The smallest unit scheduled by an operating system for processing is commonly known as an execution thread. OpenMP parallelization can involve insertion of simple compiler directives in a serial program or complex subroutines to set locks, nested locks, nested parallelism, and even multiple level parallelism. OpenMP parallelization works on a fork-and-join parallel execution model as shown in Fig. 6. The master thread starts as a single process and keeps executing serially (sequentially) until it comes across the parallel construct region. Then, the master thread creates a team of threads (fork) as required to simultaneously process the statements within the parallel region. Once the newly created threads (slave threads) execute parallelly and complete the parallel region, they synchronize/terminate (join), leaving solely the master thread to continue. This kind of parallelism is famously known as fork-and-join parallel model. The slave

Algorithm 1: Pseudo code of RBSOR method

```
1. for $i_{min} = 1$ to $m$ do                          // iterations for red cells
2.    for $j_{min} = 2 - ( i \% 2 )$ to $n$ do          // inner iterations
3.       $P^{(k+1)}_{i,j} = ( 1 - w ) * P^{(k)}_{i,j} + N^{(k+1)}_{i,j}$    // $N^{(k+1)}_{i,j}$ is sum of neighbouring points
4.       $j = j + 2$
5.    end for
6. end for
7. for $i_{min} = 1$ to $m$ do                          // iterations for black cells
8.    for $j_{min} = 1 + ( i \% 2 )$ to $n$ do          // inner iterations
9.       $P^{(k+1)}_{i,j} = ( 1 - w ) * P^{(k)}_{i,j} + N^{(k+1)}_{i,j}$    // $N^{(k+1)}_{i,j}$ is sum of neighbouring points
10.      $j = j + 2$
11.   end for
12. end for
```

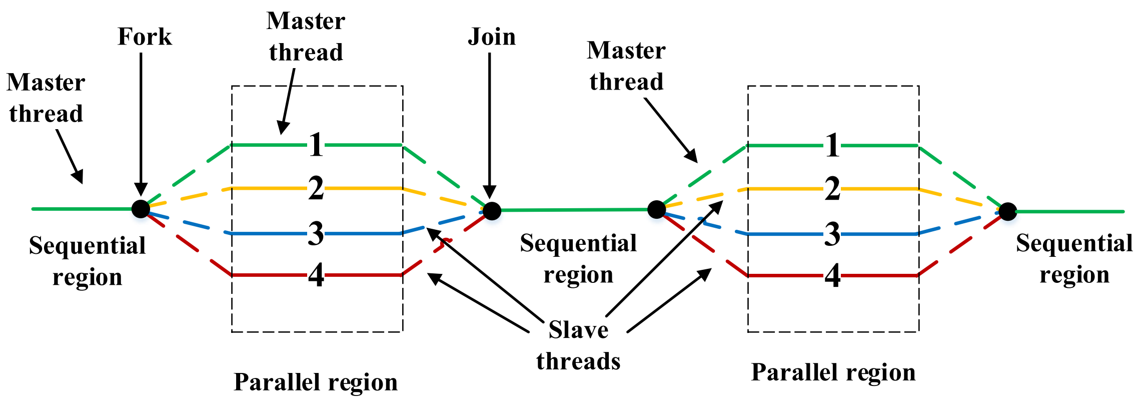


Fig. 6 Fork-and-join model of OpenMP parallel programming

threads can be made to wait at a point until all the threads have reached a common point in the parallel region using proper barriers.

As the OpenMP paradigm works on a shared memory model, all the threads have access by default to global memory. The slave threads communicate by writing and/or reading to global memory. Their simultaneous update to global/shared memory may result in a race condition which again changes with the scheduling of threads [39, 40]. A data race condition occurs when two or more threads access the same memory without proper synchronization. OpenMP also allows parallelization of parallel regions, i.e., it permits placing of parallel loops inside parallel loops. In such cases, the created slave threads further spawn to form again a team of threads, as shown in Fig. 7. Once the slave threads are distributed on different processors, the thread id will remain public. Hence to avoid this issue, the private clause can be used to keep the thread id specific to the processor. This enables to parallelly process instructions on different processors for a specific range of iterations easily. The OpenMP model during the runtime provides, to keep the threads static or to change the number of threads dynamically. The OpenMP parallel constructs placed with some clauses are the compiler directives. The inner `for()` loops of U, V, and T functions are parallelized using the appropriate omp parallel constructs, and the SOR is parallelized using the above-mentioned RBSOR scheme. The algorithm of the FVM code parallelized as discussed above is mentioned in Algorithm 2. Four CMs (computing machines) with different configurations

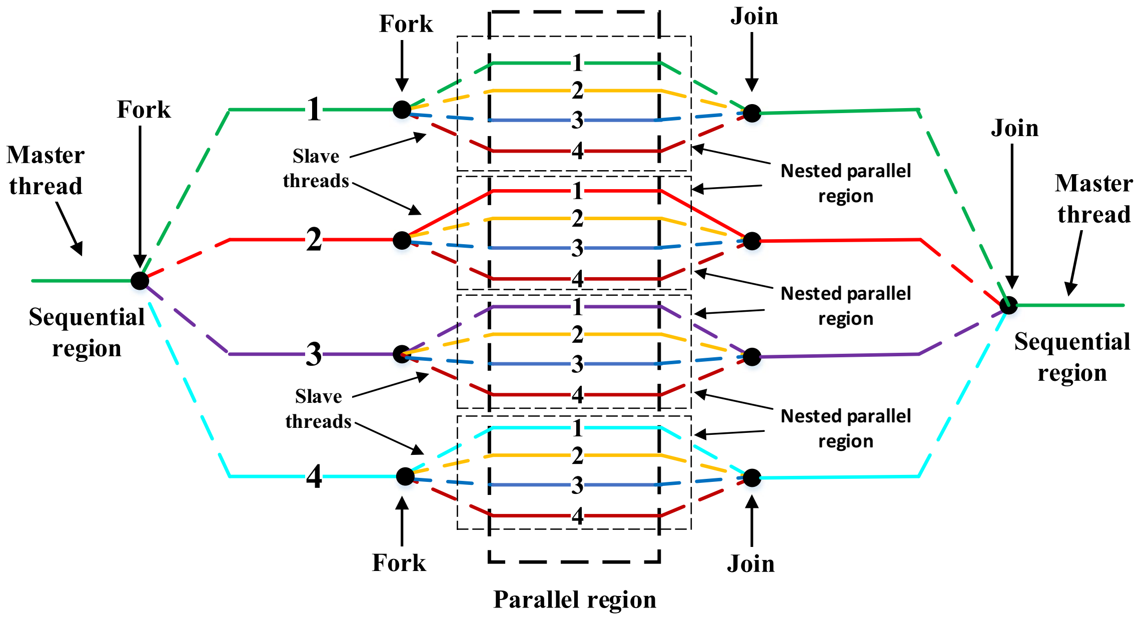


Fig. 7 Nested parallel region inside a parallel region using OpenMP

Table 1 Specifications of CMs used for OpenMP parallelization

CM number	Processor	Frequency (GHz)	RAM (GB)	Number of logical cores	Memory (TB)	Operating system
CM1	Intel® core ™ i7-4970	3.6	16	8	1.3	64-bit Windows 10
CM2	Intel® core ™ i7-4970	3.6	8	8	0.7	64-bit Windows 10
CM3	Intel® core ™ i5-3470 s	2.9	8	4	0.5	64-bit Windows 7 Professional
CM4	Intel® core ™ i3-3240	3.4	4	4	0.5	64-bit Windows 7 Professional

are used for OpenMP parallelization performance analysis. The specifications of these four CMs are mentioned in Table 1. CM1 is selected as the common machine for parallel computational analysis and to compare with different approaches.

Algorithm 2: Algorithm of OpenMP parallelized FVM code

```
1. Input                  // parametric values and options
2. Initialization         // initializing the variables
3. for $It$ = 1 to $It_{max}$ do
4.     update boundary grid points
5.     compute $U$ velocity          // solved using TDMA
6.         omp parallel construct for inner for loop
7.     compute $V$ velocity          // solved using TDMA
8.         omp parallel construct for inner for loop
9.     compute $P_{prime}$ if RBSOR is opted // solved using SOR
10.        omp parallel construct
11.            red cells
12.        omp parallel construct
13.            black cells
14.    compute $T$ temperature of battery cell // TDMA
15.        omp parallel construct for inner for loop
16.    compute $T$ temperature of fluid domain // TDMA
17.        omp parallel construct for inner for loop
18. end for
19. print $U, V, P$ and $T$ values
```

3.3 Nested Loop and Without Nested Loop Parallelization

OpenMP parallelization of the developed FVM code is also tried using both the `for()` loops (nested) parallelized and with only the outer `for()` loop parallelized (without nested), as shown in Figs. 6 and 7. This attempt to analyze the parallel performance of the code with nested loop parallelized and without nested loop parallelized is provided separately. In parallelization of a nested loop, i.e., of both the outer and inner `for()` loops, hyperthreading occurs among the logical cores of the physical processor. When parallelization of only outer `for()` loop, i.e., without nested, is executed, then the computations of complete cells/grids along a particular indices of outer loop occur on different cores. This kind of parallel execution refers to simple domain decomposition technique. As there are a limited number of cores used (maximum eight cores), repeated execution occurs on cores for processing both nested and without nested parallelism. In without nested parallelism, the inner `for()` loop computations are performed serially.

3.4 Speedup and Parallel Efficiency

Use of multiple processors to work together simultaneously on a common task is commonly known as parallel computing. The performance of a parallel algorithm implemented on a parallel architecture for parallel computations is measured by speedup and parallel efficiency. The ratio of time taken to execute the sequential algorithm on a single processor to the time taken by the parallel algorithm to execute on multiple processor is known as speedup. Parallel efficiency is defined as the ratio of parallel speedup achieved to the number of processers. Parallel efficiency gives the measure of the fraction of computational time at which a processor is used efficiently [6, 41–43].

According to the definition of speedup and parallel efficiency, they are calculated as given by Eq. 10 [41, 42]:

$$S_{(\text{par})} = \frac{T_{(\text{seq})}}{T_{(\text{par})}} \quad E_{(\text{par})} = \frac{S_{(\text{par})}}{N_{(\text{par})}} \tag{10}$$

where $S_{(\text{par})}$ is the parallel speedup achieved, $T_{(\text{seq})}$ is the elapsed (wall) time taken by the sequential program, $T_{(\text{par})}$ is the wall time taken by the parallel program for execution, $E_{(\text{par})}$ is the parallel efficiency and $N_{(\text{par})}$ is the number of processors employed for parallel execution. The efficiency of loss occurred due to communication of data, computational task partitioning, processors scheduling, management of data, etc., during parallel implementation is accounted by parallel efficiency. The elapsed time for computations in parallel on N processors can be written as composed of [44]:

$$T_{(\text{par})} = T_{(\text{seq})N} + T_{(N)} + T_{(\text{comm})N} + T_{(\text{misc})N} \tag{11}$$

where $T_{(\mathrm{seq})N}$ is the time taken by CPU for computations of sequential part of the program. $T_{(N)}$ is the time taken by CPU for computations of parallel part of the program on N processors, $T_{(\mathrm{comm})N}$ is the time taken by CPU for communication with N processors, and $T_{(\mathrm{misc})N}$ is the idle time or extra time spent induced due to parallelization of program.

4 Results and Discussion of Speedup and Parallel Efficiency Using OpenMP

Using OpenMP parallel computing paradigm, the parallelization of FVM code developed in-house is analyzed in the form of parallel speedup and efficiency. Four different computing machines (CMs), namely CM1, CM2, CM3, and CM4, are employed having different configurations mentioned in Table 1 to implement OpenMP parallelization. In this section, the speedup and parallel efficiency of the parallelized FVM code are provided in detail. Using RBSOR scheme, various grid sizes, and Re for internal and external flow, the computational time analysis is carried out. In this entire parallel performance analysis, the operating heat and fluid flow parameters are kept fixed at $\bar{S_q} = 0.5$, $\zeta_{\mathrm{cc}} = 0.06$, and Ar = 10, for both the flow conditions. For computational time analysis, the grid sizes chosen are 42 × 82, 52 × 102, 6 × 122, and 72 × 142 for internal flow. The grid size considered for outlet and inlet domain extended is 24 × 122. In case of external flow, 122 × 122, 162 × 162, 202 × 202, and 242 × 242 grid sizes are chosen, while Re is varied from 250 to 1750.

4.1 Elapsed Time

The elapsed time of the FVM code for internal and external flow on all the four machines is noted at first to get an idea of computational cost for different conditions. In Fig. 8, the elapsed time on CM1, CM2, CM3, and CM4 is shown for internal and external flow. The flow Re = 250 to 1750 and grid size of 62 × 122 and 202 × 202 for internal and external flow, respectively, are fixed to note the elapsed time. The elapsed time is higher for CM1 and other CMs at Re = 250 and keeps on reducing for both the flow conditions compared to other Re. At low Re = 250, the boundary layer thickness grows wider, which needs more pressure correction in the numerical analysis causing more computational time, while at higher Re = 1750, the boundary layer stays close to surface, and hence, less correction is required leading to reduced computational time. However, the time on CM4 is highest compared to other CMs at all Re. CM4 has the lowest clock speed and lower RAM, which causes the execution of the code to be slower among other CMs. CM1 and CM2 have a very similar configuration, which causes them to behave similar in the execution of code on them. Hence, time for CM1 and CM2 is almost same for all flow Re. This elapsed time is for outlet and inlet domains not extended during internal flow analysis. Surely, if higher grid size and extra flow domains are considered, the computational time increases up to 24 h and above.

4.2 Speedup Achieved

The speedup obtained on four machines CM1, CM2, CM3, and CM4 using nested OpenMP parallelization for internal flow applying RBSOR scheme is shown in Fig. 9. The speedup is shown for different grid sizes and Re without considering any extended domain. From Fig. 9a–d, it can be noted that with an increase in grid size at all, Re considered the speedup increases on all CMs. The main cause in increase in speedup with the increase in grid size is increased parallel computation on multiple processors. Parallel computation required on multiple processors will be less if the number of grids are lesser. Hence, the ratio

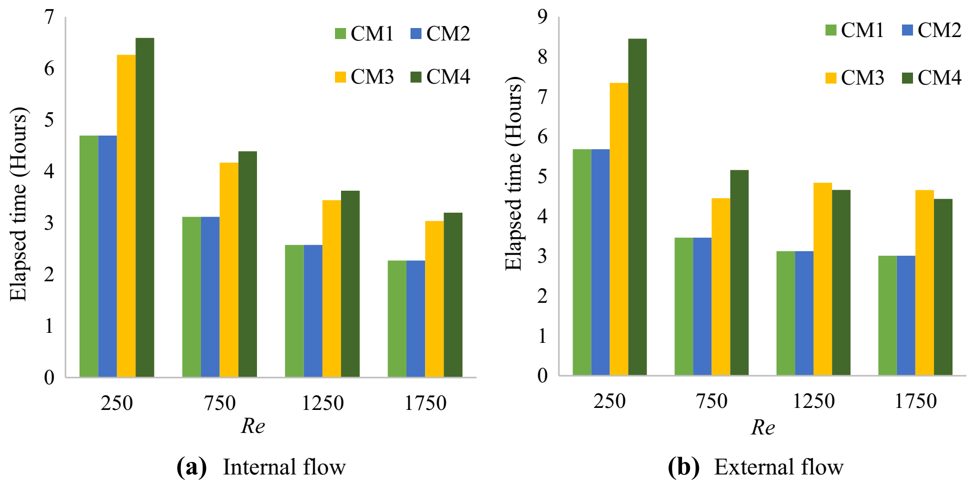


Fig. 8 Elapsed time on different machines for different flow conditions

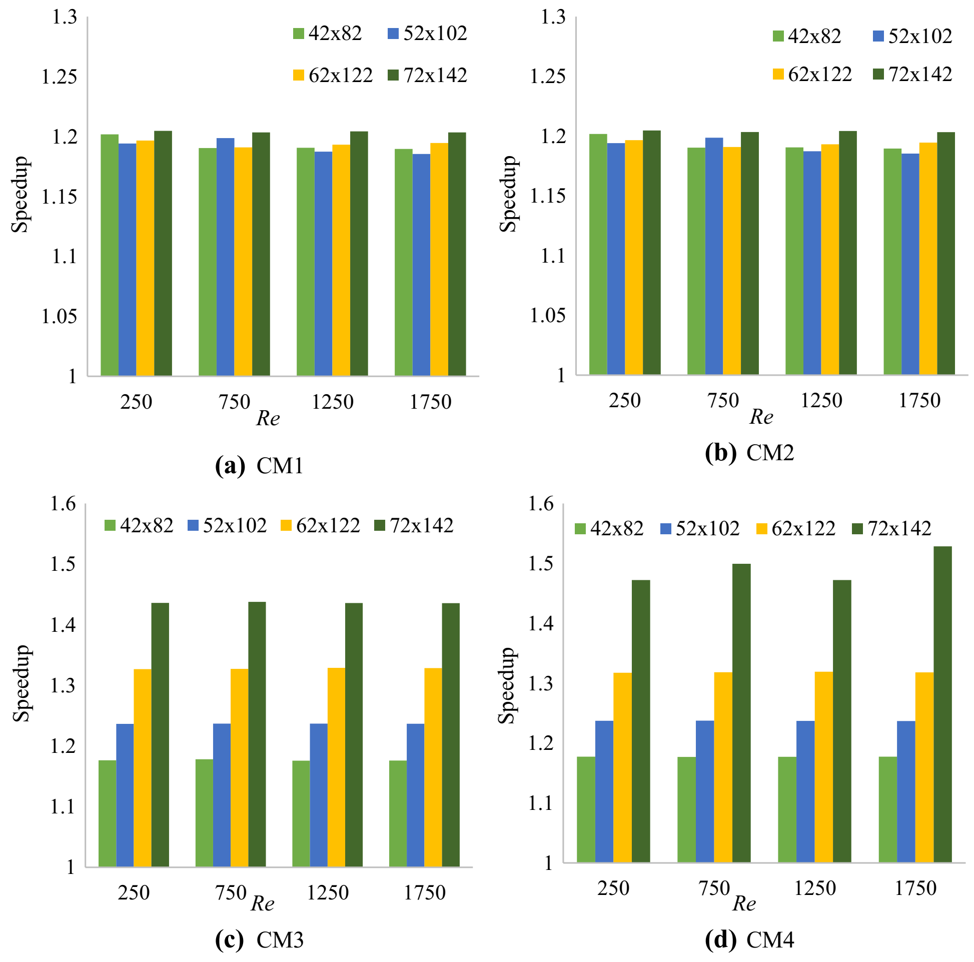


Fig. 9 Speedup obtained for internal flow without extended domain with nested OpenMP

of parallel computations compared to serial computations increases with increase in grid numbers.

It is easily understood from Fig. 9 that compared to CM1 and CM2, the speedup achieved on CM3 and CM4 is higher for different flow Re. As the clock speed and number of processors are more on CM1 and CM2 compared to CM3 and CM4, the serial computations are itself faster as shown previously in Fig. 8. On CM1 and CM2, little extra time is consumed on forking, joining, accessing/writing of data, and synchronizing of salve threads on eight processors compared to four processors on CM3 and CM4. Hence, the serial and parallel computational time ratio remains lower on CM1 and CM2. To achieve better speedup on CM1 and CM2, much more grid sizes are required. However, the speedup on CM1 and CM2 is very close to each other due to their similar configuration except RAM and hard disk capacity. Another observation that can be made is the effect of Re on speedup. With an increase in Re from 250 to 1750, there is a slight improvement in speedup on all CMs, which is clearly noticeable from Fig. 9d. The speedup achieved at Re = 250 is nearly the same all CMs at 42 × 82, and on CM3 and CM4, significant improvement is observed and is achieved with an increase in grid size.

The speedup achieved on different CMs and Re for internal flow with extended domains using nested OpenMP parallelization is shown in Fig. 10. The grid size for this analysis is fixed at 62 × 122, and for outlet and inlet domains, grid size is 24 × 122 each. When only the outlet domain is extended with extra 24 × 122 grids, the speedup achieved is shown in Fig. 10a. For both outlet and inlet domain extended, the speedup is shown in Fig. 10b. The speedup obtained for both the cases is similar at all Re on CM1 and CM2. On CM3 and CM4, the speedups are quite more at all Re compared to CM1 and CM2. Speedups are slightly increased for both the domains extended due to increased number of grid points compared to only outlet domain extended. Speedup on CM1 and CM2 in this case is same as speedup obtained for without extended domain shown previously in Fig. 9.

The speedup obtained for external flow with grid size 122 × 122, 162 × 162, 202 × 202, and 242 × 242 on different CMs is shown in Fig. 11. With increase in grid size, the speedup obtained increased on all machines at all

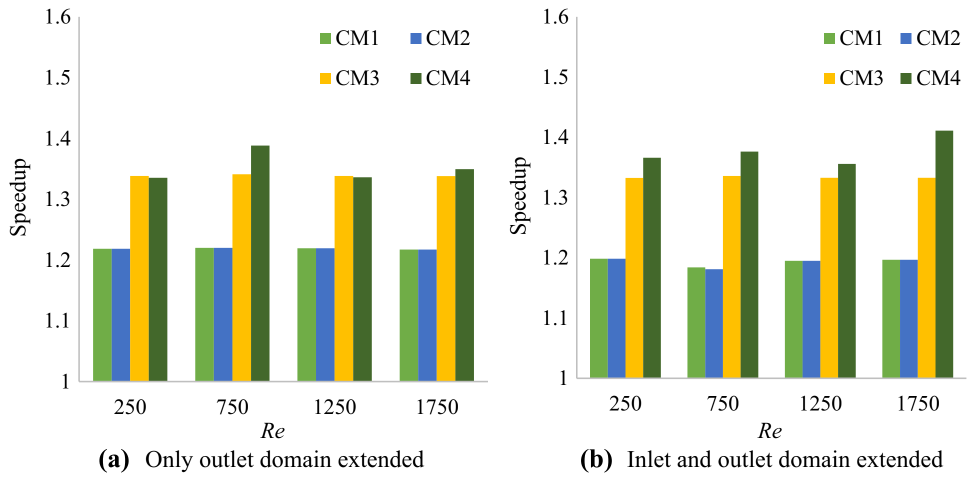


Fig. 10 Speedup for inlet and exit domain extended in internal flow with nested OpenMP

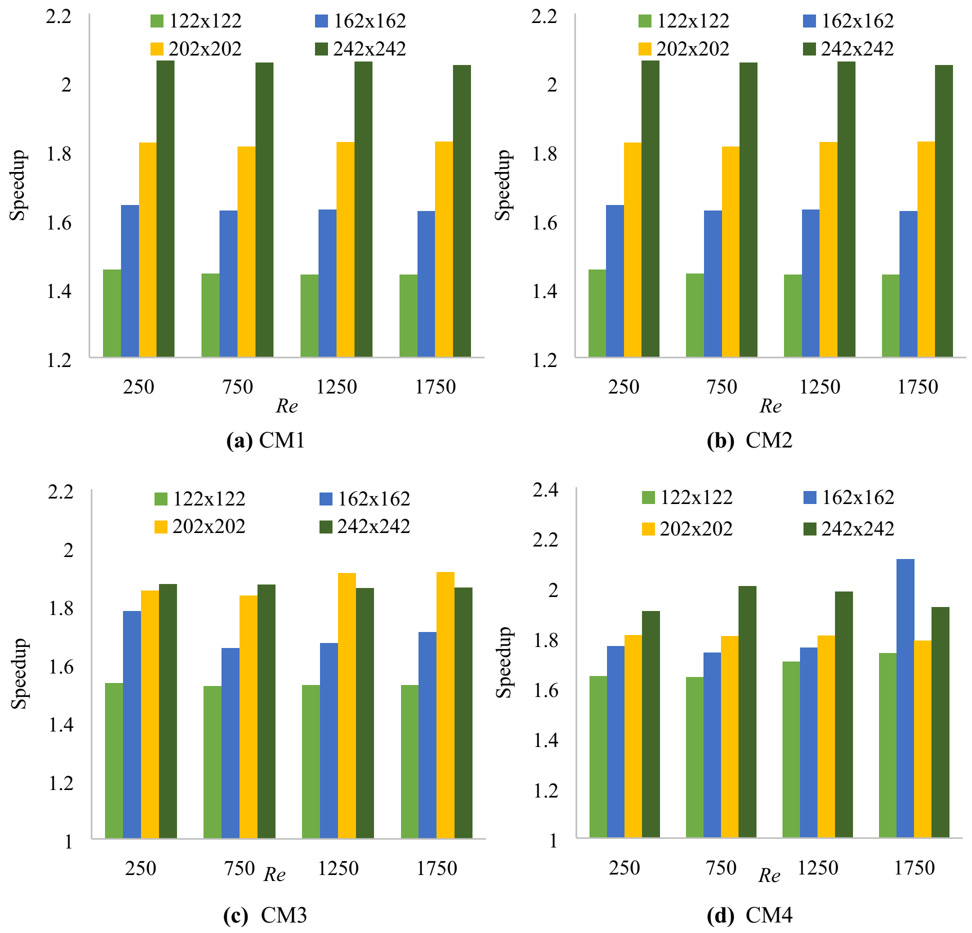


Fig. 11 Speedup obtained on different machines for external flow with nested OpenMP

flow Re. The speedups are significant on CM1 and CM2 compared to CM3 and CM4 due to increased number of grid size. This is in agreement with preceding discussion made on increasing the speedup by increasing the grid size for CM1 and CM2. With nested OpenMP parallelization, the outer `for()` loop indices are forked on different processors and further the newly created threads are again forked on each processor for inner `for()` loop which is generally termed as multithreading. However, this speedup is same at all Re for CM1 and CM2, whereas on CM3 and CM4, the speedups are not same.

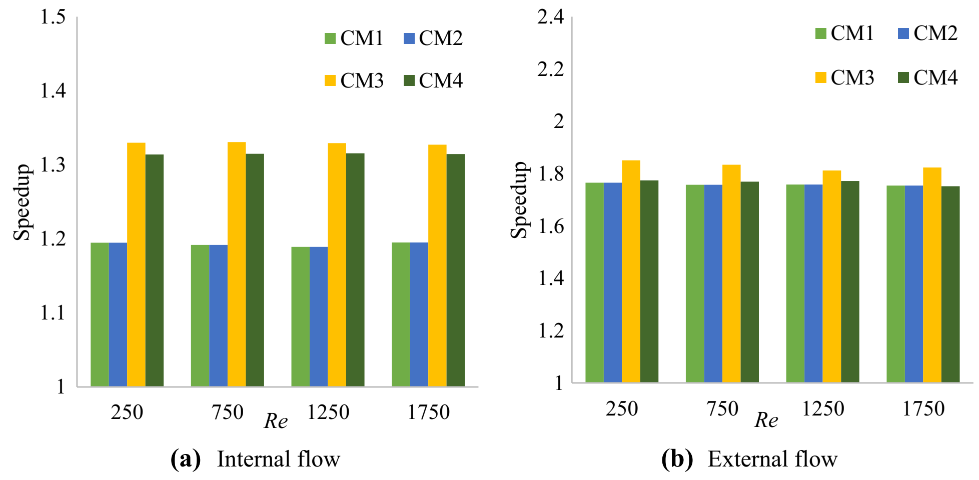


Fig. 12 Speedup for internal and external flow without nested OpenMP

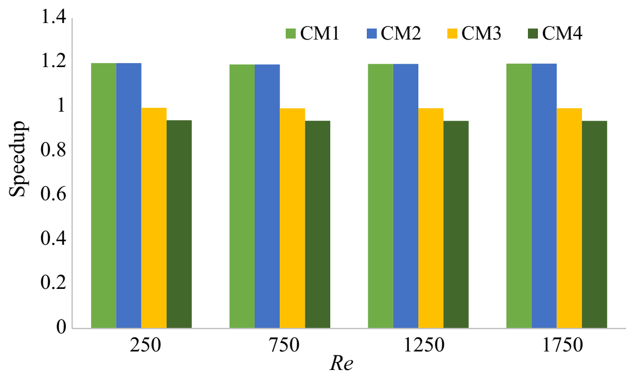


Fig. 13 Speedups with respect to serial time of CM1 for internal flow

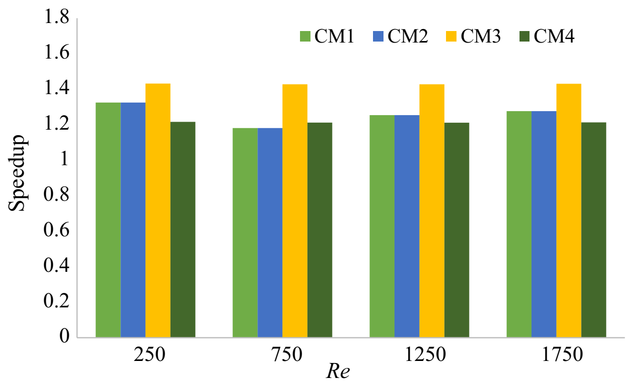


Fig. 14 Speedups with respect to serial time of CM1 considering external flow

Parallel performance speedup of the FVM code using without nested OpenMP `for()` loop in case of internal and external flow is depicted in Fig. 12. The speedup achieved on all the CMs is shown at different Re with fixed grid size of 62×122 and 202×202 for internal and external flow, respectively, without any extended domain. In without nested OpenMP parallelization, the outer `for()` loop indices are spawned on multiprocessors and then the inner `for()` loop indices are executed serially on every processor. Here, the time spent for further threading, joining, and synchronizing for inner `for()` loop is avoided. Hence, the speedups obtained in both the internal and external flow problem on all CMs are very similar to speedups reported earlier in case of with nested OpenMP parallelization. For higher grid size and then present consideration, the speedup will surely fall compared to nested OpenMP parallelization.

4.3 Speedup by Different Comparison

In Fig. 13, speedup obtained on different CMs considering the ratio of parallel execution on a CM and serial execution time of CM1 in all cases is depicted. Different Re and fixed grid size of 62×122 without any extended domain for internal flow and applying RBSOR scheme with nested OpenMP parallelization are implemented. It is quite obvious that the speedup on CM1 and CM2 will remain unchanged as the serial execution and parallel execution time on both the CMs is very similar due to their similar configuration. However, it is worth noticing that speedup on CM3 and CM4 is reduced when serial execution time of CM1 is considered. It can be seen that the speedup on CM3 is very close to but less than unity. This means that the parallel execution time on CM3 and serial execution time on CM1/CM2 are almost equal. If further look at Fig. 13 is focused, it will be seen that the speedup of CM4 is worse than the serial execution on CM1/CM2. It is appropriate to mention that for internal flow with 62×122 grid size, it is unimportant to parallelize the FVM code on CMs with less clock speed compared to clock speed of CM1 and CM2.

Speedup on different CMs considering the sequential execution time of CM1 for external flow is shown in Fig. 14. Re is changed from 250 to 1750, and RBSOR scheme is applied for the external flow using nested OpenMP parallelization.

It can be seen that the speedup on all the CMs considering serial time of CM1 is better in this case compared to internal flow. The reason again is the use of higher grid size with which the parallel computations become effective due to better usage of multiprocessors. Speedups for internal and external flow are completely contradictory. One thing for sure is understanding the use of proper CM with suitable configuration for a given fluid flow problem and grid size. While in internal flow problem the grid size was less, the speedup on CM3 and CM4 deteriorated. And now in case of external flow, the speedup on CM3 and CM4 is better than the speedup on CM1 and CM2 due to increased grid size. Hence, increased grid size leads to increased utilization of multiprocessors for longer duration resulting in improved speedup. Again, CMs with lower frequency consume more time than CMs with higher frequency. Therefore, the speedup will also be more on low-frequency CMs as the multiprocessors are used for more time than those of higher-frequency CMs. Nevertheless, increased speedup on low-frequency CMs need not necessarily mean providing faster results than that of higher-frequency CMs.

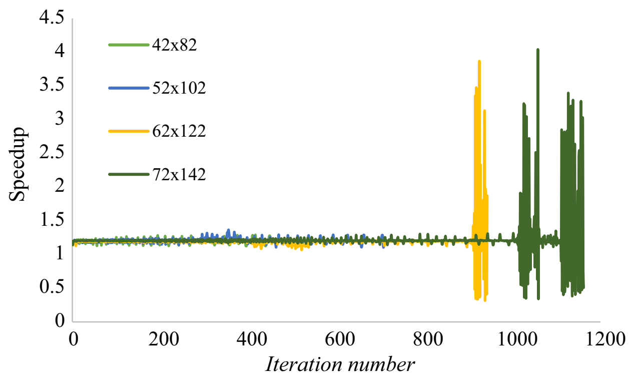


Fig. 15 Speedup obtained for each iteration for different grid sizes during internal flow

In Fig. 15, the speedup obtained on CM1 during each iteration of the FVM code during internal flow for different grid sizes and Re = 750 is shown. It can be seen that the speedup for most of the iterations is slightly more than unity. During the last few iterations at which the code is near about convergence, the speedup drastically fluctuates above and below the mean speedup of 1.2. Such kind of behavior during the final iterations needs to be investigated in detail as such the nature is not yet known and neither reported anywhere. However, there is an immense speedup before the convergence of iterations. For all grid sizes, the nature of speedup achieved during each iteration looks very similar.

4.4 Parallel Efficiency of the FVM Code

Parallel efficiency gives an idea of time utilized by the multiprocessors to perform computations parallelly compared to time used by a single processor for serial execution. The parallel efficiency of the FVM code using OpenMP and RBSOR scheme on different CMs for different conditions is presented in detail. During internal flow with different Re and grid sizes, the parallel efficiency on all CMs is shown in Fig. 16. The parallel efficiency of the serial program will be always unity as only one processor is used for computations. The parallel efficiency on CM1 and CM2 and on CM3 and CM4 is the same (Fig. 16a) for different Re, while the grid size is fixed at 62 × 122. This is due to the same number of processors on CM1 and CM2 and on CM3 and CM4. Additionally, for different Re, the respective serial and parallel computational time ratio is the same as explained with respect to Fig. 8; hence, the parallel efficiency on all CMs remains the same. However, the parallel efficiency on CM3 and CM4 is better than CM1 and CM2 due to reduced number of processors and increased speedups obtained. On another hand, increasing grid size at a fixed Re = 750 results in improving parallel efficiency on CM3 and CM4, as shown in Fig. 16b. Again, this behavior is due to increasing speedups with reduced processors on CM3 and CM4 compared to

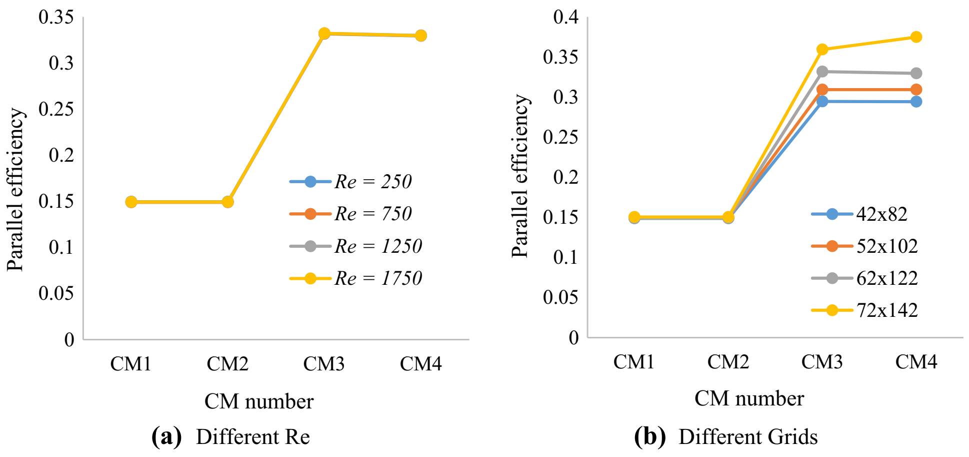
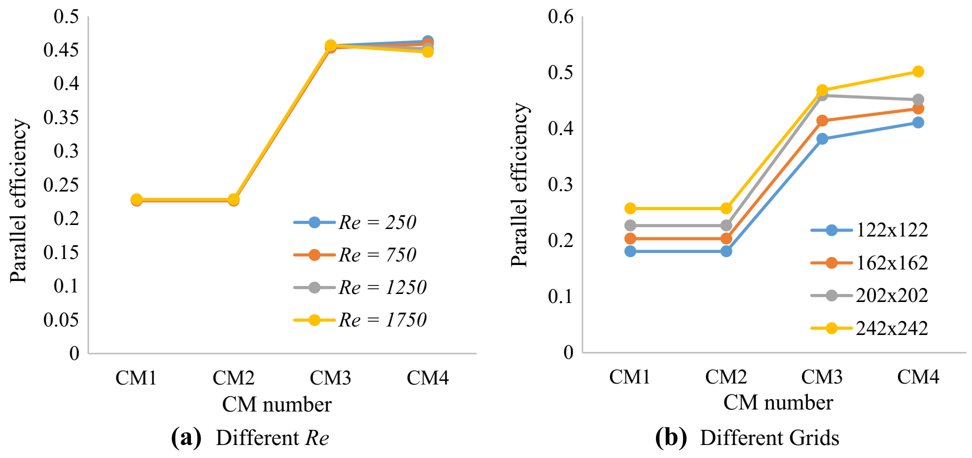


Fig. 16 Parallel efficiency on different machines for various conditions and internal flow

Fig. 17 Parallel efficiency of the FVM code on different machines, grid sizes, and flow Re for external flow



CM1 and CM2. Much higher grid size than those considered in this study can further cause better parallel efficiency on CM1 and CM2.

Parallel efficiency using RBSOR scheme for external flow and different conditions is illustrated in Fig. 17. The grid size is fixed at 202 × 202 for different Re to analyze the parallel efficiency of the FVM code. As shown in Fig. 17a the parallel efficiency is same for all Re due to same speedups obtained on CMs as shown in Fig. 11. However with increasing grid size for external flow, the parallel efficiency increases on all CMs as shown in Fig. 17b. As mentioned previously, by increasing the grid size compared to grid size considered for internal flow, the parallel efficiency increases due to more computational time spent on multiple processors. Hence, grid size has a prominent role in effecting parallel efficiency, whereas the effect of Re is negligible.

5 Conclusions

Parallel performance analysis of the parallelized FVM code developed in-house is analyzed in the form of parallel speedup and parallel efficiency. OpenMP parallel computing paradigm is used for parallelization applying RBSOR scheme. Four computing machines CM1, CM2, CM3, and CM4 for OpenMP parallelization are employed. For computational time analysis, the grid sizes chosen are 42 × 82, 52 × 102, 6 × 122, and 72 × 142 for internal flow. The grid sizes considered for outlet and inlet domain extended are 24 × 122. In case of external flow, 122 × 122, 162 × 162, 202 × 202, and 242 × 242 grid sizes are chosen, while Re is varied from 250 to 1750.

From the complete speedup and parallel efficiency analysis of the parallelized FVM code using different methods, the following important conclusions are drawn:

1. The computational time of the FVM code significantly changes with change in Re and grid size during internal and external flow. Even though for internal flow, less grid size is considered, the computational time is comparatively similar to higher grid size used in the external flow.
2. OpenMP parallelization of the FVM code provides a maximum speedup of up to 1.5 in the present investigation for considered conditions. This speedup can be increased by selecting a higher grid size in which the parallel utilization of multiprocessors will increase.
3. In without nested OpenMP parallelization, the speedup obtained on CMs is similar to speedup obtained using nested OpenMP parallelization.
4. When internal and external flow speedups are compared with respect to serial execution on CM1, it is found that only for higher grid size, the parallelization on CMs with low clock speed is worth and then serial execution on CM1 with higher clock speed.
5. It is observed that grid size affects parallel efficiency to some extent compared to Re, whereas the effect of Re is found to be trivial.

References

1. Ling, Z.; Zhang, Z.; Shi, G.; Fang, X.; Wang, L.; Gao, X.; et al.: Review on thermal management systems using phase change materials for electronic components, Li-ion batteries, and photovoltaic modules. Renew. Sustain. Energy Rev. **31**, 427–438 (2014). https://doi.org/10.1016/j.rser.2013.12.017
2. Bazdidi-Tehrani, F.; Aghaamini, M.; Moghaddam, S.: Radiation effects on turbulent mixed convection in an asymmetrically heated vertical channel. Heat Transf. Eng. **38**, 475–497 (2017). https://doi.org/10.1080/01457632.2016.1194695
3. Shukla, A.; Singh, A.K.; Singh, P.: A comparative study of finite volume method and finite difference method for

convection-diffusion problem. Am. J. Comput. Appl. Math. **1**, 67–73 (2012). https://doi.org/10.5923/j.ajcam.20110102.13
4. Kolditz, O.: Finite Volume Method. Computation. Springer, Berlin, Heidelberg (2013)
5. Patankar, S.: Numerical Heat Transfer and Fluid Flow. CRC Press, Boca Raton (1980)
6. Wang, X.; Guo, L.; Ge, W.; Tang, D.; Ma, J.; Yang, Z.; et al.: Parallel implementation of macro-scale pseudo-particle simulation for particle-fluid systems. Comput. Chem. Eng. **29**, 1543–1553 (2005). https://doi.org/10.1016/j.compchemeng.2004.12.006
7. Gerndt, A.; Sarholz, S.; Wolter, M.; Mey, D.A.; Bischof, C.; Kuhlen, T: Nested OpenMP for efficient computation of 3D critical points in multi-block CFD datasets. In: 2006 ACM/IEEE Conf Supercomput, p. 46 (2006). https://doi.org/10.1145/1188455.1188553
8. Duvigneau, R.; Kloczko, T.; Praveen, C.: A three-level parallelization strategy for robust design in aerodynamics. In: 20th Int Conf Parallel Comput Fluid Dyn, pp. 101–108 (2008)
9. Gourdain, N.; Gicquel, L.; Staffelbach, G.; Vermorel, O.; Duchaine, F.; Boussuge, J.-F.; et al.: High performance parallel computing of flows in complex geometries: I. Methods Comput. Sci. Discov. **2**, 015003 (2009). https://doi.org/10.1088/1749-4699/2/1/015003
10. Gropp, W.D.; Kaushik, D.K.; Keyes, D.E.; Smith, B.F.: High-performance parallel implicit CFD. Parallel Comput. **27**, 337–362 (2001). https://doi.org/10.1016/S0167-8191(00)00075-2
11. Passoni, G.; Cremonesi, P.; Alfonsi, G.: Analysis and implementation of a parallelization strategy on a Navier-Stokes solver for shear flow simulations. Parallel Comput. **27**, 1665–1685 (2001). https://doi.org/10.1016/S0167-8191(01)00114-4
12. Schulz, M.; Krafczyk, M.; Tölke, J.; Rank, E.: Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high-performance computers. High Perform. Sci. Eng. Comput. **21**, 115–122 (2002). https://doi.org/10.1007/978-3-642-55919-8
13. Zhang, L.T.; Wagner, G.J.; Liu, W.K.: A parallelized meshfree method with boundary enrichment for large-scale CFD. J. Comput. Phys. **176**, 483–506 (2002). https://doi.org/10.1006/jcph.2002.6999
14. Peigin, S.; Epstein, B.: Embedded parallelization approach for optimization in aerodynamic design. J. Supercomput. **29**, 243–263 (2004). https://doi.org/10.1023/B:SUPE.0000032780.68664.1b
15. Eyheramendy, D.: Object-oriented parallel CFD with JAVA. Parallel Comput. Fluid Dyn. 2003 Adv. Numer. Methods, Softw. Appl. **2004**, 409–416 (2003). https://doi.org/10.1016/B978-044451612-1/50052-4
16. Jia, R.; Sundén, B.: Parallelization of a multi-blocked CFD code via three strategies for fluid flow and heat transfer analysis. Comput. Fluids **33**, 57–80 (2004). https://doi.org/10.1016/S0045-7930(03)00029-X
17. Lehmkuhl, O.: Borrell, R.; Soria, M.; Oliva, A.: TermoFluids: a new parallel unstructured CFD code for the simulation of turbulent industrial problems on low cost PC cluster. In: Parallel Comput. Fluid Dyn. 2007. Lect. Notes Comput. Sci. Eng., vol. 67, pp. 275–282. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-540-92744-0
18. Oktay, E.; Akay, H.U.; Merttopcuoglu, O.: Parallelized structural topology optimization and CFD coupling for design of aircraft wing structures. Comput. Fluids **49**, 141–145 (2011). https://doi.org/10.1016/j.compfluid.2011.05.005
19. Amritkar, A.; Tafti, D.; Liu, R.; Kufrin, R.; Chapman, B.: OpenMP parallelism for fluid and fluid-particulate systems. Parallel Comput. **38**, 501–517 (2012). https://doi.org/10.1016/j.parco.2012.05.005
20. Amritkar, A.; Deb, S.; Tafti, D.: Efficient parallel CFD-DEM simulations using OpenMP. J. Comput. Phys. **256**, 501–519 (2014). https://doi.org/10.1016/j.jcp.2013.09.007
21. Steijl, R.; Barakos, G.N.: Parallel evaluation of quantum algorithms for computational fluid dynamics. Comput. Fluids **173**, 22–28 (2018). https://doi.org/10.1016/j.compfluid.2018.03.080
22. Gorobets, A.; Soukov, S.; Bogdanov, P.: Multilevel parallelization for simulating compressible turbulent flows on most kinds of hybrid supercomputers. Comput. Fluids **173**, 171–177 (2018). https://doi.org/10.1016/j.compfluid.2018.03.011
23. Lai, J.; Li, H.; Tian, Z.: CPU/GPU heterogeneous parallel CFD solver and optimizations. In: Proc. 2018 Int. Conf. Serv. Robot. Technol., pp. 88–92. ACM, Chengdu (2018)
24. Niedermeier, C.A.; Janssen, C.F.; Indinger, T.: Massively-parallel multi-GPU simulations for fast and accurate automotive aerodynamics. In: 6th Eur. Conf. Comput. Mech. (ECCM 6), Glasgow, UK, pp. 1–8 (2018)
25. Shan, P.; Zhu, R.; Wang, F.; Wu, J.: Efficient approximation of free-surface Green function and OpenMP parallelization in frequency-domain wave–body interactions. J. Mar. Sci. Technol. (2018). https://doi.org/10.1007/s00773-018-0568-9
26. Sheraton, M.V.; Sloot, P.M.A.: Parallel performance analysis of bacterial biofilm simulation models. In: Comput. Sci.—ICCS 2018. Lect. Notes Comput. Sci., vol. 10862, pp. 496–505. Springer International Publishing, Berlin (2018). https://doi.org/10.1007/978-3-319-93713-7
27. Wang, Y.X.; Zhang, L.L.; Liu, W.; Cheng, X.H.; Zhuang, Y.; Chronopoulos, A.T.: Performance optimizations for scalable CFD applications on hybrid CPU + MIC heterogeneous computing system with millions of cores. Comput. Fluids **173**, 226–236 (2018). https://doi.org/10.1016/j.compfluid.2018.03.005
28. Afzal, A.; Mohammed Samee, A.D.; Abdul Razak, R.K.; Ramis, M.K.: Effect of spacing on thermal performance characteristics of Li-ion battery cells. J. Therm. Anal. Calorim. **135**, 1797–1811 (2019). https://doi.org/10.1007/s10973-018-7664-2
29. Mudigere, D.; Sridharan, S.; Deshpande, A.; Park, J.; Heinecke, A.; Smelyanskiy, M.; et al.: Exploring shared-memory optimizations for an unstructured mesh cfd application on modern parallel systems. In: 29th Parallel Distrib. Process. Symp. (IPDPS), 2015 IEEE Int., pp. 723–732 (2015)
30. Couder-Castaneda, C.; Barrios-Pina, H.; Gitler, I.; Arroyo, M.: Performance of a code migration for the simulation of supersonic ejector flow to SMP, MIC, and GPU using OpenMP, OpenMP + LEO, and OpenACC directives. Sci. Program **2015**, 1–20 (2015). https://doi.org/10.1155/2015/739107
31. Xu, Z.; Zhao, H.; Zheng, C.: Accelerating population balance-Monte Carlo simulation for coagulation dynamics from the Markov jump model, stochastic algorithm and GPU parallel computing. J. Comput. Phys. **281**, 844–863 (2015). https://doi.org/10.1016/j.jcp.2014.10.055
32. Niemeyer, K.E.; Sung, C.-J.: Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. J. Supercomput. **67**, 528–564 (2014). https://doi.org/10.1007/s11227-013-1015-7
33. Grelck, C.S.S.: SAC—from high-level programming with arrays to efficient parallel execution. Parallel Process Lett. **13**, 401–412 (2003)
34. Wang, J.L.R.: Assessment of linear finite-difference Poisson-Boltzmann solvers. J. Comput. Chem. **31**, 1689–1698 (2010)
35. Vanderbauwhede, W.T.T.T.: Buffering: a simple and highly effective scheme for parallelization of successive over-relaxation on GPUs and other accelerators. In: Int Conf High Perform Comput Simul, pp. 436–43 (2015)
36. Mattson, T., Eigenmann, R.: OpenMP: an API for writing portable SMP application software. In: Supercomput. 99 Conf. (1999)

37. Kobler, R.; Kranzlmüller, D.; Volkert, J.: Debugging OpenMP programs using event manipulation. In: Eig, R., Voss, M.J. (eds.) OpenMP Shar. Mem. Parallel Program. WOMPAT 2001. Lect. Notes Comput. Sci., pp. 81–89. Springer, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44587-0_8
38. Chapman, B.; Jost, G.; Van Der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming, 10th edn. MIT Press, Cambridge (2008)
39. McCool, M.; Robison, A.; Reinders, J.: Structured Parallel Programming: Patterns for Efficient Computation. Elsevier, Amsterdam (2012)
40. Bücker, H.; Rasch, A.; Wolf, A.: A class of OpenMP applications involving nested parallelism. In: Proc. 2004 ACM Symp. Appl. Comput., ACM; n.d., pp. 220–224
41. Darmana, D.; Deen, N.G.; Kuipers, J.A.M.: Parallelization of an Euler-Lagrange model using mixed domain decomposition and a mirror domain technique: application to dispersed gas-liquid two-phase flow. J. Comput. Phys. **220**, 216–248 (2006). https://doi.org/10.1016/j.jcp.2006.05.011
42. Walther, J.H.; Sbalzarini, I.F.: Large-scale parallel discrete element simulations of granular flow. Eng. Comput. **26**, 688–697 (2009). https://doi.org/10.1108/02644400910975478
43. Mininni, P.D.; Rosenberg, D.; Reddy, R.; Pouquet, A.: A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence. Parallel Comput. **37**, 316–326 (2011)
44. Shang, Y.: A distributed memory parallel Gauss—Seidel algorithm for linear algebraic systems. Comput. Math Appl. **57**, 1369–1376 (2009). https://doi.org/10.1016/j.camwa.2009.01.034