EPTT-2022-0060
Lid-Driven Cavity simulation using the Chapel programming language

Anna Caroline Felix Santos de Jesus
Livia S. Freire
Instituto de Ciências Matemáticas e de Computação (ICMC) - University of São Paulo (USP)
carolinefelix@usp.br, liviafreire@usp.br

Nelson Luis Dias
Federal University of Paraná
nldias@ufpr.br

Abstract. *Most softwares in the context of computational fluid dynamics are developed with the languages C or Fortran, due to their low computational cost. These languages, however, require additional tools such as MPI or OpenMP for parallelism. In this context, the performance of the new programming language Chapel was investigated. This language purports to be fast like Fortran, portable like C and easy to code as Matlab, with a parallelization that is significantly simpler to implement compared to MPI. A code was implemented to solve the two-dimensional, transient, incompressible lid-driven cavity flow in the Chapel 1.24 and Fortran languages. The implementation is based on the finite volume method, incorporating the classical projection method to decouple the velocity and pressure fields. Preliminary verification results using the method of the manufactured solution demonstrate a correct implementation and a convergence of second order in space. Moreover, the velocity profiles obtained for Reynolds number 100 with different meshes are in agreement with literature data. Regarding the performance of the new language, in serial, Chapel 1.24 compiled with the* `--fast` *flag presented a higher computational cost compared to Fortran using the GNU compiler and the* `-Ofast` *optimization flag. However, the difference in computational cost between the languages decreased with increase in the number of grid points, indicating a tendency of similar cost for heavier simulations. Finally, the parallel version of the Chapel code was obtained by including simple commands, resulting in a low speedup that also improved with increase in the number of grid points. For a better evaluation, future work will test heavier simulations in a cluster.*

Keywords: *Computational fluid dynamics (CFD), Incompressible flow, Finite volume method, Projection Method, Chapel, Fortran.*

1. INTRODUCTION

In the context of computational fluid dynamics (CFD), most of the available softwares that simulate the Navier-Stokes equations are implemented in the C or Fortran languages due to their efficiency in solving computationally intensive problems. Such languages, however, are not intrinsically parallelizable, and external tools such as MPI (Message Passing Interface) or OpenMP (Open Multi-Processing) are required. As an alternative, Chamberlain *et al.* (2007) developed a new language that should be as fast as Fortran, as portable as C and as easy to implement as Matlab or Python. The Chapel programming language (an acronym for *Cascade High Productivity Language*) incorporates the best of the aforementioned languages, without the need for programmers to know the details of parallelization. The commands are intuitive and allow building simpler and faster codes, making it a potential tool for the development of new CFD softwares in a more efficient way.

Chapel was designed to support general parallel programming through the use of high-level language abstractions. For example, the commands for parallelization of loops, such as `forall`, are responsible for automatically distributing and sharing data between processes without any explicit input from the programmer. The focus on parallel programming is so relevant that the serial implementation has to be enforced through a `serial` statement. More details about the simplified implementation syntax compared to MPI are given by Barrett *et al.* (2007).

In this work, we test the Chapel language using the two-dimensional cavity as a benchmark flow. The numerical solution is provided by the finite volume approximation and the projection method. The code is verified using the Method of the Manufactured Solution (MMF) and validated against literature results for the same flow. In addition, the efficiency of the serial version of the code is compared to the same code written in Fortran 90. Different implementation strategies regarding the syntax were explored in order to get the maximum serial performance from each code. Finally, initial tests of Chapel in parallel were performed, using up to four processors with hyper-threading in a laptop. Concluding remarks are presented in Section 5.

2. GOVERNING EQUATIONS AND NUMERICAL METHODS

The two-dimensional, incompressible cavity flow is governed by the Navier-Stokes equations described here in conservative, dimensional form, i.e.,

$$\rho\left[\frac{\partial \mathbf{u}}{\partial t} + \nabla\cdot(\mathbf{u}\otimes\mathbf{u})\right] - \mu[\nabla\cdot(\nabla\mathbf{u})] + \nabla\cdot(p\mathbb{I}) = 0 \quad \text{in} \quad \Omega, \tag{1}$$

$$\nabla\cdot\mathbf{u} = 0 \quad \text{in} \quad \Omega, \tag{2}$$

defined in a domain $\Omega \subset \mathbb{R}^2$. In Eqs. (1) and (2), $\mathbf{u}$ and p represent the unknowns for the velocity and pressure fields of the flow, respectively, and ρ and μ are the fluid's density and dynamic viscosity. The notation $\mathbf{u}\otimes\mathbf{u}$ denotes the tensor $[u_i u_j]$, $i, j = 1, 2$ and $\mathbb{I}$ is the identity matrix $\mathbb{I}_{2\times 2}$.

The appropriate boundary conditions on partial Ω must be satisfied, namely:

1. the walls are solid and impermeable;
2. at the instant $t_0 = 0$, the lid of the cavity is instantaneously accelerated to the velocity u_0, which remains constant.

Due to viscous stresses, the motion of the lid "pulls" the fluid that is adjacent to it, driving the flow. The Reynolds number used in the present study is $Re = 100$, with $\nu = 10^{-2}$, $L_x = Ly = 1$ (the length of the box) and $u_0 = 1$. The boundary conditions of the problem are presented in Fig. 1.

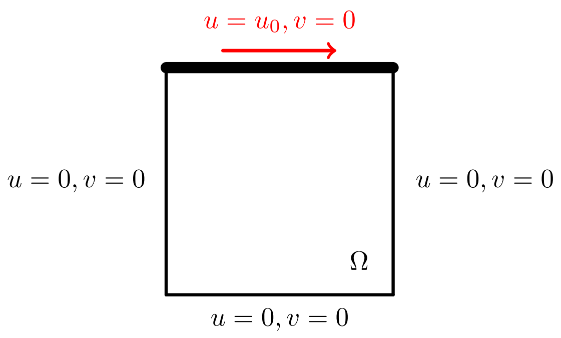


Figure 1: Boundary conditions of the cavity flow. The flow is driven by the movement of the lid.

The spatial discretization was done using the conservative finite volume method. The idea of the method is to subdivide the computational domain into small subdomains and integrate the governing equations over each of them. In this particular work, a uniformly spaced two-dimensional mesh was considered, i.e., $\Omega_i = \Delta x \times \Delta y = L_x/N_x \times L_y/N_y$, where Ω_i is a subdomain of the computational mesh of $\Omega = (-\Delta x, L_x + \Delta x) \times (-\Delta y, L_y + \Delta y)$, and $N_x \times N_y$ are the number of grids. The time discretization was performed using the Forward Euler method. The boundary condition treatment uses the reflection technique, which requires the use of the 4 bands of ghost cells as schematized in Fig. 2.

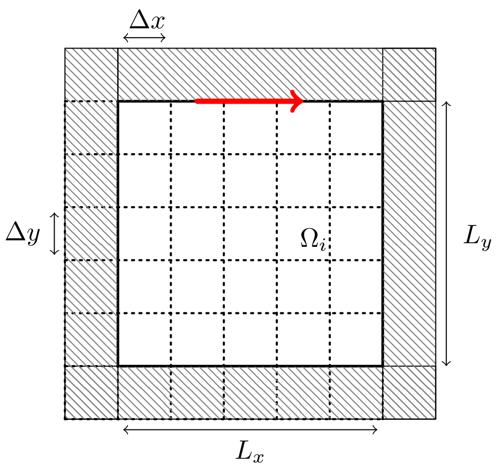


Figure 2: Domain discretization of the cavity flow. Ghost cells are hatched.

In this study a staggered grid was used, in which the unknowns corresponding to the velocity components are located at the faces of the control volume Ω_i, while the pressure unknowns are located in the centers of each volume. In addition, the velocity and pressure fields are segregated by the projection method proposed by Chorin (1968), as described next.

Let $\mathbf{U}^n \approx \mathbf{u}(x,y,t_n) = (u(x,y,t_n), v(x,y,t_n))$ be an approximation for the velocity vector at the n^{th} time step. An intermediate velocity $\mathbf{U}^*$ is calculated using the momentum equation without the pressure gradient term, i.e.,

$$\frac{\mathbf{U}^* - \mathbf{U}^n}{\Delta t} = \frac{1}{\rho}\Big(\mu\mathbf{VISC}^n - \rho\mathbf{CONV}(\mathbf{U}^n)\Big),$$

or

$$\mathbf{U}^* = \mathbf{U}^n + \Delta t\Big(\nu\mathbf{VISC}^n - \mathbf{CONV}(\mathbf{U}^n)\Big). \tag{3}$$

Here, $\mathbf{VISC}^n$ represents the result of $\mathbf{LU}^n$ ($\mathbf{L} \approx \nabla^2$) and $\mathbf{CONV}(\mathbf{U}^n)$ is the operator that approximates the convective (nonlinear) term. Explicitly, they can be written as

$$\begin{aligned}\Big(\mathbf{CONV}(\mathbf{U}^n)\Big)_x =& \frac{1}{\Delta y}\left[\left(\frac{U_{i,j+1}+U_{ij}}{2}\right)\left(\frac{V_{i-1,j+1}+V_{ij}}{2}\right) - \left(\frac{U_{ij}+U_{i,j-1}}{2}\right)\left(\frac{V_{i-1,j}+V_{ij}}{2}\right)\right] \\ &+ \frac{1}{\Delta x}\left[\left(\frac{U_{ij}+U_{i+1,j}}{2}\right)^2 - \left(\frac{U_{i-1,j}+U_{ij}}{2}\right)^2\right],\end{aligned} \tag{4}$$

$$\begin{aligned}\Big(\mathbf{CONV}(\mathbf{U}^n)\Big)_y =& \frac{1}{\Delta y}\left[\left(\frac{V_{ij}+V_{i+1,j}}{2}\right)^2 - \left(\frac{V_{i,j-1}+V_{ij}}{2}\right)^2\right] + \\ &+ \frac{1}{\Delta x}\left[\left(\frac{U_{i,j+1}+U_{ij}}{2}\right)\left(\frac{V_{i-1,j+1}+V_{ij}}{2}\right) - \left(\frac{U_{ij}+U_{i,j-1}}{2}\right)\left(\frac{V_{i-1,j}+V_{ij}}{2}\right)\right],\end{aligned} \tag{5}$$

$$\Big(\mathbf{VISC}^n\Big)_x = \left(\frac{U_{i,j+1}-2U_{ij}+U_{i,j-1}}{\Delta y^2} + \frac{U_{i+1,j}-2U_{ij}+U_{i-1,j}}{\Delta x^2}\right), \tag{6}$$

and

$$\Big(\mathbf{VISC}^n\Big)_y = \left(\frac{V_{i,j+1}-2V_{ij}+V_{i,j-1}}{\Delta y^2} + \frac{V_{i+1,j}-2V_{ij}+V_{i-1,j}}{\Delta x^2}\right). \tag{7}$$

After solving Eq. (3), the solution for $\mathbf{U}^*$ is used in the calculation of the pressure field by solving a Poisson Equation. Note that $\mathbf{U}^*$ does not satisfy the incompressibility condition, i.e., $\mathbf{DU}^* \neq 0$ ($\mathbf{D}$ is the discrete divergence operator). Using the fact that $\mathbf{DU}^{n+1} = 0$, the following Poisson equation for the pressure is obtained:

$$\mathbf{G}P^{n+1} = \frac{\rho}{\Delta t}\mathbf{DU}^*, \tag{8}$$

where $P^{n+1} \approx p(x,y,t{+}1)$ and $\mathbf{G}$ is the discrete gradient operator. Defining RHS $= \rho\mathbf{DU}^*/\Delta t$, the descrete Poisson equation can be written as

$$\frac{P^{n+1}_{i+1,j} - 2P^{n+1}_{ij} + P^{n+1}_{i-1,j}}{\Delta x^2} + \frac{P^{n+1}_{i,j+1} - 2P^{n+1}_{ij} + P^{n+1}_{i,j+1}}{\Delta y^2} = \text{RHS}^{n+1}, \tag{9}$$

where

$$\text{RHS}^{n+1} = \frac{\rho}{\Delta t}\left(\frac{U^*_{i+1,j} - U^*_{i,j}}{\Delta x} + \frac{V^*_{i,j+1} - V^*_{i,j}}{\Delta y}\right). \tag{10}$$

The linear system obtained from Eq. (9) is solved by the iterative SOR (successive over-relaxation) method given by

$$P_{ij} = (1-\omega)P_{ij} + \omega P^*_{ij}, \tag{11}$$

where ω we defined in the interval (0,2).

$$P^*_{ij} = \frac{1}{(2\beta+1)}\left(\frac{-\Delta x^2}{2}\text{RHS}^{n+1} + P_{i-1,j} + P_{i+1,j} + \beta(P_{i,j-1} - P_{i,j+1})\right), \quad \beta := \frac{\Delta x^2}{\Delta y^2}. \tag{12}$$

The boundary condition is imposed as the normal gradient $\partial p/\partial n = 0$ along the boundaries (for details see Fortuna (2000)). The method is iterated while

$$\|\text{error}_p\|_2 \geqslant \text{tol}, \tag{13}$$

where tol is the tolerance for the error in the Poisson equation. In Eq. (13), the Euclidean norm is calculated over the residual error$_p$, defined as

$$\text{error}_p := \text{RHS}^{n+1} - \left(\frac{P^{n+1}_{i+1,j} - 2P^{n+1}_{ij} - P^{n+1}_{i-1,j}}{\Delta x^2} + \frac{P^{n+1}_{i,j+1} - 2P^{n+1}_{ij} - P^{n+1}_{i,j+1}}{\Delta y^2} \right). \tag{14}$$

In the last step of the projection method, the velocity field is updated. Thus, the velocity at t_{n+1} is given by

$$\frac{\mathbf{U}^{n+1} - \mathbf{U}^n}{\Delta t} = -\frac{1}{\rho}\mathbf{G}P^{n+1}, \tag{15}$$

that is,

$$U_{ij}{}^{n+1} = U^*_{ij} - \frac{\Delta t}{\rho}\left(\frac{P^{n+1}_{i+1,j} - P^{n+1}_{i,j}}{\Delta x} \right) \quad \text{and} \quad V^{n+1}_{ij} = V^*_{ij} - \frac{\Delta t}{\rho}\left(\frac{P^{n+1}_{i,j+1} - P^{n+1}_{i,j}}{\Delta y} \right). \tag{16}$$

Eliminating $\mathbf{U}^*$, we have

$$\frac{\mathbf{U}^{n+1} - \mathbf{U}^n}{\Delta t} - \mathbf{CONV}(\mathbf{U}^n) + \nu\mathbf{VISC}^n + \frac{1}{\rho}\mathbf{G}P^{n+1} = 0, \tag{17}$$

$$\mathbf{DU}^{n+1} = 0, \tag{18}$$

which corresponds to the original system of equations.

3. CODE VERIFICATION AND VALIDATION

3.1 Code verification

The Method of Manufactured Solution (MMS) was developed to verify if the code correctly solves its governing mathematical equations, by creating an exact (manufactured) solution that should be obtained if the code is free of typos and programming errors (Roache, 2002). In addition, the difference between the exact solution and the solution generated by the code using different spatial resolutions provides an estimate of the order of accuracy of the code.

As proposed by Chorin (1968), we consider the exact solution

$$u(x, y, t) = -\cos x \sin y e^{-2t}, \tag{19}$$

$$v(x, y, t) = -\sin x \cos y e^{-2t}, \tag{20}$$

$$p(x, y, t) = -\frac{1}{4}(\cos 2x + \cos 2y)e^{-4t}, \tag{21}$$

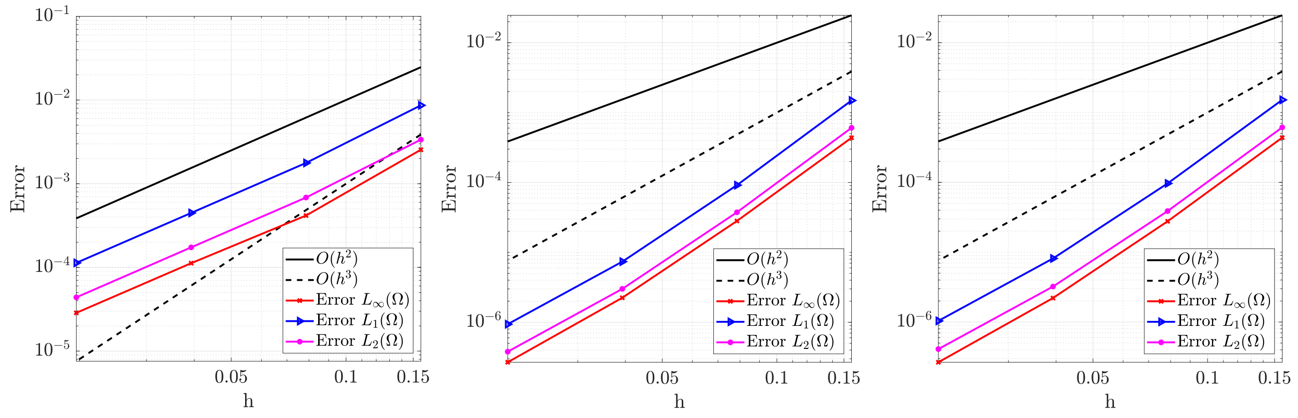
defined in the domain $\Omega = (0, \pi) \times (0, \pi)$. Substituting Eqs. (19)–(21) in Eq. (1) no additional forcing term is obtained, which simplifies the evaluation. The initial condition corresponds to Eqs. (19)–(21) at time $t = 0$. Note that this method does not require solutions with physical meaning, and the choice here does not correspond to the boundary conditions for the lid driven cavity problem. The exact equation for the pressure field, in particular, does not satisfy the criterion of $\partial p/\partial n = 0$. Therefore, we impose the exact values of the velocity field on $\partial\Omega$ (Dirichlet condition) and impose the exact values of the pressure in the volumes adjacent to the boundary.

The errors obtained in the verification test indicates that the code is correctly resolving the two-dimensional Navier-Stokes equation. Furthermore, by fixing the time at $t = 0.15$ s, a convergence study is done in space, considering different meshes (20^2, 40^2, 80^2 and 160^2). For the iterative method, the parameter ω was set as 1.94 for all proposed meshes, since the method is sensitive to the chosen value of ω in the interval $0 < \omega < 2$. Table 1 shows the results of the error measured with the $L_\infty(\Omega)$ norm.

Table 1: Errors in L_∞ norm to U^n, V^n, P^{n+1}

h	$\max \lvert U^n - u(t_n)\rvert$	O	$\max \lvert V^n - v(t_n)\rvert$	O	$\max \lvert P^{n+1} - p(t_{n+1})\rvert$	O
$\pi/20$	2.5468e-03	-	4.3647e-04	-	4.3696e-04	-
$\pi/40$	4.1653e-04	2.6122e+00	2.8112e-05	3.9566e+00	2.76911e-05	3.9800e+00
$\pi/80$	1.1227e-04	1.8914e+00	2.2419e-06	3.6483e+00	2.2065e-06	3.6495e+00
$\pi/160$	2.8640e-05	1.9708e+00	2.6770e-07	3.0660e+00	2.6545e-07	3.0552e+00

The plot of the error as a function of $h = \Delta x = \Delta y$, measured with different norms, is shown in Fig. 3. Convergences of $O(h^2)$ and $O(h^3)$ were obtained for the pressure and velocity fields, respectively, which is in agreement with the order of the numerical method used.

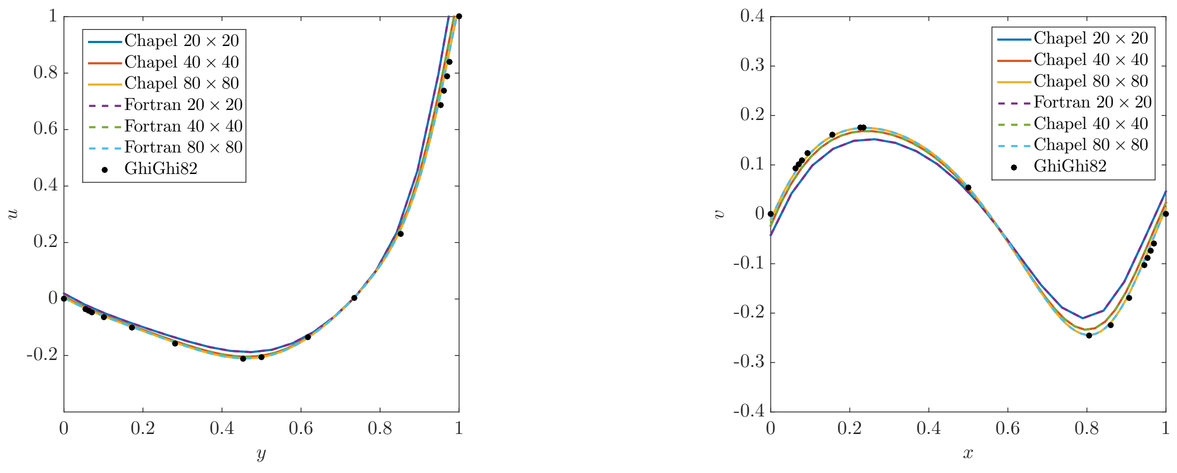


(a) Error in pressure (b) Error in velocity (x - component) (c) Error in velocity (y - component)

Figure 3: Errors in the code as a function of grid size h, for different error norms.

3.2 Code validation

In this section, the validation of the code using results from the literature is presented. The velocity profiles, shown in Fig. 4, are plotted for different meshes on top of the result by (Ghia *et al.*, 1982). The comparison uses $\omega = 1.94$ for the iterative method SOR. The agreement between the present study and the reference results increases with resolution, indicating that the code is correctly simulating the cavity flow.



(a) (b)

Figure 4: (a) x-component velocity profile along the vertical line ($u(x = 0.5, y)$) and (b) y-component velocity profile along the horizontal line ($v(x, y = 0.5)$).

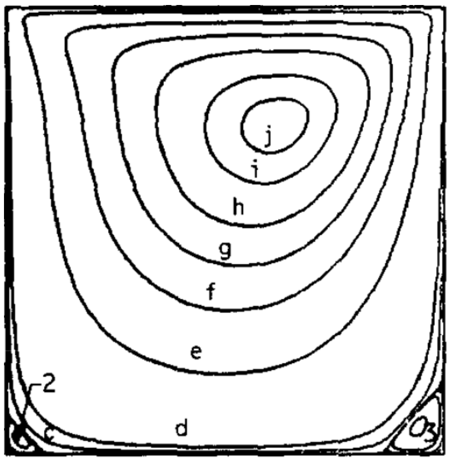
Another way of verifying the result of this simulation is presented in Fig. 5b for a mesh of 80^2 grids. The generated solution was obtained at the time instant $t = 30$s. The formation of a principal vortex in the bulk of the flow and two secondary vortices at the inferior corners can be observed, as obtained in the reference study.

4. COMPARISON BETWEEN CHAPEL AND FORTRAN

4.1 Impact of syntax on efficiency

In this section, we highlight the main implementation details that significantly impact code runtime. Programming two versions of a code in different languages requires taking into account their specificities. In other words, to build an efficient code in Fortran and Chapel requires knowing their particular technical details. For example, the way arrays are accessed in memory affects the performance of the code. In Fortran, as in MATLAB, the array is stored in *column-major*, whereas C and Chapel uses *row-major* (see Suh and Kim (2014)). As accessing memory in the correct order makes the code runs faster, the comparison between Chapel and Fortran cannot be performed with identical codes.

As another example, when calculating the error in Eq. (13), the Euclidean norm can be calculated in Fortran directly



(a) Velocity field obtained by Ghia Ghia *et al.* (1982) for a mesh 129^2 and $Re = 100$.

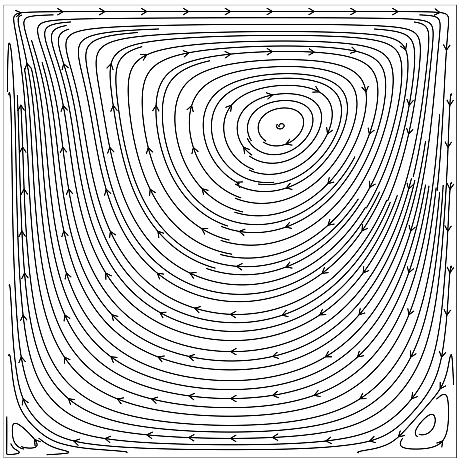

(b) Velocity field from the present study using 80^2 grids and $Re = 100$.

Figure 5: Streamlines of the flow field compared to the literature.

with the command `norm2(errop)`. In Chapel, on the other hand, it is preferred to use an alternative expression with reduction operators. The command `reduce`, an operator with implicit parallelism in Chapel, is responsible for combining a set of values to produce a single value (see `https://github.com/chapel-lang/chapel/blob/main/test/release/examples/primers/reductions.chpl`), and it can be used in the construction of the Euclidean norm calculation, instead of using the function `norm`, pre-defined in the library `LinearAlgebra`, as follows

```
// param p : normType;                  // defaut: euclidian norm
// errop = norm(erro, p);               // residue norm
errop = sqrt( + reduce (erro ** 2) );   // residue norm
```

Also, when implementing the iterative method section for solving Poisson equation, it is necessary to normalize the pressure field at each iteration of the method to prevent the pressure values from increasing or decreasing arbitrarily. In Fortran, this step is implemented as

```
pnew = pnew -  pnew(2,2);               // pressure normalization
```

whereas in Chapel the implementation is

```
pnew -= pnew[2,2];                      //pressure normalization
```

similar to what would be done in the C programming code.

We note that the choice of the time step of the adopted numerical method must satisfy the following conditions:

$$\Delta t \leqslant \tau \min\{\Delta x^2, \Delta y^2\} \quad \text{and} \quad \Delta t \leqslant \frac{\min\{\Delta x, \Delta y\}}{\max\{|u|, |v|\}}, \quad 0.2 < \tau < 0.6. \tag{22}$$

In Fortran, this corresponds to

```
! Find the maximum velocity, in absolute value
mu = maxval(abs(unew))
mv = maxval(abs(vnew))

dt1 = min(dx/mu, dy/mv)                            ! CFL condition
dt2 = 0.5d0/vkinem/(1.d0/(dx**2) + 1.d0/(dy**2))
dt = 0.4d0*min(dt1,dt2)                            ! constant  tal=0.4
```

and in Chapel

```
// Find the maximum velocity, in absolute value
mu = max reduce(abs(unew));
mv = max reduce(abs(vnew));

dt1 = min(dx/mu, dy/mv);                          // CFL condition
dt2 = 0.5/vkinem/(1.0/(dx**2) + 1.0/(dy**2));
dt = 0.4*min(dt1,dt2);                            // constant tal = 0.4
```

The `max reduce` command is also part of Chapel's reduction operators. It is normally used to terminate the maximum input of an array and is a parallel command.

Finally, we note that the Chapel code implementation is algebra-sensitive. For example, when calculating the quadratic value of the finite volume size Δx for the time-step calculation (22), we have

```
dt2 = 0.5/vkinem/(1.0/(dx**2) + 1.0/(dy**2));
```

rather than

```
dx2 = dx*dx;
dy2 = dy*dy;

dt2 = 0.5/vkinem/(1.0/dx2 + 1.0/dy2);
```

which has a higher runtime value. Although a significant effort was put into identifying ways of making each code as efficient as possible, we recognize that there is likely room for improvement on each code, which should be taken into account when evaluating the results from the next subsection.

4.2 Serial code comparison

Our experimental setup is performed on a laptop Lenovo Ideapad S145-15IWL Intel Core i5 whose configuration is shown in Tab. 2. As can be seen, this machine has 4 cores, and it also includes hyper-threading.

Table 2: Machine and Operating System.

Machine	Notebook Lenovo IdeaPad S145-15IWL
Memory	7.5 GiB
Processador	Intel© Core™ i5-8265U CPU @ 1.60GHz × 4
Operating System	Linux Mint 20.3 Cinnamon

Because we used implicitly parallel operators in the Chapel code (arrays initialization and reduce operator), it was necessary to disable parallelism using the statement `serial`, in order to compare it with the serial version of Fortran. The modification is easily implemented by doing

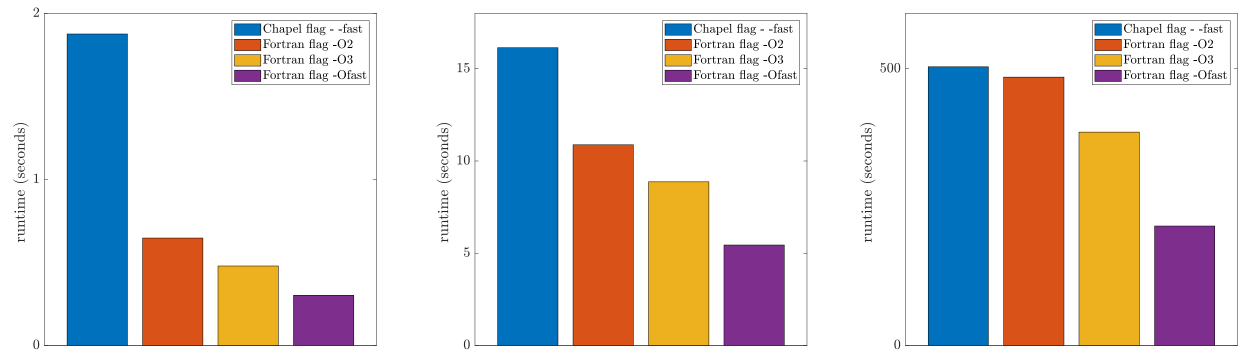
```
serial{

// instructions

}
```

allowing the evaluation of the serial performance between the two languages for different grids (20^2, 40^2 and 80^2).

The Chapel code was compiled with the `--fast` optimization flag, whereas the Fortran code was compiled with different optimization flags of the GNU compiler (`-O2`, `-O3` `-Ofast`). For more details see `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`. The average runtimes are shown in Fig. 6.



(a) mesh dimension 20^2 (b) mesh dimension 40^2 (c) mesh dimension 80^2

Figure 6: Average runtimes for different grids.

Overall, Chapel's simulation is slower than Fortran with any optimization flag. However, Chapel's runtime results compared to Fortran improve as the computational cost of the problem increases (by increasing the number of grids).

Therefore, if the tendency continues, it is possible that Chapel's performance becomes comparable to Fortran for heavier simulations.

4.3 Parallel Implementation in Chapel

Because Chapel was developed with a focus on parallel programming, in this section we investigate its parallel performance by removing the serial command from the code. In addition, we keep the reduction operators of the code and replace the traditional loop command `for` with the parallel command `forall`. The `forall` command divides the tasks between the number of maximum threads at each "locale" (similar to the nodes in a cluster), which is the number of threads of the system by default. One of the changes is exemplified in the following excerpt, which shows the calculation of the source term of Poisson equation:
(serial version)

```
//Step 2: Build the right-hand side of the equation Poisson equation for the pressure
//D G(p^{n+1}) = (rho/dt)*D(u^*)
for i in 2..Nx-1 do{
        for j in 2..Ny-1 do{
                rhs[i,j] = rho/dt*((ustar[i+1,j] - ustar[i,j])/dx +
                                   (vstar[i,j+1] - vstar[i,j])/dy);
        }
}
```

(parallel version)

```
//Step 2: Build the right-hand side of the equation Poisson equation for the pressure
//D G(p^{n+1}) = (rho/dt)*D(u^*)
const Inner = {2..Nx-1, 2..Ny-1};
forall (i,j) in Inner do{
        rhs[i,j] = rho/dt*((ustar[i+1,j] - ustar[i,j])/dx +
                           (vstar[i,j+1] - vstar[i,j])/dy);
}
```

Another parallelization command tested here corresponds to the embedding statement `cobegin`, which creates a fixed number of independent tasks based on the number of available threads, when the tasks inside the loop do not depend on previous values. For example, when implementing the boundary condition during the velocity field update, we can do:

```
// Update velocity field
cobegin{
    unew[3..Nx-1,1] = - unew[3..Nx-1,2];
    unew[3..Nx-1,Ny] = 2.0*u0 - unew[3..Nx-1,Ny-1];

    vnew[1,3..Ny-1] = - vnew[2,3..Ny-1];
    vnew[Nx,3..Ny-1] = - vnew[Nx-1,3..Ny-1];
}
```

In Chapel, the number of threads available on the machine can be found using the command `locale.numPUs()`, which returns the number of processor cores in the "locale" (nodes). For example, if we use

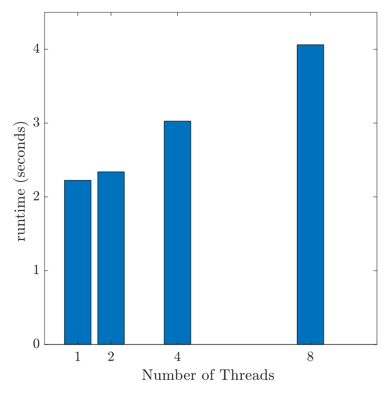
```
var npu = Locales[0].numPUs(logical=true);  // threads (maximum parallelism)
var npc = Locales[0].numPUs(logical=false); // cores
```

the output is number of cores = 4 and number of thread = 8. Therefore, each core can run two processes simultaneously.

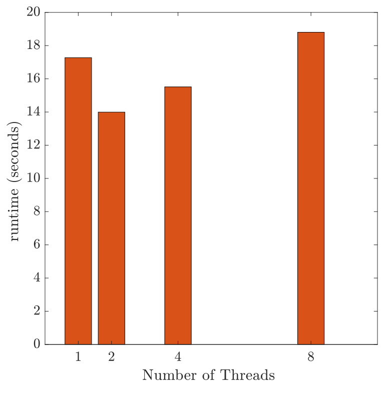
In order to evaluate the parallelization, we run each code with a different number of threads by changing only the environment variable `CHPL_RT_NUM_THREADS_PER_LOCALE`. If the number of threads desired is 2, for example, then we can set `CHPL_RT_NUM_THREADS_PER_LOCALE=2`, and the commands `forall(i,j)` and `cobegin` will divide the tasks inside the loop into 2 processors.

Two parallel versions of the code were tested here, the first replacing only the loop command `for` by `forall`, and the second keeping the `forall` command and adding the `cobegin` statement. In these tests, each code was run three times and speedups were calculated from the average result. Figures 7 and 8 show the average execution time for each adopted mesh and Fig. 9 provides the speedup as a function of the number of threads ($S_{\text{npu}} = T_1/T_{\text{npu}}$, where T_1 is the serial runtime and T_{npu} is the runtime on npu threads).

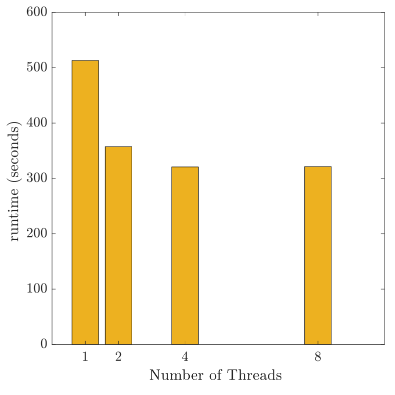
Based on these results, we can observe that, for a small number of grid points (20^2), there is no gain in parallelization when using only `forall` (Fig. 7a), and only a small gain when using two threads if `cobegin` is included (Fig. 8a). As the number of grid points increases, the effectiveness of parallelization becomes more evident, in particular for 2 threads. The tendency of these results indicate that, for heavier codes (more grid points, for example), the gain in parallelization will likely become more relevant, as can be observed in the speedup result (Fig. 9).



(a) mesh dimension 20^2



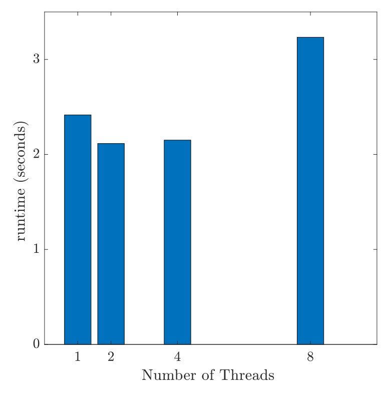
(b) mesh dimension 40^2



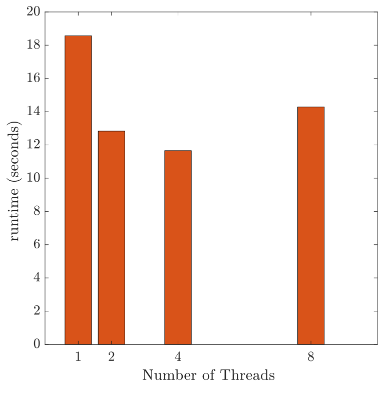
(c) mesh dimension 80^2

Figure 7: Average runtimes for different meshes and varying the number of threads (using only the forall command)

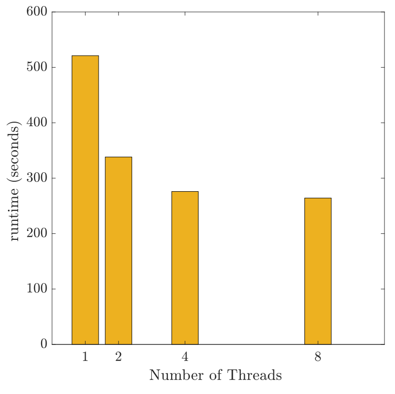
.



(a) mesh dimension 20^2

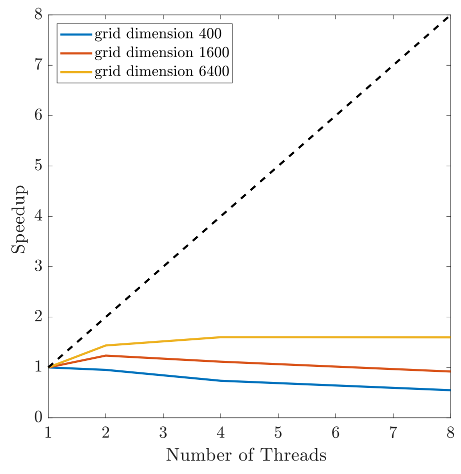


(b) mesh dimension 40^2

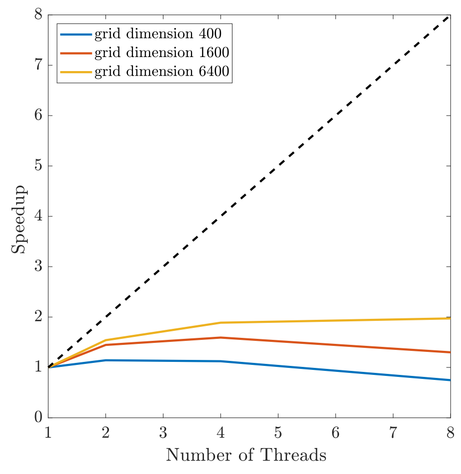


(c) mesh dimension 80^2

Figure 8: Average runtimes for different meshes and varying the number of threads (using forall + cobegin).



(a)



(b)

Figure 9: Speedup for varying number of threads. The dashed line corresponds to an ideal speedup. Speedup for (a) forall (b) forall + cobegin.

In the future, this code will be evaluated in a cluster, which is a more controlled environment. More than 8 processes will be tested using heavier simulations, such as higher Reynolds numbers and the three-dimensional cavitation problem.

5. CONCLUSIONS

In this study, we evaluated the potential of the Chapel programming language for writing a CFD code. The two-dimensional cavity benchmark flow was used with $Re = 100$. The code was verified by the method of manufactured solution and validated against literature results. For a simulation in serial, the Chapel code was slower than the corresponding Fortran code, but the difference in time decreased when the number of grid points increased, indicating a tendency of similar runtime for heavier simulations. The parallel version of the Chapel code, which is extremely simple to implement, was not effective for a small simulation (20^2 grid points), but presented a tendency of improvement in speedup with the increase in the number of grid points. Since the tests presented here were performed in a laptop, it is desired to perform additional tests in a cluster, with a larger number of points, taking advantage of more processors and without the interference of the background operating system. This test will be performed in the future for larger Reynolds numbers and the 3D cavity flow.

6. ACKNOWLEDGEMENTS

This study was funded by the Coordination for the Improvement of Higher Education Personnel (CAPES grant PROEX 88887.671252/2022-00) and the São Paulo Research Foundation (FAPESP grant Nº. 2018/24284-1).

7. REFERENCES

Barrett, R., Roth, P. and Poole, S., 2007. "Finite difference stencils implemented using Chapel". Technical Report TM-2007/119, Oak Ridge National Laboratory.

Chamberlain, B., Callahan, D. and Zima, H., 2007. "Parallel programmability and the chapel language". *The International Journal of High Performance Computing Applications*, Vol. 21, No. 3, pp. 291–312. doi:10.1177/1094342007078442.

Chorin, A., 1968. "Numerical solution of the navier–stokes equations". *Mathematics of Computation*, Vol. 22. doi: 10.2307/2004575.

Fortuna, A.d.O., 2000. *Técnicas Computacionais para Dinâmica dos Fluídos Vol. 30.* Edusp.

Ghia, U., Ghia, K. and Shin, C., 1982. "High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method". *Journal of Computational Physics*, Vol. 48, No. 3, pp. 387–411. doi:10.1016/0021-9991(82)90058-4.

Roache, P., 2002. "Code verification by the method of manufactured solutions". *Journal of Fluids Engineering*, Vol. 124, p. 4. doi:10.1115/1.1436090.

Suh, J.W. and Kim, Y., 2014. "1 - accelerating matlab without gpu". In J.W. Suh and Y. Kim, eds., *Accelerating MATLAB with GPU Computing*, Morgan Kaufmann, Boston, pp. 1–17. doi:10.1016/B978-0-12-408080-5.00001-8.

8. RESPONSIBILITY NOTICE

The authors Anna Caroline Felix Santos de Jesus, Livia S. Freire and Nelson Luís Dias are the only responsible for the printed material included in this paper.