THE UNIVERSITY OF MELBOURNE

MCEN90031

Applied High Performance Computing

Assignment 1

The Shallow Water Equations

Andres Chaves (706801)
Diego Montufar (661608)

Master of Information Technology
29/09/2014

Summary

Differential equations can be used to model real world problems and phenomena like weather, hydrodynamics, biology, etc. In this Assigment, we solved The Shallow water equations, with the 4^{th} order finite central difference and the 4^{th} order Runge-Kutta method, in a Matlab and C++ program. Then we parallelized the serial C++ version with both OpenMP and MPI. The visualization of results was created with Paraview. The process of implementing the programs in order to solve the problem includes the derivation of the governing equations and the explanation of how we parallelized each version of the program. Finally we performed a speed up test for the parallelized versions getting as a result an improvement in time execution of 333% for the OpenMP version and a linear scaling for the MPI one.

Table of Contents

The Shallow Water Equations

1. Introduction

Many engineering problems can be solved through mathematical models that allow us to study and simulate complex systems. With differential equations is possible to model a wide variety of problems, for example we can study stationary systems, heat, sound and wave propagation, biology, electrostatics and electrodynamics, making possible to analyze phenomena with application in many science areas.

Since in many problems the analytical solution of a Differential Equation cannot be derived, several numerical techniques were developed in order to approximate the solution. As computers execute arithmetical operations faster than humans, they are the perfect tool to use in order to implement a numerical method solution (Canale & Chapra, 2010).

However, the computational power needed to solve complex problems also requires a mechanism to distribute the data processing across independent machines something known as Distributed Systems. So nowadays in order to take full advantage of the computers processing power we use parallel and multicore computing as a tool for improve performance.

In this document we are going to work through the application of numerical methods in order to implement the solution of a system of PDE's, in this case the Shallow Water Equations, with programs developed in Matlab and C++. We will develop a single threaded, parallel and distributed version as well. Then we will evaluate results by performing a graphical visualization of each one and present the scalability results obtained in weak and strong scaling statistical charts.

The learning outcomes of this Assignment will be the capacity to apply and analyze numerical methods to solve a system of ODE's/PDE's by discretizing space and time and also implement computer programs with OpenMP and MPI and explore the problematic and limitations of them in terms of performance.

2. Methods

2.1 Model Problem

The Shallow Water equations describe the motion of a liquid when its depth is small compared to the wavelength of the surface waves. These equations derive from the principles of conservation of mass and conservation of momentum and give rise to a coupled system of first order nonlinear PDE's of the form (Moore, Assignment 1, Applied High Performance Computing, 2014):

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} = -g\,\nabla h \tag{1}$$

$$\frac{\partial h}{\partial t} + \nabla \cdot \boldsymbol{v}h = 0 \tag{2}$$

Where:

- $\boldsymbol{h}$ is the depth of the water [m],
- v is the velocity of the fluid with components v_x and v_y [ms^{-1}]
- g is the gravitational constant $9.81 N kg^{-1}$.

This is the non-conservative form of the Shallow Water Equations ignoring the *Coriolis* and the *viscous drag* coefficients in order to idealize the model problem. (Wikipedia, The Shallow Water Equations, 2014)

The computational domain of the problem (a square region of space) will be $x \in [0, 100]$; $y \in [0, 100]$ and $t \in [0, 100]$, with periodic boundary conditions for all three fields and initial conditions:

- $V_x(x, y, 0) = 0$,
- $V_y(x, y, 0) = 0$,
- $h(x, y, 0) = 1 + 0.5e^{-\frac{1}{25}[(x-30)^2+(y-30)^2]}$

2.2 Spatial and temporal discretization

In order to use numerical methods to solve the PDE's of this problem, we are going to use the 2nd and 4th order finite difference method for the spatial discretization. Then for the temporal discretization we will use the 4th order Runge-Kutta Method.

The first step is to translate the vector notation of the PDE's into an expanded notation.

From equation (1) we get the resulting matrix of operator *nabla* (∇) to the vector $\boldsymbol{v}$:

$$\nabla \boldsymbol{v} = \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} \end{bmatrix}$$

Then the obtained result is multiplied by the vector $\boldsymbol{v}$:

$$\boldsymbol{v} \cdot \nabla \boldsymbol{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} \end{bmatrix} = \begin{bmatrix} v_x \frac{\partial v_x}{\partial x} + v_y \frac{\partial v_x}{\partial y} \\ v_x \frac{\partial v_y}{\partial x} + v_y \frac{\partial v_y}{\partial y} \end{bmatrix}$$

And then replacing the expressions in equation (1) we get:

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} = \begin{bmatrix} \frac{\partial v_x}{\partial t} \\ \frac{\partial v_y}{\partial t} \end{bmatrix} + \begin{bmatrix} v_x \frac{\partial v_x}{\partial x} + v_y \frac{\partial v_x}{\partial y} \\ v_x \frac{\partial v_y}{\partial x} + v_y \frac{\partial v_y}{\partial y} \end{bmatrix} = \begin{bmatrix} -g \frac{\partial h}{\partial x} \\ -g \frac{\partial h}{\partial y} \end{bmatrix}$$

And finally the corresponding momentum equations are:

$$\frac{\partial v_x}{\partial t} = -v_x \frac{\partial v_x}{\partial x} - v_y \frac{\partial v_x}{\partial y} - g \frac{\partial h}{\partial x} \tag{3}$$

$$\frac{\partial v_y}{\partial t} = -v_x \frac{\partial v_y}{\partial x} - v_y \frac{\partial v_y}{\partial y} - g \frac{\partial h}{\partial y} \tag{4}$$

Following a similar approach with equation (2) we have:

$$\nabla \cdot \boldsymbol{v} h = \frac{\partial v_x h}{\partial x} - \frac{\partial v_y h}{\partial y}$$

Hence:

$$\frac{\partial h}{\partial t} = -\frac{\partial v_x h}{\partial x} - \frac{\partial v_y h}{\partial y} \tag{5}$$

With equations (3), (4) and (5) (system of PDE's) we can start applying the numerical methods for the spatial discretization.

We first evaluate the spatial discretization by using the 2nd order central difference method using equation (6) and then the 4th order central difference method with equation (7):

$$\left.\frac{d\phi}{dx}\right|_{x_i} = \frac{1}{2\Delta x}(\phi_{i+1} - \phi_{i-1}) \tag{6}$$

$$\left.\frac{d\phi}{dx}\right|_{x_i} = \frac{1}{12\Delta x}(\phi_{i-2} - 8\phi_{i-1} + 8\phi_{i+1} - \phi_{i+2}) \tag{7}$$

Taking into account that this is a 2D problem, we apply equation (6) in equations (3), (4) and (5) obtaining as result three discretized equations which will be used as our first approach in order to solve the problem. The reason we are using the 2nd order as well as the 4th order is because

we want to assess the accuracy of each method when they are implemented in a program. Also using the more simple equations of the 2nd central difference will help us to adjust parameters prior to evaluate the 4th. We know in advance that the 4th method is better than the 2nd because it minimizes error by adding more terms of the Fourier series expansion (which is the origin of equation 6 and 7) into the equation. Table 1. shows the resulting discretized Shallow water equations.

$$\left.\frac{\partial V_{x,i,j}}{\partial t}\right|_{x_i} = -\frac{g}{2\Delta x}\left(h_{i+1,j} - h_{i-1,j}\right) - \frac{V_{x,i,j}}{2\Delta x}\left(V_{x,i+1,j} - V_{x,i-1,j}\right) - \frac{V_{y,i,j}}{2\Delta y}\left(V_{x,i,j+1} - V_{x,i,j-1}\right)$$

$$\left.\frac{\partial V_{y,i,j}}{\partial t}\right|_{y_i} = -\frac{g}{2\Delta y}\left(h_{i,j+1} - h_{i,j-1}\right) - \frac{V_{x,i,j}}{2\Delta x}\left(V_{y,i+1,j} - V_{y,i-1,j}\right) - \frac{V_{y,i,j}}{2\Delta y}\left(V_{y,i,j+1} - V_{y,i,j-1}\right)$$

$$\left.\frac{\partial h_{i,j}}{\partial t}\right|_{h_i} = -\frac{V_{x,i,j}}{2\Delta x}\left(h_{i+1,j} - h_{i-1,j}\right) - \frac{h_{i,j}}{2\Delta x}\left(V_{x,i+1,j} - V_{x,i-1,j}\right) - \frac{V_{y,i,j}}{2\Delta y}\left(h_{i,j+1} - h_{i,j-1}\right) - \frac{h_{i,j}}{2\Delta y}\left(h_{i,j+1} - h_{i,j-1}\right)$$

6.a 2nd order central difference discretized equations

$$\left.\frac{\partial V_{x,i,j}}{\partial t}\right|_{x_i} = -\frac{g}{12\Delta x}\left(h_{i-2,j} - 8h_{i-1,j} + 8h_{i+1,j} - h_{i+2,j}\right) - \frac{V_{x,i,i}}{12\Delta x}\left(V_{x,i-2,j} - 8V_{x,i-1,j} + 8V_{x,i+1,j} - V_{x,i+2,j}\right) - \frac{V_{y,i,i}}{12\Delta y}\left(V_{x,i,j-2} - 8V_{x,i,j-1} + 8V_{x,i,j+1} - V_{x,i,j+2}\right)$$

$$\left.\frac{\partial V_{y,i,j}}{\partial t}\right|_{y_i} = -\frac{g}{12\Delta y}\left(h_{i,j-2} - 8h_{i,j-1} + 8h_{i,j+1} - h_{i,j+2}\right) - \frac{V_{x,i,i}}{12\Delta x}\left(V_{y,i-2,j} - 8V_{y,i-1,j} + 8V_{y,i+1,j} - V_{y,i+2,j}\right) - \frac{V_{y,i,i}}{12\Delta y}\left(V_{y,i,j-2} - 8V_{y,i,j-1} + 8V_{y,i,j+1} - V_{y,i,j+2}\right)$$

$$\left.\frac{\partial h_{i,j}}{\partial t}\right|_{h_i} = -\frac{V_{x,i,j}}{12\Delta x}\left(h_{i-1,j} - 8h_{i-1,j} + 8h_{i+1,j} - h_{i+2,j}\right) - \frac{h_{i,j}}{12\Delta x}\left(V_{x,i-2,j} - 8V_{x,i-1,j} + 8V_{x,i+1,j} - V_{x,i+2,j}\right) - \frac{V_{y,i,j}}{12\Delta y}\left(h_{i,j-2} - 8h_{i,j-1} + 8h_{i,j+1} - h_{i,j+2}\right) - \frac{h_{i,j}}{12\Delta y}\left(V_{y,i,j-2} - 8V_{y,i,j-1} + 8V_{y,i,j+1} - V_{y,i,j+2}\right)$$

7.a 4th order central difference discretized equations

Table 1. Spatial discretization of the Shallow water equations

Once we have obtained the spatial discretization of the PDE's, the next step is to discretize the temporal domain. In order to do that, we are going to use the 4th order Runge-Kutta method which can be expressed as follows (Moore, Applied Numerical Methods, 2013):

$$\phi^{l+1} = \phi^l + \Delta t\left(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4\right) \qquad (8)$$

Where:

$$k_1 = f(\phi^l, t^l)$$

$$k_2 = f\left(\phi^l + \frac{\Delta t}{2}k_1, t^l + \frac{\Delta t}{2}\right)$$

$$k_3 = f\left(\phi^l + \frac{\Delta t}{2}k_2, t^l + \frac{\Delta t}{2}\right)$$

$$k_4 = f(\phi^l + \Delta t k_3, t^l + \Delta t)$$

For this problem ϕ represents the velocity components $\boldsymbol{V_x}$ and $\boldsymbol{V_y}$ as well as the $\boldsymbol{h}$ field. We are going to develop a method in order to evaluate the k_n values (which will be column vectors since we are dealing with a systems of ODE's) at each time step by calling a function f that is going to evaluate those three vectors repeatedly as we'll explain in part 2.4.

2.3 Stability and error Analysis

So let's now look at the stability of the 4th order Runge-Kutta method. To do so we will perform a stability analysis on the method considering the model initial value problem:

$$\frac{d\phi}{dt} = \lambda\phi \qquad (9)$$

We first substitute k_n values into equation (8):

$$\begin{aligned}\phi^{l+1} = \phi^l + \Delta t\Bigg(&\frac{1}{6}f(\phi^l, t^l) + \frac{1}{3}f\left(\phi^l + \frac{\Delta t}{2}f(\phi^l, t^l), t^l + \frac{\Delta t}{2}\right)\\ &+ \frac{1}{3}f\left(\phi^l + \frac{\Delta t}{2}f\left(\phi^l + \frac{\Delta t}{2}f(\phi^l, t^l), t^l + \frac{\Delta t}{2}\right), t^l + \frac{\Delta t}{2}\right)\\ &+ \frac{1}{6}f\left(\phi^l + \Delta t f\left(\phi^l + \frac{\Delta t}{2}f\left(\phi^l + \frac{\Delta t}{2}f(\phi^l, t^l), t^l + \frac{\Delta t}{2}\right), t^l + \frac{\Delta t}{2}\right), t^l + \Delta t\right)\Bigg)\end{aligned}$$

Then we apply equation (9):

$$\phi^{l+1} = \phi^l + \Delta t\left(\frac{\lambda\phi^l}{6} + \frac{1}{3}\lambda\left(\phi^l + \frac{\Delta t}{2}\lambda\phi^l\right) + \frac{1}{3}\lambda\left(\phi^l + \frac{\Delta t}{2}\lambda\left(\phi^l + \frac{\Delta t}{2}\lambda\phi^l\right)\right) + \frac{1}{6}\lambda\left(\phi^l + \Delta t\lambda\left(\phi^l + \frac{\Delta t}{2}\lambda\left(\phi^l + \frac{\Delta t}{2}\lambda\phi^l\right)\right)\right)\right)$$

After that we reduce terms:

$$\phi^{l+1} = \phi^l + \Delta t\left(\frac{\lambda\phi^l}{6} + \frac{\lambda\phi^l}{3} + \frac{\Delta t\lambda^2\phi^l}{6} + \frac{\lambda\phi^l}{3} + \frac{\Delta t\lambda^2\phi^l}{6} + \frac{\Delta t^2\lambda^3\phi^l}{12} + \frac{\lambda\phi^l}{6} + \frac{\Delta t\lambda^2\phi^l}{6} + \frac{\Delta t^2\lambda^3\phi^l}{12} + \frac{\Delta t^3\lambda^3\phi^l}{24}\right)$$

$$\phi^{l+1} = \phi^l + \Delta t\lambda\phi^l\left(1 + \frac{(\Delta t\lambda)^2}{2} + \frac{(\Delta t\lambda)^3}{6} + \frac{(\Delta t\lambda)^4}{24}\right)$$

$$\phi^{l+1} = \phi^l\left[1 + \Delta t\lambda\left(1 + \frac{(\Delta t\lambda)^2}{2} + \frac{(\Delta t\lambda)^3}{6} + \frac{(\Delta t\lambda)^4}{24}\right)\right]$$

$$\phi^{l+1} = \phi^l\left[1 + \Delta t\lambda + \frac{(\Delta t\lambda)^2}{2} + \frac{(\Delta t\lambda)^3}{6} + \frac{(\Delta t\lambda)^4}{24}\right]$$

And finally we get the expression:

$$\phi^l = \phi^0\sigma^l$$

Where σ represents the stability function of the fourth-order Runge-Kutta method:

$$\sigma = \left(1 + \Delta t\lambda + \frac{(\Delta t\lambda)^2}{2} + \frac{(\Delta t\lambda)^3}{6} + \frac{(\Delta t\lambda)^4}{24}\right)$$

In order to have stability we want $|\sigma| \leq 1$, we are going to evaluate the expression for σ at a number of points within the complex plane, so we can compute the magnitude of σ at each point, then extract a contour of $|\sigma| = 1$. We used a Matlab code to plot stability diagram as shown in Figure 1:

```
[X, Y] = meshgrid(-4:0.1:4, -4:0.1:4);
Z      = X + i*Y;
sigma  = abs(1 + Z + (Z.^2)/2 + (Z.^3)/6 + (Z.^4)/24);

% Plot the stability diagram
figure('WindowStyle', 'docked');
contourf(X, Y, sigma, [1 1],'-k');
hold on;
axis('equal', [-4 4 -4 4]);
```

```
grid on;
xlabel('\lambda_{Re}\Delta t');
ylabel('\lambda_{Im}\Delta t');
```

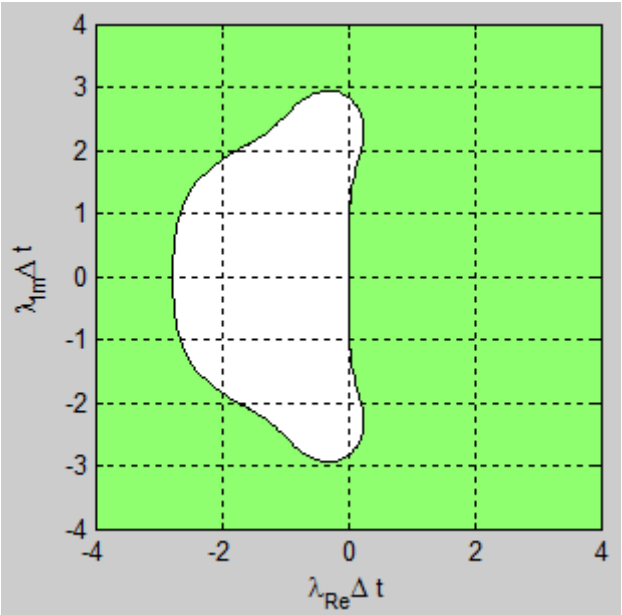


Figure 1. 4th order Runge-Kutta stability region.
The absolute stability regions are shown in white.
The ordinate and abscissa are $Im(\Delta t\lambda)$ and $Re(\Delta t\lambda)$ respectively.

In order to perform the error analysis we again consider the case where σ is purely imaginary and get the amplification factor into polar form as:

$$\sigma = \left(1 + \Delta t\lambda_{Im} + \frac{(\Delta t\lambda_{Im})^2}{2} + \frac{(\Delta t\lambda_{Im})^3}{6} + \frac{(\Delta t\lambda_{Im})^4}{24}\right) = Ze^{i\theta}$$

Where:

$$Z = \sqrt{\left(1 - \frac{(\Delta t\lambda_{Im})^2}{2} + \frac{(\Delta t\lambda_{Im})^4}{24}\right)^2 + \left(\Delta t\lambda_{Im} - \frac{(\Delta t\lambda_{Im})^4}{6}\right)^2}$$

And:

$$\theta = \tan^{-1}\left(\frac{4\Delta t\lambda_{Im}\left(6 - \Delta t^2{\lambda_{Im}}^2\right)}{\Delta t^4{\lambda_{Im}}^4 - 12\Delta t^2{\lambda_{Im}}^2 + 24}\right)$$

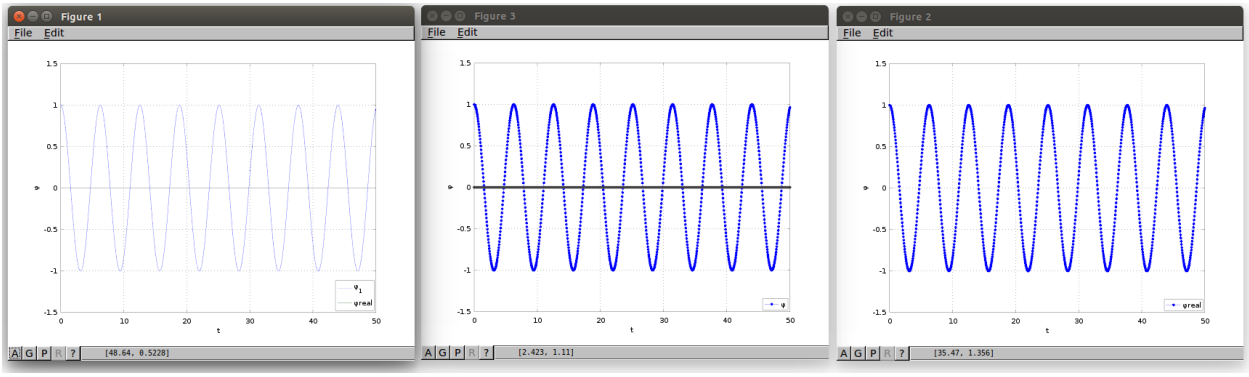


Figure 2. Phase and Amplitude errors

Figure 2 Shows the solution of the model problem with 4th order Runge-Kutta method. On the left side we can see the analytical solution and in the right side the numerical solution with 4RK. In the middle we combined both in order to compare results and we can see there's no a big difference between them.

2.4 Model Application

In order to solve the problem, we implemented a Matlab program using the Finite difference method with 2nd and 4th order central difference for spatial discretization and the 4th order Runge-Kutta method for temporal discretization. After getting the desired results, we translated it into a C++ program and finally we improved performance and efficiency by parallelizing it with OpenMP and MPI.

2.4.1 *Implementation in Matlab*

Let *phiVx*, *phiVy* and *phiH* be the arrays corresponding to the *Vx*, *Vy* and *h* fields from equations ***6.a***. We store the solution as a 2D array in Matlab as:

```
phiVx = zeros(N_x, N_y);
phiVy = zeros(N_x, N_y);
phiH  = zeros(N_x, N_y);
```

Where *N_x* and *N_y* represents the number of the grid points spaced Δx and Δy respectively.

We experimented with $\Delta t = 0.1$, $\Delta x = 0.5$ and $\Delta y = 0.5$. Then in order to actually compute the solution at each time step (using 4th order Runge-Kutta) we obtained the updated values of discretized phi as follows:

```
% Time marching loop
    for l=1:N_t-1

        [k1Vx, k1Vy, k1h]  = f(phiVx(:,:), phiVy(:,:), phiH(:,:));
        [k2Vx, k2Vy, k2h]  = f(phiVx(:,:) + Delta_t/2*k1Vx, phiVy(:,:) +
                           Delta_t/2*k1Vy, phiH(:,:) + Delta_t/2*k1h);
        [k3Vx, k3Vy, k3h]  = f(phiVx(:,:) + Delta_t/2*k2Vx, phiVy(:,:) +
                           Delta_t/2*k2Vy, phiH(:,:) + Delta_t/2*k2h);
        [k4Vx, k4Vy, k4h]  = f(phiVx(:,:) + Delta_t*k3Vx, phiVy(:,:) + Delta_t*k3Vy,
                           phiH(:,:) + Delta_t*k3h);
```

```
    phiVx(:,:)  =   phiVx(:,:) + Delta_t  *(k1Vx/6 + k2Vx/3 + k3Vx/3 + k4Vx/6);
    phiVy(:,:)  =   phiVy(:,:) + Delta_t  *(k1Vy/6 + k2Vy/3 + k3Vy/3 + k4Vy/6);
    phiH(:,:)   =   phiH(:,:) + Delta_t  *(k1h/6 + k2h/3 + k3h/3 + k4h/6);

end
```

We defined a function *f* to evaluate the right hand side of equations ***6.a*** at the various stages of the method. In our Matlab code, this took the form:

```
function [kVx,kVy,kH] = f(phiVx,phiVy,phiH)

    global g Delta_x Delta_y N_x N_y;

    kVx = zeros(N_x,N_y);
    kVy = zeros(N_x,N_y);
    kH = zeros(N_x,N_y);

    for i=1:N_x;
        for j=1:N_y;

            % Periodic Boundary condition
             ip1=i+1; ip2=i-1; jp1=j+1; jp2=j-1;

            if(i==N_x)
                  ip1=1;
            end
            if(i==1)
                ip2=N_x;
            end
            if(j==N_y)
                  jp1=1;
            end
            if(j==1)
                jp2=N_y;
            end

            % Function evaluation
            kVx(i,j) = (-g/(2*Delta_x)) * (phiH(ip1,j)-phiH(ip2,j)) -
                       (phiVx(i,j)/(2*Delta_x))*(phiVx(ip1,j)-phiVx(ip2,j)) -
                       (phiVy(i,j)/(2*Delta_y))*(phiVx(i,jp1)-phiVx(i,jp2));
            kVy(i,j) = (-g/(2*Delta_y)) * (phiH(i,jp1)-phiH(i,jp2)) -
                       (phiVx(i,j)/(2*Delta_x))* (phiVy(ip1,j)-phiVy(ip2,j)) -
                       (phiVy(i,j)/(2*Delta_y))*(phiVy(i,jp1)-phiVy(i,jp2));

           kH(i,j)  = (-phiVx(i,j)/(2*Delta_x))*(phiH(ip1,j)-phiH(ip2,j)) -
                       (phiH(i,j)/(2*Delta_x))*(phiVx(ip1,j)-phiVx(ip2,j)) -
                       (phiVy(i,j)/(2*Delta_y))*(phiH(i,jp1)-phiH(i,jp2)) -
                       (phiH(i,j)/(2*Delta_y))*(phiH(i,jp1)-phiH(i,jp2));
        end
    end
```

In the first part of this code snippet we apply the periodic boundary condition by assigning the values of the *i* and *j* steps in auxiliary variables ip1, ip2, jp1, jp2 in order to go through the appropriate array positions. This conditionals allow us to avoid redundancy of the code as well.

The new calculated values of *kVx, kVy* and *kH* will be returned by the *f* method in a 3 column vector and will help to update each part of the time marching loop repeatedly. Note that in each iteration the stored values of *phiVx, phiVy* and *phiH* will be overwritten.

Then, inside the time marching loop we draw the phiH array which will show us the animated simulation of the problem solution.

```
% Set up animated visualization of solution
    figure('WindowStyle', 'docked');
    colormap winter;
    axes;
    Solution = surf(x, y, phiH(:,:));
    ax=axis;
    grid on;
    xlabel('x');
    ylabel('y');
    zlabel('phiH');
    view([45 25]);


% time marching loop update of the figure
    set(Solution, 'ZData', phiH(:,:));
    axis(ax);
    title(['t = ' num2str(l+1)]);
    drawnow;
```

At this point, as we'll see in part 3, the results are distorted as time elapses when using the 2nd order central difference. In order to improve this weakness of the program we redefine our method f(phiVx,phiVy,phiH), using the 4th order central difference by using the ***7.a*** equations. It's not necessary to rewrite all the code, we only need to consider the boundary conditions since now we have to move along i+2, i-2, j+2 and j-1 through the grid points when the function is evaluated. The code snipped that shows this approach is:

```
% Function evaluation 4th order central differences
kVx(i,j)= (-g/(12*Delta_x)*(phiH(ip4,j)-8*(phiH(ip2,j))+8*phiH(ip1,j)-phiH(ip3,j))) -
      (phiVx(i,j)/(12*Delta_x)*(phiVx(ip4,j)-8*(phiVx(ip2,j))+8*phiVx(ip1,j)-
      phiVx(ip3,j))) - (phiVx(i,j)/(12*Delta_y)*(phiVx(i,jp4)-
      8*(phiVx(i,jp2))+8*phiVx(i,jp1)-phiVx(i,jp3)));

kVy(i,j)= (-g/(12*Delta_y)*(phiH(i,jp4)-8*(phiH(i,jp2))+8*phiH(i,jp1)-phiH(i,jp3))) -
      (phiVx(i,j)/(12*Delta_x)*(phiVy(ip4,j)-8*(phiVy(ip2,j))+8*phiVy(ip1,j)-
      phiVy(ip3,j))) - (phiVy(i,j)/(12*Delta_y)*(phiVy(i,jp4)-
      8*(phiVy(i,jp2))+8*phiVy(i,jp1)-phiVy(i,jp3)));

kH(i,j) = (-phiVx(i,j)/(12*Delta_x)*(phiH(ip4,j)-8*(phiH(ip2,j))+8*phiH(ip1,j)-
      phiH(ip3,j))) - (phiH(i,j)/(12*Delta_x)*(phiVx(ip4,j)-
      8*(phiVx(ip2,j))+8*phiVx(ip1,j)-phiVx(ip3,j))) -
      (phiVy(i,j)/(12*Delta_y)*(phiH(i,jp4)-8*(phiH(i,jp2))+8*phiH(i,jp1)-
      phiH(i,jp3))) - (phiH(i,j)/(12*Delta_y)*(phiVy(i,jp4)-
      8*(phiVy(i,jp2))+8*phiVy(i,jp1)-phiVy(i,jp3)));
```

Where auxiliary variables ip1, ip2, ip3, ip4, jp1, jp2, jp4 and jp4 help us to go through the appropriate array positions when reaching the borders. The animation results obtained with these approaches are shown in Appendix A and Appendix B.

2.4.2 *Implementation in C++*

Once we have developed the Matlab version of the program, we translated it to C++. The only difference between them is that in C++ we have to work with memory allocation, implement array arithmetic operations, and use Paraview as a tool for visualization.

In the serial and MPI versions of the solution we represent *PhiH*, *PhiVx* and *PhiVy* matrixes as 2D array dynamically allocated:

```
double** phiVx = new double* [N_x];
double** phiVy = new double* [N_x];
double** phiH  = new double* [N_x];

        for(i=0; i<N_x; i++)
        {
                tempPhiVx[i]   = new double[N_y];
                tempPhiVy[i]   = new double[N_y];
                tempPhiH[i]    = new double[N_y];
                …
        }
```

We also have to use temporal 2D arrays to store the intermediate matrix operations of Runge-Kutta:

```
double** tempPhiVx    = new double* [N_x];
double** tempPhiVy    = new double* [N_x];
double** tempPhiH     = new double* [N_x];
```

The overall serial algorithm implementation is similar to Matlab's implementation with an initial parameter assignment, a time marching loop with Runge-Kutta implementation and a Function *f* with the 4th order finite difference space discretization. As in C++ there are not matrix operators we have to implement matrix manipulation using loops and temporal arrays:

```
…
        f(k1Vx, k1Vy, k1H, phiVx, phiVy, phiH);
        for(i=0; i<N_x; i++)
        {
                for(j=0; j<N_y; j++)
                {
                        tempPhiVx[i][j]= phiVx[i][j] + (Delta_t/2)*k1Vx[i][j];
                        tempPhiVy[i][j]= phiVy[i][j] + (Delta_t/2)*k1Vy[i][j];
                        tempPhiH[i][j] = phiH[i][j] + (Delta_t/2)*k1H[i][j];
                }
        }

        f(k2Vx, k2Vy, k2H, phiVx, phiVy, phiH);

        for(i=0; i<N_x; i++)
        {
                for(j=0; j<N_y; j++)
                {
                        tempPhiVx[i][j]= phiVx[i][j] + (Delta_t/2)*k2Vx[i][j];
                        tempPhiVy[i][j]= phiVy[i][j] + (Delta_t/2)*k2Vy[i][j];
                        tempPhiH[i][j] = phiH[i][j] + (Delta_t/2)*k2H[i][j];
                }
        }

….
```

In order to visualize results, we print a single .csv file *per time step* with each value of *phiH*. These files can be opened with Paraview. The animation results obtained with this approach are shown in Appendix C.

```
void    write(fstream& file, double** phiH, double** phiVx, double** phiVy, int N_x, int N_y,
int t){
        for(int i=0; i<N_x; i++){
                double coordX = i*Delta_x;
                for(int j=0; j<N_y; j++){
                        double coordY = j*Delta_y;
                        file << std::fixed << std::setprecision(4) << phiH[i][j] <<"," << coordX
<< "," << coordY<< "\n"; //multiple files
                }
        }
        return;
}
```

The visualization results are shown in Appendix C.

2.4.2 Implementation in C++ using OpenMP

Open Multi Processing or OpenMP is an API that supports shared memory parallel computation by using a multi-threaded approach. The power of OpenMP is that it can be implemented quickly on the traditional serial code by the inclusion of a set of Pragma Clauses (compiler directives) that specify the behavior of the threads. A thread is an independent flow of control or can be seen also as the minimum runtime entity created to execute a sequence of instructions. Normally a CPU can run 1 or 2 threads on its hardware. OpenMP uses a model of Thread Fork-Join as the shown in Figure 3.

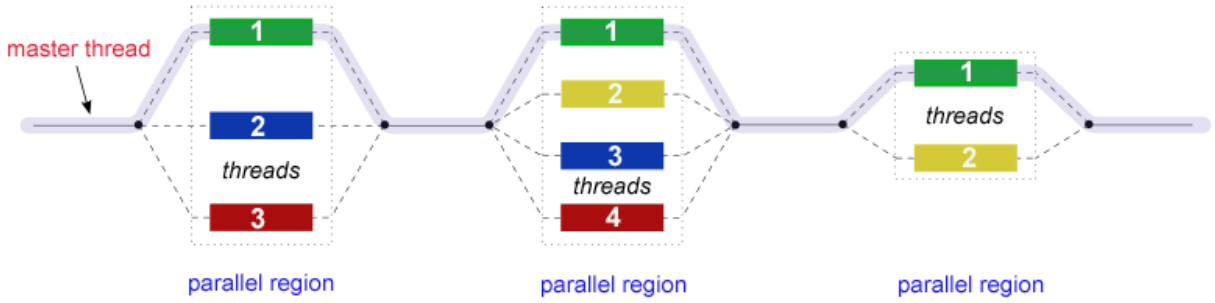


Figure 3. OpenMP Fork/Join mode (Wikipedia N.)

Numerical methods implementations can take advantage of OpenMP by using a smart parallelization of the loops required to calculate the different stages and stencils. For example in a 1D problem solved with Runge-Kutta a loop must be implemented to calculate Phi(l+1):

```
for(e=0; e<N_e; e++){
        phi[l+1][e] = phi[l][e] + Delta_t*(k1[e]/6 + k2[e]/3 + k3[e]/3 + k4[e]/6);
}
```

A multithreaded OpenMP implantation can schedule threads in different ranges between 0 and N_e. For example for a N_e of 1000 and 4 threads, OpenMP can break the problem in the following way:

Figure 4. OpenMP Loop scheduling

Implementation of multithreading using OpenMP in our C++ program consists on adding OpenMP clauses to the serial version. The strategy for parallelization is to create a Parallel section that encloses the Time Marching Loop and a set of '*omp for*' clauses for the matrix operations and grid discretization. The lines added are illustrated as follows:

```
double wtime  = omp_get_wtime(); //to get the time per time step
int N_Threads= omp_get_max_threads(); //to get the num of threads

#pragma omp parallel default(shared) private(i, l, j)
      {
            // Time marching loop
            for(l=0; l<N_t-1; l++)
            {
                  #pragma omp single
                  {
                     t += Delta_t;
                  }
                  f(k1Vx, k1Vy, k1H, phiVx, phiVy, phiH);

                  #pragma omp for schedule(static)
                  for(i=0; i<N_x; i++)
                  {
                        for(j=0; j<N_y; j++)
                        {
                        tempPhiVx[i][j]     = phiVx[i][j] + (Delta_t/2)*k1Vx[i][j];
                        tempPhiVy[i][j]     = phiVy[i][j] + (Delta_t/2)*k1Vy[i][j];
                        tempPhiH[i][j]      = phiH[i][j] + (Delta_t/2)*k1H[i][j];
                        }
                  }…. //The same apply for K2, K3 and K4…

            #pragma omp single
            {
            // Write the solution
            sprintf(myFileName, "Assignment1_Files_OMP/Shallow_Water_%d.csv", l+1);
            file.open(myFileName, ios::out);
            write(file, phiH, phiVx, phiVy, N_x, N_y, t+1);
            file.close();
            }
```

Also the *f* space discretization method does implement a '*#pragma omp for*':

```
void   f(double** kVx, double** kVy, double** kH, double** phiVx, double** phiVy,
double** phiH)
```

```
{
        int ip1, ip2, ip3, ip4, jp1, jp2, jp3, jp4;
        #pragma omp for
        for(int i=0; i<N_x; i++){
 ...
```

The visualization results are shown in Appendix D.

2.4.3 Implementation in C++ using MPI

MPI (Message Passing Interface) is a specification for message interchange between different processes that usually runs on different machines. A process can be seen as a runtime entity with an exclusive and independent assigned memory. (OpenMPI, 2014)

As memory is not shared between processes and/or machines this specification standardized all the protocols to distribute data and coordinate process and therefore allows a set of process to work in parallel to solve computational problems (OpenMP.org, 2013).

MPI can be used in solving numerical problems by partition the problem execution across different nodes. This typically involves a clever partition and data sending mechanism across processes and at the end a reduction of the calculated results of each process. For example in a 1D problem, the data structure that represents the space discretization can be partitioned across 4 processes that use MPI to exchange the Phi(x) and Phi(x) as shown in Figure 5.

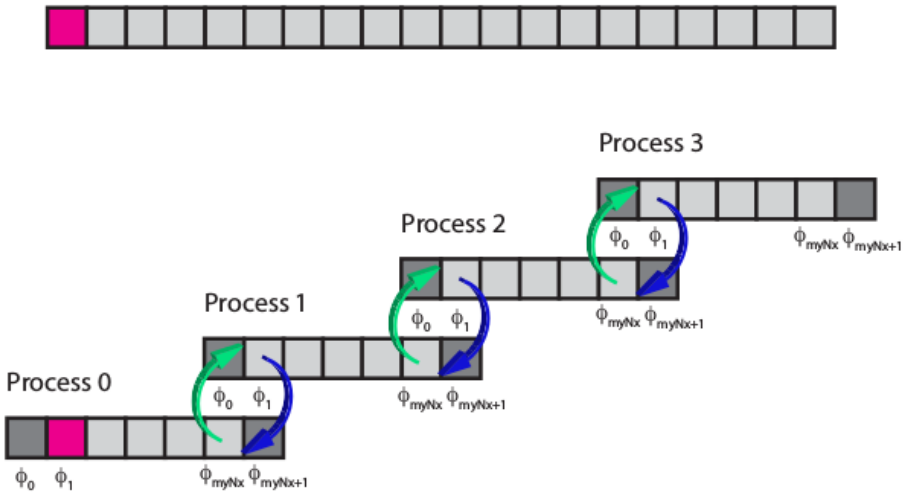


Figure 5. Communication between processes (Moore, Applied Numerical Methods, 2013)

OpenMPI is an open source implementation of MPI and is the implementation we used for the development of the MPI parallelization. The implementation using MPI differs a little bit in terms of coding from OpenMP and serial implementation because MPI requires:

- ***Call different set of functions to initialize and finalize MPI:*** In the code we must call functions MPI_Init, Comm_size, Comm_rank, Cartesian Communicator creation and MPI Finalize. The Cartesian Communicator is a MPI communicator that allows us to use

processes in the computation with a Cartesian topology (Top, bottom, right and left neighbors). Figure 6 represents a Cartesian 2D 3x3 communicator (9 processes). Following is the relevant code:

```
…
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &N_Procs);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);
…
// Map the processors to the N x N grid, determine the new rank and determine the
neighboring processes
MPI_Cart_create(MPI_COMM_WORLD, N_D, dimensions, isPeriodic, reorder, &Comm2D);
MPI_Comm_rank(Comm2D, &myID);
MPI_Cart_coords(Comm2D, myID, N_D, myCoords);
MPI_Cart_shift(Comm2D, X, 1, &leftNeighbor,   &rightNeighbor);
MPI_Cart_shift(Comm2D, Y, 1, &bottomNeighbor, &topNeighbor);

…
```

- ***An explicit strategy for data partitioning:*** Data division across processes must be implemented. For our implementation we decided to segment the problem by assigning a portion of the Grid to each process. Figure 6 shows how the total grid is partitioned across a 9 processes in a Cartesian topology.

  In the MPI implementation we chose to fix the number of cells that each process will work on, regardless of the number of process. With this approach when we add processes to the MPI computation we are using a smaller delta X and delta Y.

- ***Data Exchange:*** As in our simulation where are using a 4^{th} difference method, we need to interchange data from i±1 and i±2 points between processes in order to calculate the finite difference. As this problem is 2D we need to interchange a set of cells (row or column) instead of a single value. To facilitate this exchange we add a double halo or set of extra cells that surrounds the grid. Figure 6 illustrates data exchanging between process 1,1 and its neighbors.

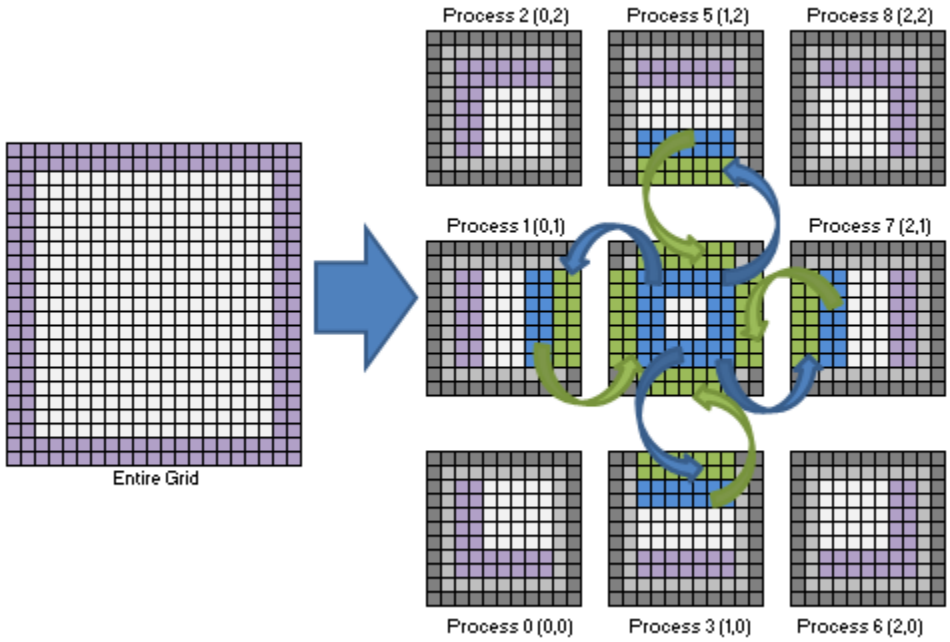


Figure 6. Cartesian communicator, data partitioning and exchanging (Moore, Applied Numerical Methods, 2013)

The relevant parts of the code are the following:

```
double** phiVx      = new double* [myN_x+4]; //Nx+4 because of the Halo
double** phiVy      = new double* [myN_x+4];
double** phiH       = new double* [myN_x+4];

…

// Time marching loop
       for(l=0; l<N_t-1; l++){
       t += Delta_t;

              // Exchange data
              exchange(phiH, phiVx, phiVy , myN_x, myN_y, myID, rightNeighbor, leftNeighbor,
topNeighbor,  bottomNeighbor, Comm2D, strideType);
              f(k1Vx, k1Vy, k1H, phiVx, phiVy, phiH, myN_x, myN_y,Delta_x, Delta_y,1);

              for(i=2; i<=myN_x+1; i++){
                     for(j=2; j<=myN_y+1; j++){
                            tempPhiVx[i][j]       = phiVx[i][j] + (Delta_t/2)*k1Vx[i][j];
                            tempPhiVy[i][j]       = phiVy[i][j] + (Delta_t/2)*k1Vy[i][j];
                            tempPhiH[i][j] = phiH[i][j] + (Delta_t/2)*k1H[i][j];
                     }
              }

              // Exchange data
              exchange(tempPhiH, tempPhiVx, tempPhiVy , myN_x, myN_y, myID, rightNeighbor,
leftNeighbor, topNeighbor,  bottomNeighbor, Comm2D, strideType);
       f(k2Vx, k2Vy, k2H, tempPhiVx, tempPhiVy, tempPhiH, myN_x, myN_y,Delta_x, Delta_y,2);
```

```
            for(i=2; i<=myN_x+1; i++){
                  for(j=2; j<=myN_y+1; j++){
                        tempPhiVx[i][j]= phiVx[i][j] + (Delta_t/2)*k2Vx[i][j];
                        tempPhiVy[i][j]= phiVy[i][j] + (Delta_t/2)*k2Vy[i][j];
                        tempPhiH[i][j] = phiH[i][j] + (Delta_t/2)*k2H[i][j];
                  }
            }


            // Exchange data
            exchange(tempPhiH, tempPhiVx, tempPhiVy , myN_x, myN_y, myID, rightNeighbor,
leftNeighbor, topNeighbor,  bottomNeighbor, Comm2D, strideType);
            f(k3Vx, k3Vy, k3H, tempPhiVx, tempPhiVy, tempPhiH, myN_x, myN_y,Delta_x,
Delta_y,3);

            for(i=2; i<=myN_x+1; i++){
                  for(j=2; j<=myN_y+1; j++){
                        tempPhiVx[i][j]= phiVx[i][j] + Delta_t*k3Vx[i][j];
                        tempPhiVy[i][j]= phiVy[i][j] + Delta_t*k3Vy[i][j];
                        tempPhiH[i][j] = phiH[i][j] + Delta_t*k3H[i][j];
                        if (l==0 && i==14 && j==58)
                  }
            }


            …
```

It is important to mention that with the Halo strategy there is no need to enforce special code for the Boundary condition, so the *f* function will go through the for loop as follows:

```
for(int i=2; i<=myN_x+1; i++){
      for(int j=2; j<=myN_y+1; j++){

      ip1=i+1;ip2=i-1;ip3=i+2;ip4=i-2;jp1=j+1;jp2=j-1;jp3=j+2;jp4=j-2;

    kVx[i][j]= (-g/(12*Delta_x)*(phiH[ip4][j]-8*(phiH[ip2][j])+8*phiH[ip1][j]-…
    kVy[i][j]= (-g/(12*Delta_y)*(phiH[i][jp4]-8*(phiH[i][jp2])+8*phiH[i][jp1]-…
    kH[i][j] = (-phiVx[i][j]/(12*Delta_x)*(phiH[ip4][j]-…
      }

}
```

- ***Contiguous Memory:*** OpenMPI only cares about exchanging Bytes from memory. When data in an array is exchanged the developer instructs MPI to send X values of N bytes in memory and provides a pointer to it. However this can be achieved only if the array is stored contiguously in memory. With the 2D array creation approach of the serial and OpenMP code we would have each row of the array separated in memory. Figure 7 demonstrates this approach.

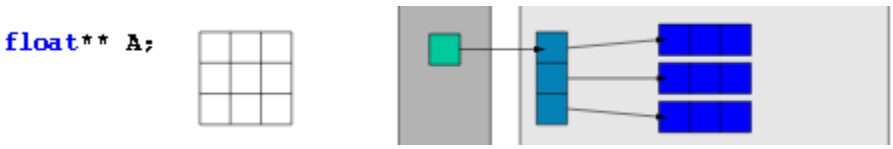


Figure 7. Traditional memory allocation (Moore, Applied Numerical Methods, 2013)

For ease of use in data exchange we want all the cells to be contiguous in memory. To accomplish this we create an array of size [MxN], where M are the number of rows and N the numbers of columns. Then we set the pointers of the 2D array to the proper cell within this array. Figure 8 illustrates this approach.

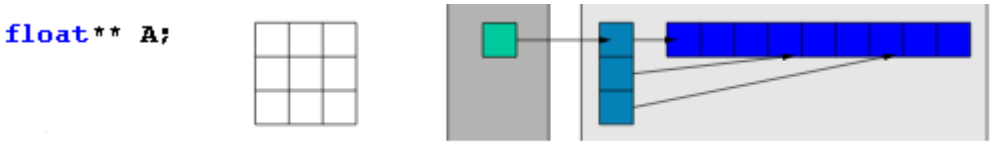


Figure 8. 2D array contiguous memory allocation (Moore, Applied Numerical Methods, 2013)

With this strategy all column's cells are contiguous in memory and while all row's cell are not contiguous there is a constant number of bytes in memory between each row cell.

The following code illustrates this approach:

```
double** phiH = new double* [myN_x+4];
MPI_Datatype strideType;
....
phiH[0] = new double [(myN_x+4)*(myN_y+4)];

  //Divide the array in a 2D way
    for(int i=1, ii=myN_y+4; i<myN_x+4; i++, ii+=(myN_y+4))
      {
            …
            phiVx[i] = &phiVx[0][ii];
            phiVy[i] = &phiVy[0][ii];
            phiH[i]       = &phiH[0][ii];
            …
      }

…

// Create a new datatype to store values on an x boundary
MPI_Type_vector(myN_x, numElementsPerBlock, myN_y+4, MPI_DOUBLE, &strideType);
MPI_Type_commit(&strideType);

…
```

Data I/O: In order to use the solution data of the program we need a way to write the information in a file. Because I/O is not parallelizable and each process has a subset of the grid it is important to have an I/O strategy. We decided that each process should write in its

own independent file the result at every time step, and then post process all the files generating one sorted file per time step. Following is the write function of each MPI process, note that we are printing the time step in the file:

```
…

// Write the solution
sprintf(myFileName, "Assignment1_Files_MPI/Shallow_Water_MPI_%d.csv", myID);
file.open(myFileName, ios::out);
write(file, phiH, phiVx, phiVy, myN_x, myN_y, 0, myCoords,Delta_x, Delta_y);

…
```

The write function is defined as follows:

```
void write(fstream& file, double** phiH, double** phiVx, double** phiVy, int N_x, int N_y, int
t, int* myCoords, double Delta_x, double Delta_y){
       for(int i=2; i<=N_x+1; i++)  {
           double coordX = (myCoords[X]*N_x + (i-2))*Delta_x;
              for(int j=2; j<=N_y+1; j++)         {
                     double coordY = (myCoords[Y]*N_y + (j-2))*Delta_y;
                     if (t==1 && coordX==6 && coordY==28 )
                            file << t <<","<< std::fixed << std::setprecision(10) << coordX
                            << "," << coordY<< "," << phiH[i][j] <<"\n";
              }
       }
       return;
}
```

After finishing the execution of the program, we will have several .csv files as output. In order to visualize results in Paraview we need to split those files in multiple single time step files. For this reason we use the following batch script to process all files:

```
#!/bin/bash
rm -f Assignment1_Files_MPI/*
for i in $(ls Shallow_Water_MPI.csv.*); do
    awk -F, '{print $2","$3","$4 >> "Assignment1_Files_MPI/Shallow_Water_MPI."$1".csv.tmp"}'
$i
done
cd Assignment1_Files_MPI
for i in $(ls Shallow_Water_MPI*.csv.tmp); do
    name=$(echo $i | cut -d "." -f 1,2)
    sort -t\, -n -k 1,1n -k 2,2n -k 3,3n -k 4,4n $i > $name".csv"
done
cd ..
rm -f Assignment1_Files_MPI/*.csv.tmp
```

The visualization results are shown in Appendix E.

3. Results

One of the main goals of this assignment is to test the scalability and speedup of the implemented programs. We have 3 versions of our C++ program so far: The sequential and the two parallelized versions using OpenMP and MPI. Also we have the first approaches using Matlab. All the results of the Matlab programs are shown in Appendix A and Appendix B.

In order to show the obtained performance results we ran our programs on a Laptop computer (2 cores, 8 threads) and on the BlueGeneQ supercomputer (Avoca).

Saying that N is the amount of computation a parallel program has to do to solve a certain problem, the program's running time T depends on the problem size N and on the number of threads/processes K. Strong scaling is where we keep N the same while increasing K and Weak scaling is where we increase both N and K proportionally. We tested the performance of the OpenMP version using strong scaling and weak scaling for the MPI version.

- We ran the OpenMP version on our **Laptop** with parameters: Delta_x = Delta_y = 0.3, Delta_t=0.05. As we can see in Chart 1. The performance improves 9.7% with 2 threads in comparison to the one thread version and by increasing the number of threads we got the best result which was with 4 threads. We removed the file I/O (which is non-parallelizable and computationally intensive) from the code and the results presented a better response improving until we get our best result with 8 threads as shown in Chart 2.

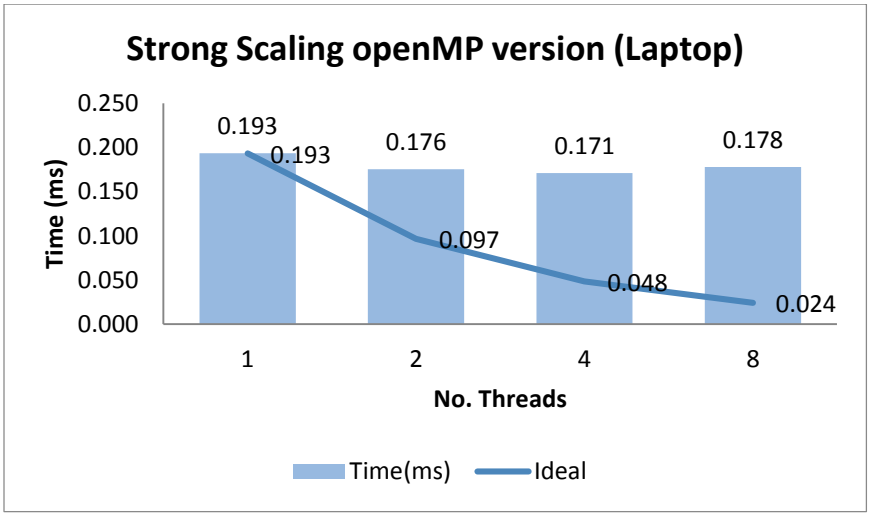


Chart 1. Strong scaling on Laptop

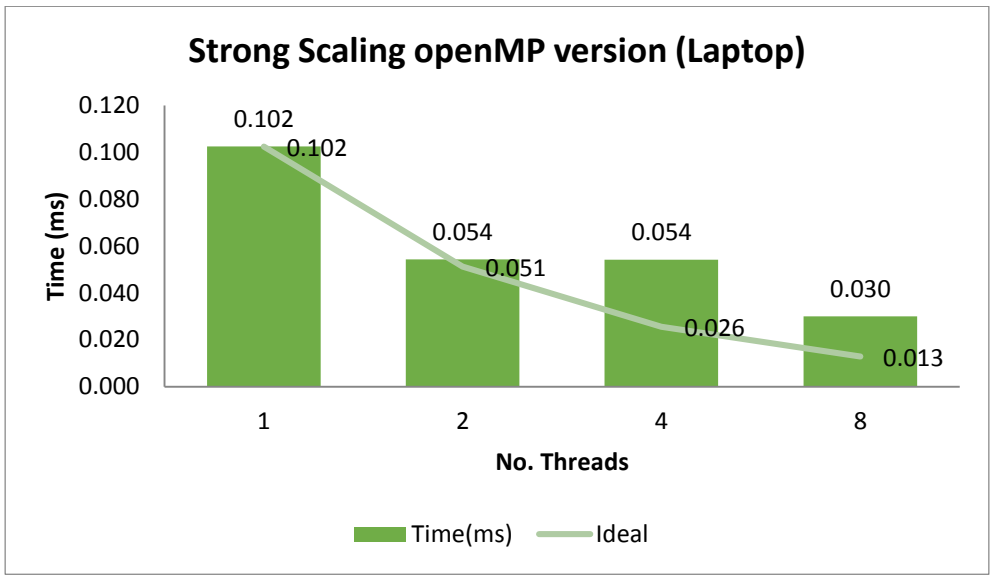


Chart 2. Strong scaling on the Laptop (with no I/O)

- We ran the OpenMP version on the **BlueGeneQ** with parameters: Delta_x = Delta_y = 0.1, Delta_t=0.02 (That is a grid size of 1000x1000). As we can see in Chart 3. the performance improves 25% with 2 threads in comparison to the one thread version and by increasing the number of threads we got the best result which was with 16 threads with an important improvement of 333% .

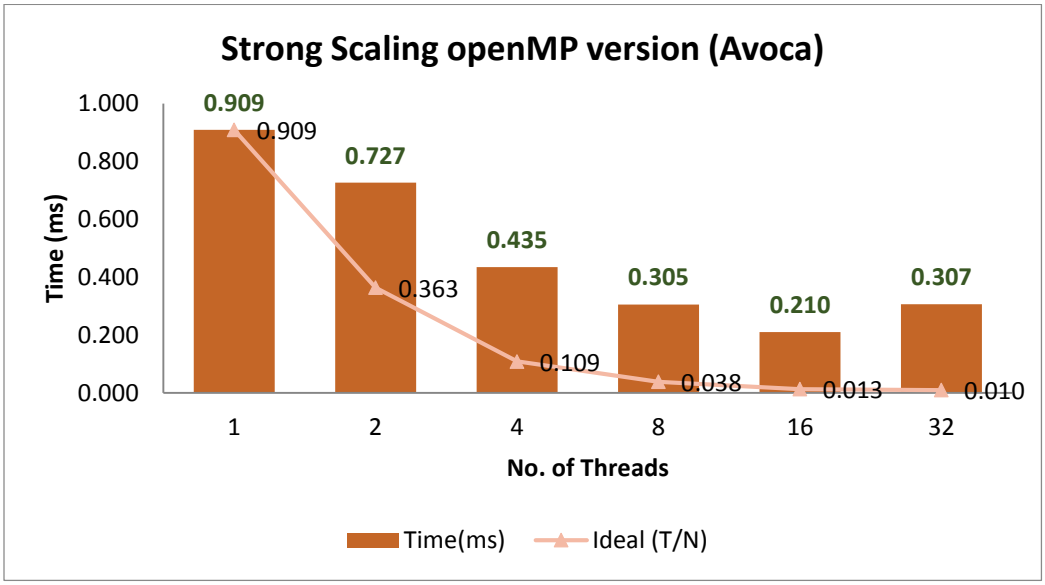


Chart 3. Strong scaling on the BlueGeneQ (Avoca)

- We ran the MPI version on the **BlueGeneQ** with the following parameters: Number of grid points per process 1000x1000 and Delta_t = 0.4. We also commented the IO parts in order to focus only in the MPI efficiency. Chart 4 displays the result of the weak scaling test, note that the time is similar while we increase the problem size and number of processors:

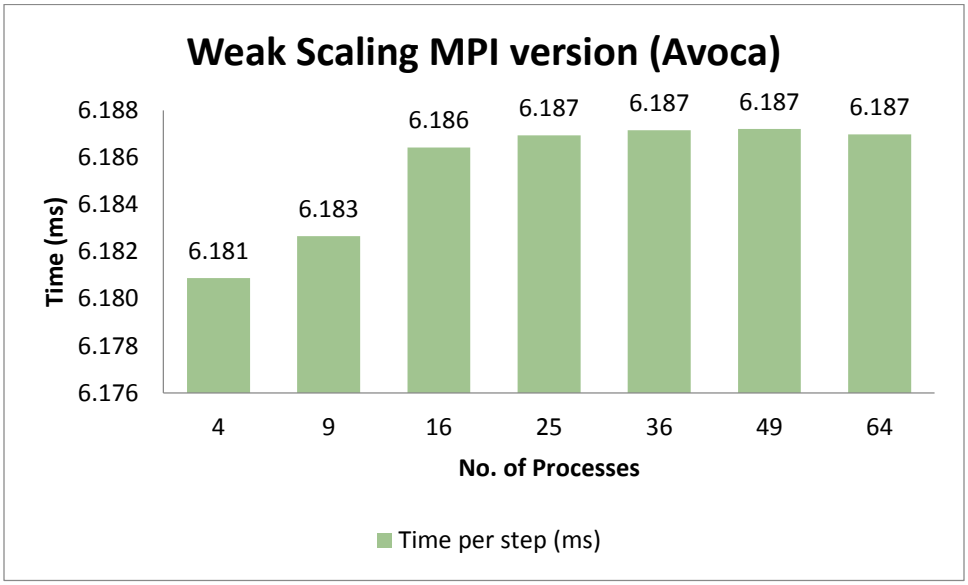


Chart 4. Weak scaling on the BlueGeneQ (Avoca)

4. Discussion

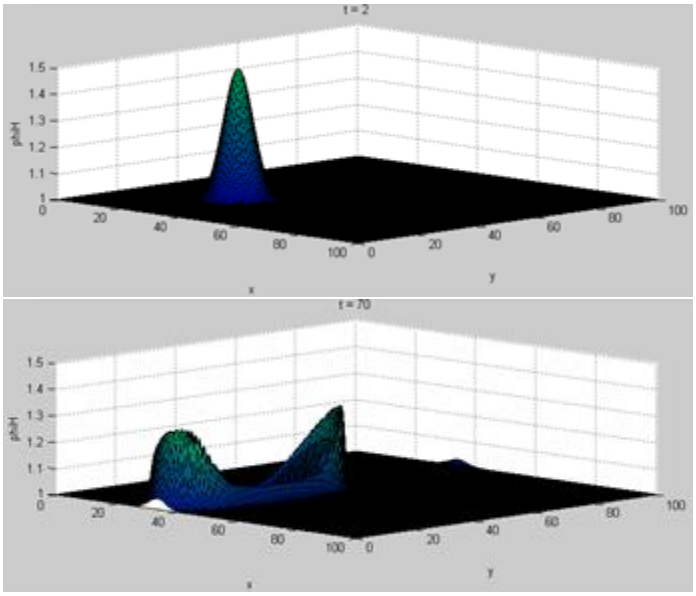
The main outcomes of the performed tests were:

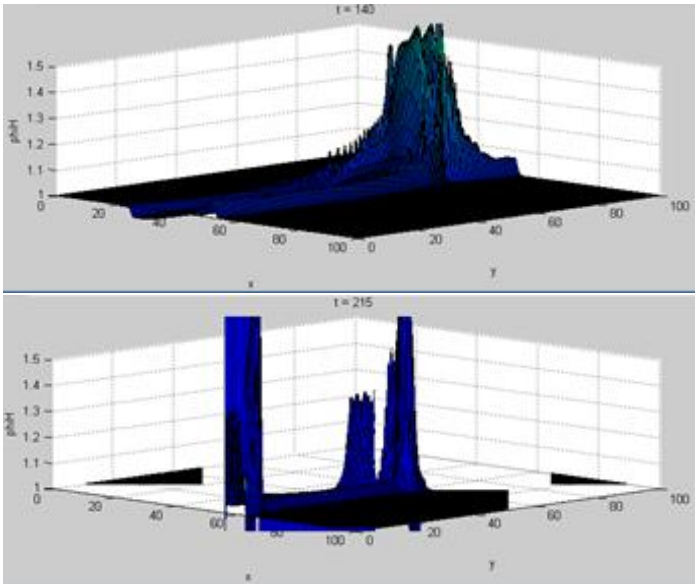
- Utilizing numerical methods for solving differential equations allow us to model and solve complex real world problems.
- Developing the Matlab versions of the program allowed us to get our first idea on how to model the solution of the problem (logic and graphic animation), then translating it to C++ version was pretty straightforward.
- The 4th order finite difference method with 4th order Runge-Kutta method, showed better approach of the solution in comparison to the 2nd order finite difference method. The last one simply 'blowed up' after some time steps.
- We found (by trial and error) the most stable parameters for the simulation was: Delta_x = Delta_y = 0.5 and Delta_t between 0.05 and 0.1. So we used them in our C++ versions of the code as the 'default' parameters for testing them with Paraview.
- The use of OpenMP in the code allowed us to have a better understanding on how to convert a program into a multithreaded one and easily improve performance. OpenMP allows a quick implementation of multithreaded parallelization for scientific purposes.

- The use of MPI allowed us to have a better understanding on how to dynamically allocate memory, parallelize and communicate a multiprocessor program and improve performance of programs that solve big size problems.
- The strong scaling results show how the parallelized version of the program can improve speed up to a 333%.
- With the weak scaling chart we can see that linear scaling is achieved because the run time stays constant while the workload is increased in direct proportion to the number of processors.
- I/O is an important issue when executing a parallel program. Depending on the size problem it can become critical with a direct impact to the performance of the execution. That's the reason we omitted that part for the tests performed on the BlueGeneQ, otherwise the results wouldn't scale well.
- Debugging and finding bugs in OpenMP and especially in OpenMPI represents a challenge because the existence of race conditions and also because it is not easy to follow the parallel/distributed processing.

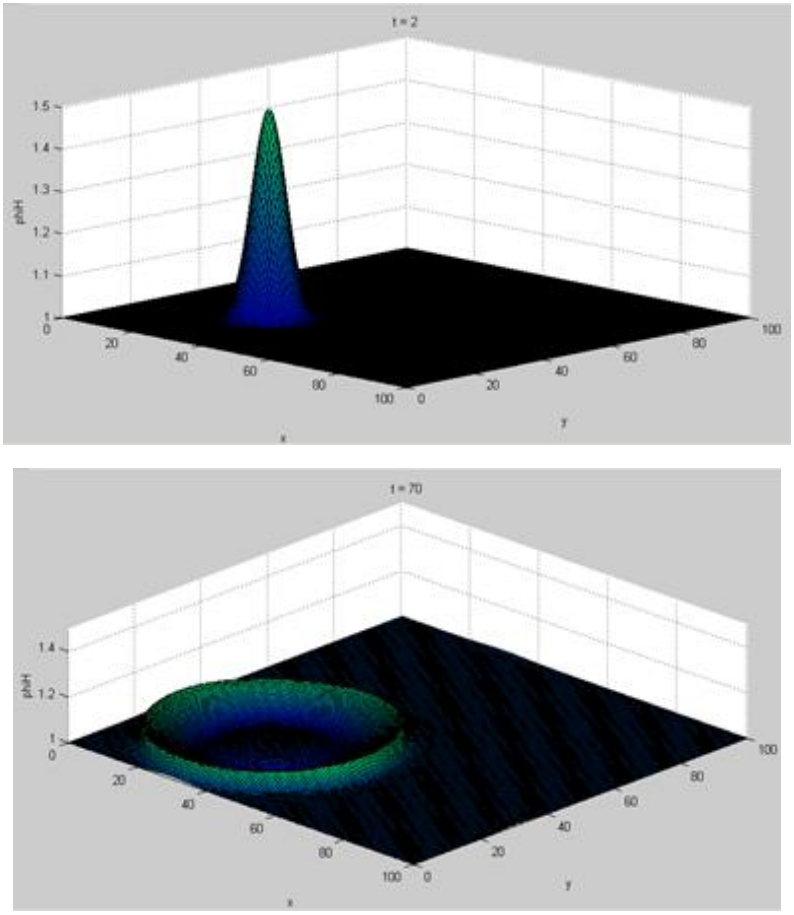
5. Appendices

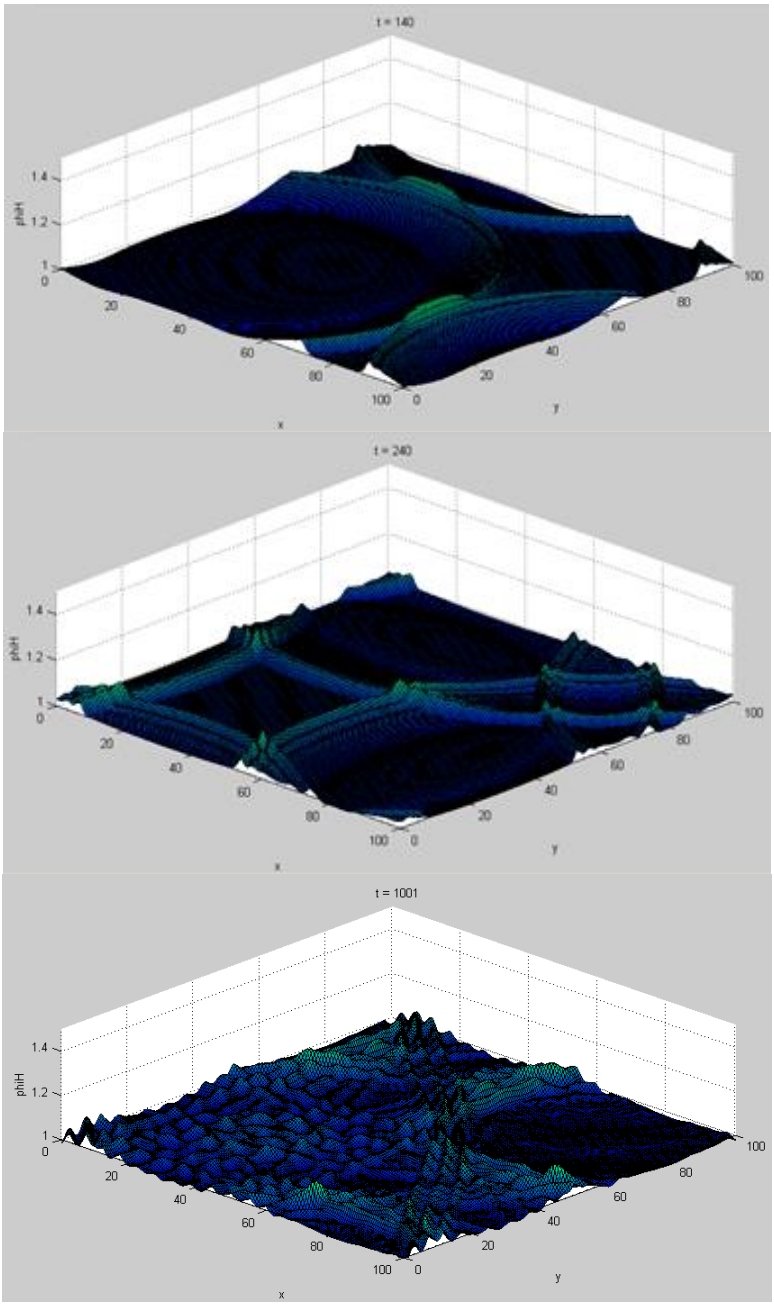
Appendix A. Matlab Animation of the solution using 2nd order finite difference method with 4th order Runge-Kutta.



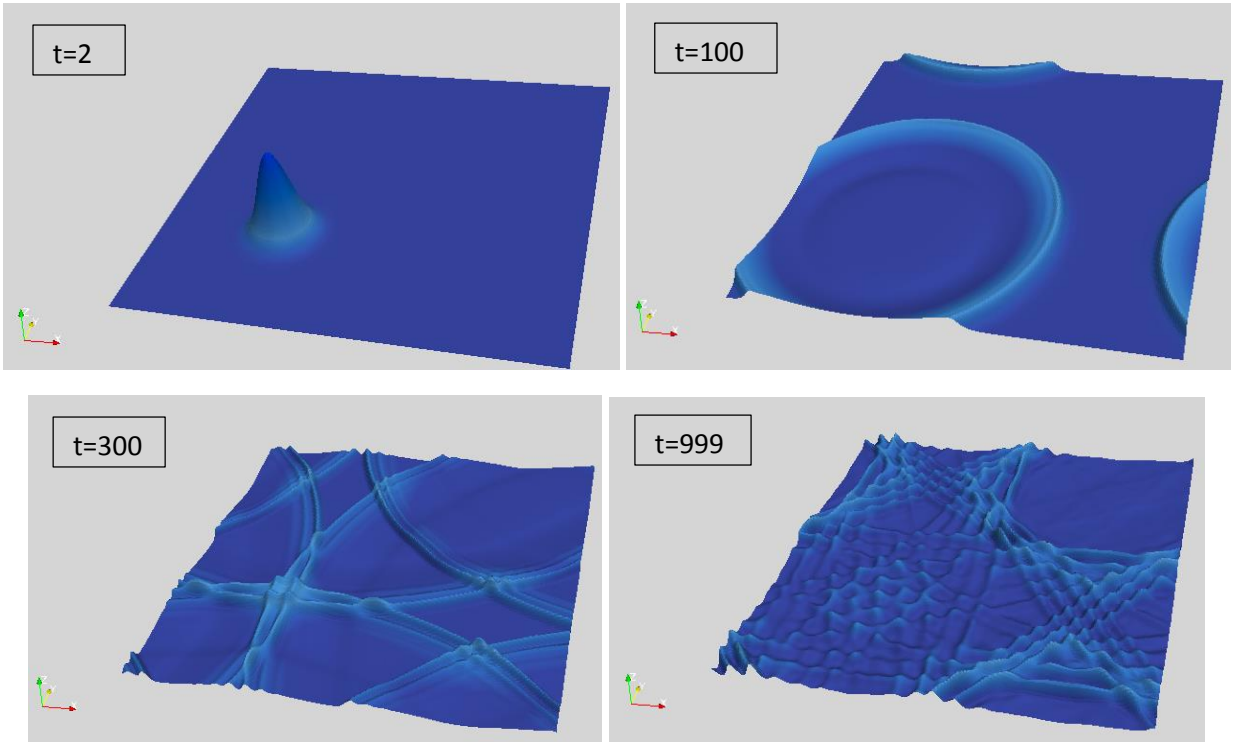


Appendix B. Matlab Animation of the solution using 4th order finite difference method with 4th order Runge-Kutta.

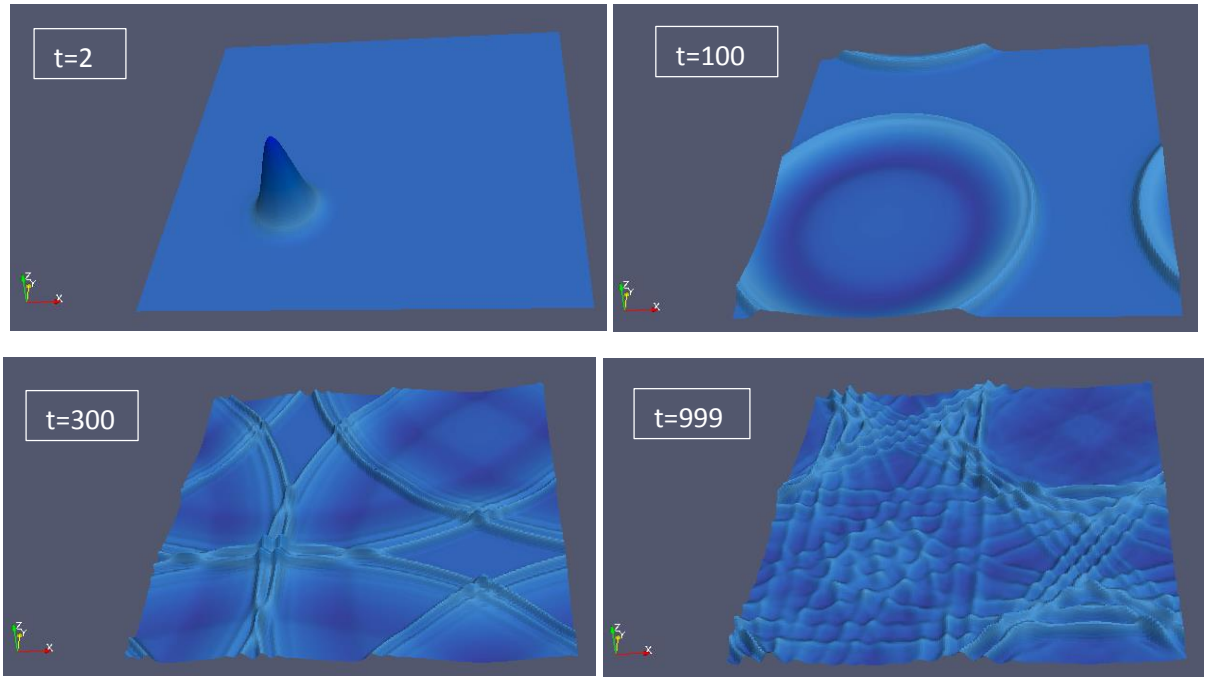




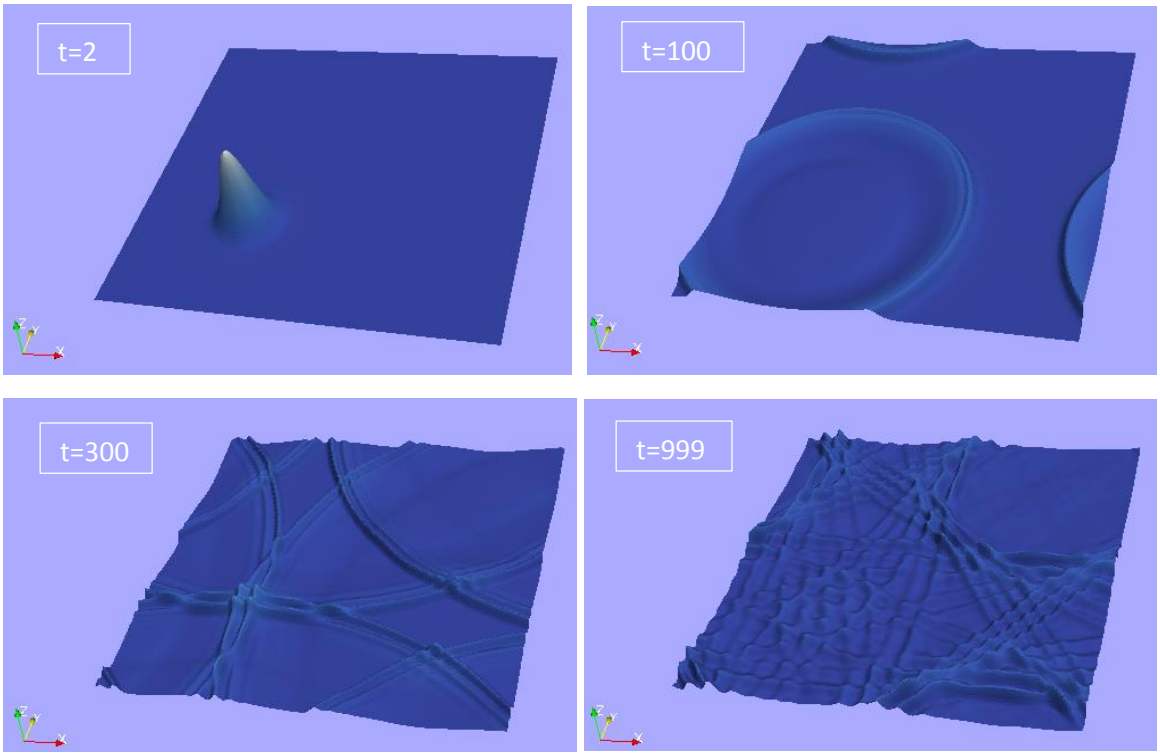
Appendix C. Paraview Animation of the solution using 4th order finite difference method with 4th order Runge-Kutta. (Serial version of the c++ Program)



Appendix D. Paraview Animation of the solution using 4th order finite difference method with 4th order Runge-Kutta. (openMP version of the c++ Program)



Appendix E. Paraview Animation of the solution using 4th order finite difference method with 4th order Runge-Kutta. (MPI version of the c++ Program)



6. References

Canale, S. C., & Chapra, R. P. (2010). *Numerical Methods for Engineers.* New York: McGraw-Hill.

Frank, J. (n.d.). *Numerical Modelling of Dynamical Systems.* Retrieved from Stability of Runge Kutta Methods: http://www.staff.science.uu.nl/~frank011/Classes/numwisk/ch10.pdf

Mattson, T., & Chapman, B. (n.d.). *OpenMP in Action.* Retrieved from http://openmp.org/wp/presos/omp-in-action-SC05.pdf

Moore, S. (2013). *Applied Numerical Methods.* The University of Melbourne.

Moore, S. (2014). *Assignment 1, Applied High Performance Computing.* University of Melbourne.

OpenMP.org. (2013, 8 14). *OpenMP*. Retrieved from http://openmp.org/wp/about-openmp/

OpenMPI. (2014). *Open Source High Performance Computing.* Retrieved from http://www.open-mpi.org/

Wikipedia. (2014). *The Shallow Water Equations.* Retrieved from http://en.wikipedia.org/wiki/Shallow_water_equations

Wikipedia. (n.d.). *Moore's law.* Retrieved from http://en.wikipedia.org/wiki/Moore%27s_law

Wikipedia, N. (n.d.). *OpenMP http://en.wikipedia.org/wiki/OpenMP.*