MCEN90031 Applied High Performance Computing
Assignment 1

The Shallow Water Equations

Due date 29th September, 2017

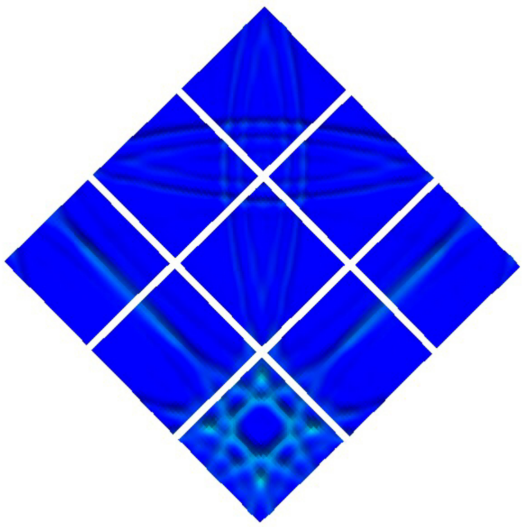


Figure 1: Example solution of Shallow Water equation with MPI

Name	ID
Li Lindong	795501
Zheng Bujingda	838092

Contents

1 Introduction

1.1 Purpose

The aim of the assignment is to obtain the numerical solution of the shallow water equation by MATLAB and C++ program which applied fourth order Runge-Kutta method and sixth order central finite difference stencil. After getting numerical solution which is calculated data from the programmes, applying proper visualization by MATLAB and Paraview are also important to show the work result of program.

For C++ program, two parallelizing method will be applied, including OpenMP and MPI. The principle of two methods are different, and for OpenMP, it shares and arranges memory among threads to let various parts calculated separately and simultaneously. For MPI, it is actually a method which redesigns and distributes the memory for C++ program while it is more complicated in structure but has more efficient optimization than OpenMP method.

1.2 Governing Equation

From the assignment questions we got the governing equation and we need both vector notation and fully expanded to present the governing equations.

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v} = -g \cdot \nabla \tag{1.2.1}$$

$$\frac{\partial h}{\partial t} + \nabla \cdot \boldsymbol{v} h = 0 \tag{1.2.2}$$

And we can expand them fully as follow,

$$\frac{\partial v_x}{\partial t} = -v_x \frac{\partial v_x}{\partial x} - v_y \frac{\partial v_x}{\partial y} - g \frac{\partial h}{\partial x} \tag{1.2.3}$$

$$\frac{\partial v_y}{\partial t} = -v_x \frac{\partial v_y}{\partial x} - v_y \frac{\partial v_y}{\partial y} - g \frac{\partial h}{\partial y} \tag{1.2.4}$$

$$\frac{\partial h}{\partial t} = -\frac{(\partial v_x h)}{\partial x} - \frac{(\partial v_y h)}{\partial y} \tag{1.2.5}$$

1.3 Fourth order Runge-Kutta analysis

1.3.1 Fourth order Runge-Kutta for ODEs model

For this part, we start at,

$$\phi^{l+1} = \phi^l + \Delta t g(\phi^l, t^l, \Delta t) \tag{1.3.1}$$

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

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

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

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

Here we apply two-dimensional Taylor series expansion for k_2, k_3, k_4, and apply total derivative with respect to time to get following results,

$$\phi^{l+1} = \phi^l + \frac{\Delta t}{1!} f(\phi^l, t^l) + \frac{\Delta t^2}{2!} \frac{Df}{dt}\bigg|_{t^l} + \frac{\Delta t^3}{3!} \frac{D^2 f}{dt^2}\bigg|_{t^l} + \frac{\Delta t^4}{4!} \frac{D^3 f}{dt^3}\bigg|_{t^l} + O(\Delta t^5) \tag{1.3.2}$$

while g in equation (1.3.1) for RK4 should be,

$$g = a_1 k_1 + a_2 k_2 + a_3 k_3 + a_4 k_4$$

and equate coefficients we get the system of equations, we can get the coefficient of RK4,

$$a_1 = \tfrac{1}{6} \quad a_2 = \tfrac{1}{3} \quad a_3 = \tfrac{1}{3} \quad a_4 = \tfrac{1}{6}$$

and finally the RK4 result is,

$$\phi^{l+1} = \phi^l + \Delta t(\frac{1}{6} k_1 + \frac{1}{3} k_2 + \frac{1}{3} k_3 + \frac{1}{6} k_4) \tag{1.3.3}$$

1.3.2 RK4 Stability analysis

According to previous equations (1.3.1), (1.3.2), (1.3.3), we can do more expansion as following steps,

$$
\begin{aligned}
\phi^{l+1} &= \phi^l + \Delta t(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4) \\
&= \phi^l + \Delta t(\frac{1}{6}f(\phi^l, t^l) + \frac{1}{3}f(\phi^l + \frac{\Delta t}{2}k_1, t^l + \frac{\Delta t}{2}) \\
&\quad + \frac{1}{3}f(\phi^l + \frac{\Delta t}{2}k_2, t^l + \frac{\Delta t}{2}) + \frac{1}{6}f(\phi^l + \Delta t k_3, t^l + \Delta t)) \\
&= \phi^l + \Delta t(\frac{1}{6}\lambda\phi^l + \frac{1}{3}(\lambda\phi^l + \lambda(\phi^l + (\lambda\phi^l))\Delta t + ... \\
&= \phi^l(1 + \lambda\Delta t + \frac{(\lambda\Delta t)^2}{2!} + \frac{(\lambda\Delta t)^3}{3!} + \frac{(\lambda\Delta t)^4}{4!})
\end{aligned}
$$

Thus the solution at any time step l can be written as,

$$
\begin{aligned}
\phi^l &= \phi^0(1 + \lambda\Delta t + \frac{(\lambda\Delta t)^2}{2!} + \frac{(\lambda\Delta t)^3}{3!} + \frac{(\lambda\Delta t)^4}{4!})^l \\
&= \phi^0\sigma^l
\end{aligned}
\tag{1.3.4}
$$

While we know that in order to have stability $|\sigma| \leq 1$, and we can easily plot the figure of stability region,

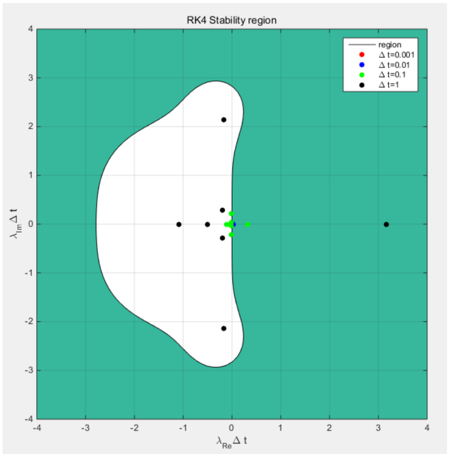


Figure 2: RK4 stability region for six order central finite difference method with different time step

1.3.3 RK4 Error Analysis

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

$$\begin{aligned}\sigma &= (1 + i\lambda_{Im}\Delta t + \frac{(i^2\lambda_{Im}\Delta t)^2}{2!} + \frac{(i^3\lambda_{Im}\Delta t)^3}{3!} + \frac{(i^4\lambda_{Im}\Delta t)^4}{4!}) \\ &= Ze^{i\theta}\end{aligned}$$

So here we can work out the value of Z and θ, and for amplitude error,

$$\begin{aligned}Z &= \sqrt{(1 - \frac{1}{2}\lambda^2\Delta t^2 + \frac{1}{24}\lambda^4\Delta t^4)^2 + (\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3)^2} \\ &= \sqrt{\frac{\lambda^8\Delta t^8}{576} - \frac{\lambda^6\Delta t^6}{72} + 1}\end{aligned}$$

and for the phase error,

$$\theta = \tan^{-1}(\frac{\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3}{1 - (\frac{1}{2}\lambda^2\Delta t^2 - \frac{1}{24}\lambda^4\Delta t^4)})$$

recall we can use the equation, $\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \alpha^4 + ...$ and the phase error equation becomes,

$$\begin{aligned}\theta &= \tan^{-1}(\frac{\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3}{1 - (\frac{1}{2}\lambda^2\Delta t^2 - \frac{1}{24}\lambda^4\Delta t^4)}) \\ &= \tan^{-1}(\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3)(1 + (\frac{1}{2}\lambda^2\Delta t^2 - \frac{1}{24}\lambda^4\Delta t) + (\frac{1}{2}\lambda^2\Delta t^2 - \frac{1}{24}\lambda^4\Delta t^4)^2 + (\frac{1}{2}\lambda^2\Delta t^2 - \frac{1}{24}\lambda^4\Delta t^4)^3) \\ &\quad + ...) \\ &= \tan^{-1}((\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3) + (\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3)(\frac{1}{2}\lambda^2\Delta t^2 - \frac{1}{24}\lambda^4\Delta t) + ...)\end{aligned}$$

Using the same approach that we did for the Euler and Crank-Nicolson methods, we can show that the phase error is given by:

$$\theta - (\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3) = \tan^{-1}((\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3) + (\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3)(\frac{1}{2}\lambda^2\Delta t^2 - \frac{1}{24}\lambda^4\Delta t) + ...) - (\lambda\Delta t - \frac{1}{6}\lambda^3\Delta t^3)$$

Well based on the above information we got the plot of amplitude and phase error,

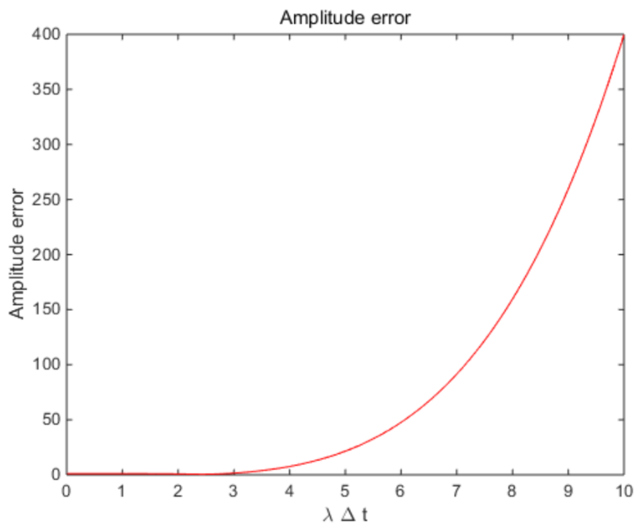


Figure 3: Amplitude error for six order central difference method

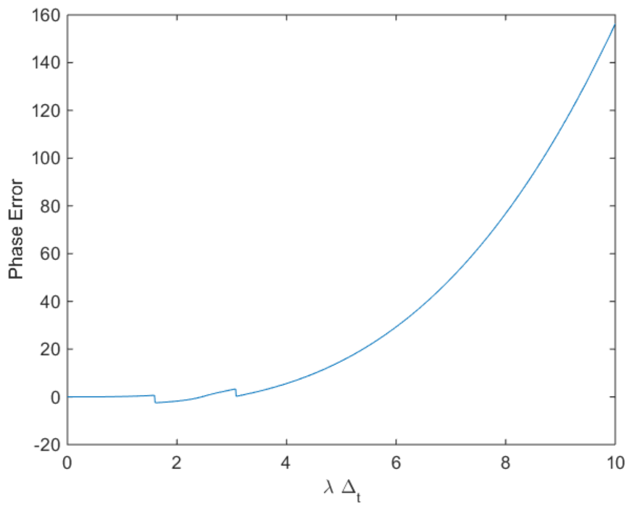


Figure 4: Phase error for six order central difference method

1.4 Six order central difference stencil method and its derivation

This part we are supposed to start at general finite difference formula,

$$\left.\frac{d^n\phi}{dx_i^n}\right|_{x_i} = \frac{1}{x^n}\left(\sum_{m=1}^{M} a_{-m}\phi_{i-m} + a_0\phi_i + \sum_{m=1}^{M} a_{+m}\phi_{i+m}\right) \tag{1.4.1}$$

and then according to our shallow water equation problem we just set $n = 1$ in this case because it is a first derivative problem and we can acquire the equation from equation (1.4.1),

$$\left.\frac{d\phi}{dx_i}\right|_{x_i} = \frac{1}{x}\left(\sum_{m=1}^{M} a_{-m}\phi_{i-m} + a_0\phi_i + \sum_{m=1}^{M} a_{+m}\phi_{i+m}\right) \tag{1.4.2}$$

Now considering the Taylor expansion,

$$\phi(x_i \pm N_m\Delta x) = phi(x_i) \pm (N_m\Delta x)\left.\frac{d\phi}{dx}\right|_{x_i} + \frac{(N_m\Delta x)^2}{2!}\left.\frac{d^2\phi}{dx^2}\right|_{x_i} + \frac{(N_m\Delta x)^3}{3!}\left.\frac{d^3\phi}{dx^3}\right|_{x_i} + O(N_m\Delta x^4)$$

In our assignment, we need to set $1 \leq m \leq 6$, and substituting the resulting expressions for $\phi_{i\pm m}$ into the Equation (1.4.1) and collect coefficients of each derivative of Φ,

$$\begin{aligned}
\left.\frac{d\phi}{dx_i}\right|_{x_i} =& \frac{1}{\Delta x}\Bigg((a_{-3} + a_{-2} + a_{-1} + a_0 + a_1 + a_2 + a_3) \quad \phi(x_i)\Bigg) \\
&+ \Bigg((a_{-3}(-3) + a_{-2}(-2) + a_{-1}(-1) + a_0(0) + a_1(1) + a_2(2) + a_3(3)) \quad \Delta x\left.\frac{d\phi}{dx}\right|_{x_i}\Bigg) \\
&+ \Bigg((a_{-3}\frac{(-3)^2}{2!} + a_{-2}\frac{(-2)^2}{2!} + a_{-1}\frac{(-1)^2}{2!} + a_0(0) + a_1\frac{(1)^2}{2!} + a_2\frac{(2)^2}{2!} + a_3\frac{(3)^2}{2!}) \quad \Delta x^2\left.\frac{d^2\phi}{dx^2}\right|_{x_i}\Bigg) \\
&+ \Bigg((a_{-3}\frac{(-3)^3}{3!} + a_{-2}\frac{(-2)^3}{3!} + a_{-1}\frac{(-1)^3}{3!} + a_0(0) + a_1\frac{(1)^3}{3!} + a_2\frac{(2)^3}{3!} + a_3\frac{(3)^3}{3!}) \quad \Delta x^3\left.\frac{d^3\phi}{dx^3}\right|_{x_i}\Bigg) \\
&+ \Bigg((a_{-3}\frac{(-3)^4}{4!} + a_{-2}\frac{(-2)^4}{4!} + a_{-1}\frac{(-1)^4}{4!} + a_0(0) + a_1\frac{(1)^4}{4!} + a_2\frac{(2)^4}{4!} + a_3\frac{(3)^4}{4!}) \quad \Delta x^4\left.\frac{d^4\phi}{dx^4}\right|_{x_i}\Bigg) \\
&+ \Bigg((a_{-3}\frac{(-3)^5}{5!} + a_{-2}\frac{(-2)^5}{5!} + a_{-1}\frac{(-1)^5}{5!} + a_0(0) + a_1\frac{(1)^5}{5!} + a_2\frac{(2)^5}{5!} + a_3\frac{(3)^5}{5!}) \quad \Delta x^5\left.\frac{d^5\phi}{dx^5}\right|_{x_i}\Bigg) \\
&+ \Bigg((a_{-3}\frac{(-3)^6}{6!} + a_{-2}\frac{(-2)^6}{6!} + a_{-1}\frac{(-1)^6}{6!} + a_0(0) + a_1\frac{(1)^6}{6!} + a_2\frac{(2)^6}{6!} + a_3\frac{(3)^6}{6!}) \quad \Delta x^6\left.\frac{d^6\phi}{dx^6}\right|_{x_i}\Bigg)
\end{aligned}$$

and work out the system,

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -3 & -2 & -1 & 0 & 1 & 2 & 3 \\ \frac{(-3)^2}{2} & \frac{(-2)^2}{2} & \frac{(-1)^2}{2} & 0 & \frac{(1)^2}{2} & \frac{(2)^2}{2} & \frac{(3)^2}{2} \\ \frac{(-3)^3}{6} & \frac{(-2)^3}{6} & \frac{(-1)^3}{6} & 0 & \frac{(1)^3}{6} & \frac{(2)^3}{6} & \frac{(3)^3}{6} \\ \frac{(-3)^4}{24} & \frac{(-2)^4}{24} & \frac{(-1)^4}{24} & 0 & \frac{(1)^4}{24} & \frac{(2)^4}{24} & \frac{(3)^4}{24} \\ \frac{(-3)^5}{120} & \frac{(-2)^5}{120} & \frac{(-1)^5}{120} & 0 & \frac{(1)^5}{120} & \frac{(2)^5}{120} & \frac{(3)^5}{120} \\ \frac{(-3)^6}{720} & \frac{(-2)^6}{720} & \frac{(-1)^6}{720} & 0 & \frac{(1)^6}{720} & \frac{(2)^6}{720} & \frac{(3)^6}{720} \end{bmatrix} \cdot \begin{bmatrix} a_{-3} \\ a_{-2} \\ a_{-1} \\ a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

solve the equation we can get the value from a_{-3} to a_3,

$$\begin{array}{llll} a_{-3} = -\dfrac{1}{60} & a_{-2} = \dfrac{3}{20} & a_{-1} = -\dfrac{3}{4} & a_0 = 0 \\ a_1 = \dfrac{3}{4} & a_2 = -\dfrac{3}{20} & a_3 = \dfrac{1}{60} & \end{array}$$

1.5 OpenMp

When paralyzing our code with OpenMp, memory used for the master thread during the serial region will be shared among other just-created sub threads within the paralyzed region, at which time code will be executed simultaneously by all of the threads. OpenMp introduced a fork/join paralyzing model, which illustrates that out of the parallel region, all code will be executed with only one master threads and the thread ID is 0. Sub threads will be created as entering the parallel region, which looks like a fork. After this parallel region, sub threads will come back to the master threads, which is called as join. Since the memory is shared among master threads and sub threads, thread with their own ID can thus utilize data of other threads.

Therefore, achieving this operation requires the programmer to let the compiler know at which part sub threads need to be created. Specific directives and clauses need to be added. For example, if a for loop will loop for 4 times, we can then add '<u>pragma omp for</u>' before the for clause to enable the for loop to loop 4 times simultaneously. For PDE problem, as we always perform the time integration by applying Euler or Rounge-Kutta method, iteration is required for both numerical methods. For loops are therefore inevitable, OpenMp can thus be applied before the time marching loop, and other time-consuming iteration part, such as assembling results at one time step.

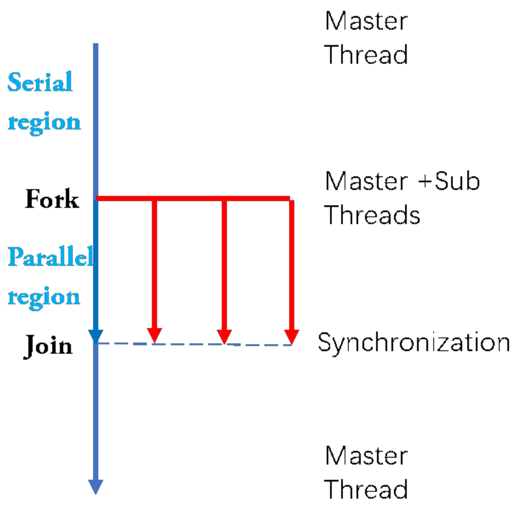


Figure 5: at serial region only Master thread will execute, at parallel region sub threads will be created

1.6 MPI

Message passing interface, is the most widely supported parallel mechanism. Unlike OpenMp, which aided the parallel with shared memory, MPI helps to shorten the running time by assigning memory to different processes. Such that there can be processes with different ID executing same part of the code at same time. In order to make use of the data or value obtained by other processes, MPI offered us with several build-in functions to use, such as MPI_Send or MPI_Rec, which helps to send and receive data required for calculation between different processes.

When it comes to PDE problems, as described before, iterations are inevitable if numerical methods are applied. With MPI, iteration process will then be divided into several jobs, which will be calculated within several processes, MPI_Send/MPI_Rec will be used to perform the data sending among these processes. Different from OpenMp that all threads are basically operating on same data, MPI assign the data grid into several parts and a single process will only have access to one single part of data grid, which greatly shortens the running time.

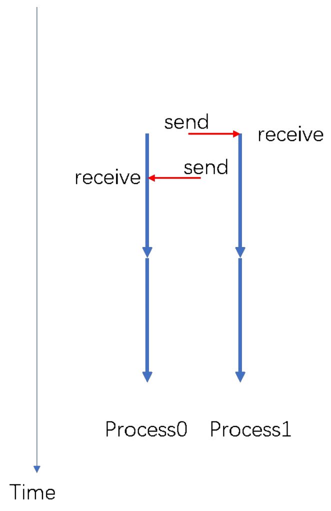


Figure 6: There will be data transfer between each process during execution

2 Method

2.1 Finite Difference method for Shallow Water Equation

To apply the six order finite central difference at shallow water equation, it is just like the following figure. In the same time period, relationship between target position and the other closed positions can be specified by this method. We need to consider values around the target position with their coefficient which we have calculated. Recalling equation 1.2.3 to 1.2.5, we transfered the PDE to serval ODE functions and then applied the superposition of each result of ODE. Furthermore, we can expand these three governing equations into six order finite central difference version and code it into the program.

$$\begin{aligned}\frac{\partial v_x}{\partial t} &= -v_x\frac{\partial v_x}{\partial x} - v_y\frac{\partial v_x}{\partial y} - g\frac{\partial h}{\partial x} \\ &= -\frac{v_x}{\Delta x}\Big(-\frac{1}{60}v_{x(i-3,j)} + \frac{3}{20}v_{x(i-2,j)} - \frac{3}{4}v_{x(i-1,j)} + \frac{3}{4}v_{x(i+1,j)} - \frac{3}{20}v_{x(i+2,j)} + \frac{1}{60}v_{x(i+3,j)}\Big) \\ &\quad - \frac{v_y}{\Delta y}\Big(-\frac{1}{60}v_{x(i,j-3)} + \frac{3}{20}v_{x(i,j-2)} - \frac{3}{4}v_{x(i,j-1)} + \frac{3}{4}v_{x(i,j+1)} - \frac{3}{20}v_{x(i,j+2)} + \frac{1}{60}v_{x(i,j+3)}\Big) \\ &\quad - \frac{gh}{\Delta x}\Big(-\frac{1}{60}h_{i-3,j} + \frac{3}{20}h_{i-2,j} - \frac{3}{4}h_{i-1,j} + \frac{3}{4}h_{i+1,j} - \frac{3}{20}h_{i+2,j} + \frac{1}{60}h_{i+3,j}\Big)\end{aligned} \tag{2.1.1}$$

$$\begin{aligned}\frac{\partial v_y}{\partial t} &= -v_x\frac{\partial v_y}{\partial x} - v_y\frac{\partial v_y}{\partial y} - g\frac{\partial h}{\partial y} \\ &= -\frac{v_x}{\Delta x}\Big(-\frac{1}{60}v_{y(i-3,j)} + \frac{3}{20}v_{y(i-2,j)} - \frac{3}{4}v_{y(i-1,j)} + \frac{3}{4}v_{y(i+1,j)} - \frac{3}{20}v_{y(i+2,j)} + \frac{1}{60}v_{y(i+3,j)}\Big) \\ &\quad - \frac{v_y}{\Delta y}\Big(-\frac{1}{60}v_{y(i,j-3)} + \frac{3}{20}v_{y(i,j-2)} - \frac{3}{4}v_{y(i,j-1)} + \frac{3}{4}v_{y(i,j+1)} - \frac{3}{20}v_{y(i,j+2)} + \frac{1}{60}v_{y(i,j+3)}\Big) \\ &\quad - \frac{gh}{\Delta y}\Big(-\frac{1}{60}h_{i,j-3} + \frac{3}{20}h_{i,j-2} - \frac{3}{4}h_{i,j-1} + \frac{3}{4}h_{i,j+1} - \frac{3}{20}h_{i,j+2} + \frac{1}{60}h_{i,j+1}\Big)\end{aligned} \tag{2.1.2}$$

$$
\begin{aligned}
\frac{\partial h}{\partial t} = -\frac{(\partial v_x h)}{\partial x} - \frac{(\partial v_y h)}{\partial y} \\
-\frac{h}{\Delta x}\Big(-\frac{1}{60}v_{x(i-3,j)} + \frac{3}{20}v_{x(i-2,j)} - \frac{3}{4}v_{x(i-1,j)} + \frac{3}{4}v_{x(i+1,j)} - \frac{3}{20}v_{x(i+2,j)} + \frac{1}{60}v_{x(i+3,j)}\Big) \\
-\frac{v_x}{\Delta x}\Big(-\frac{1}{60}h_{(i-3,j)} + \frac{3}{20}h_{(i-2,j)} - \frac{3}{4}h_{(i-1,j)} + \frac{3}{4}h_{(i+1,j)} - \frac{3}{20}h_{(i+2,j)} + \frac{1}{60}h_{(i+3,j)}\Big) \\
-\frac{h}{\Delta y}\Big(-\frac{1}{60}v_{y(i,j-3)} + \frac{3}{20}v_{y(i,j-2)} - \frac{3}{4}v_{y(i,j-1)} + \frac{3}{4}v_{y(i,j+1)} - \frac{3}{20}v_{y(i,j+2)} + \frac{1}{60}v_{y(i,j+3)}\Big) \\
-\frac{v_y}{\Delta y}\Big(-\frac{1}{60}h_{(i,j-3)} + \frac{3}{20}h_{(i,j-2)} - \frac{3}{4}h_{(i,j-1)} + \frac{3}{4}h_{(i,j+1)} - \frac{3}{20}h_{(i,j+2)} + \frac{1}{60}h_{(i,j+3)}\Big)
\end{aligned}
\tag{2.1.3}
$$

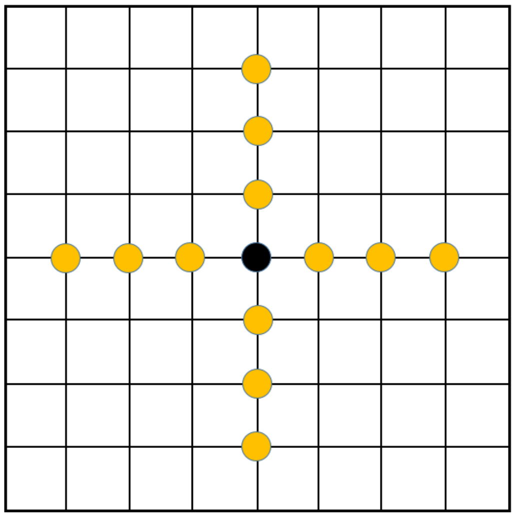
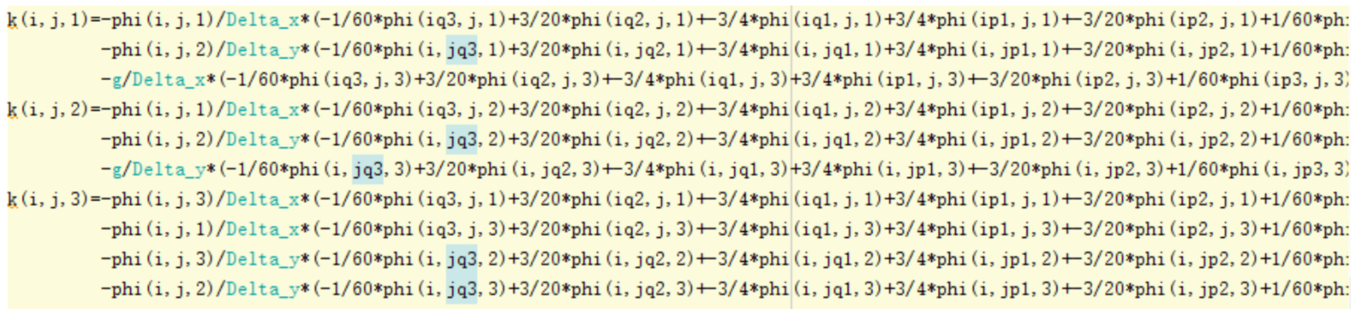


Figure 7: Six order central difference principle diagram

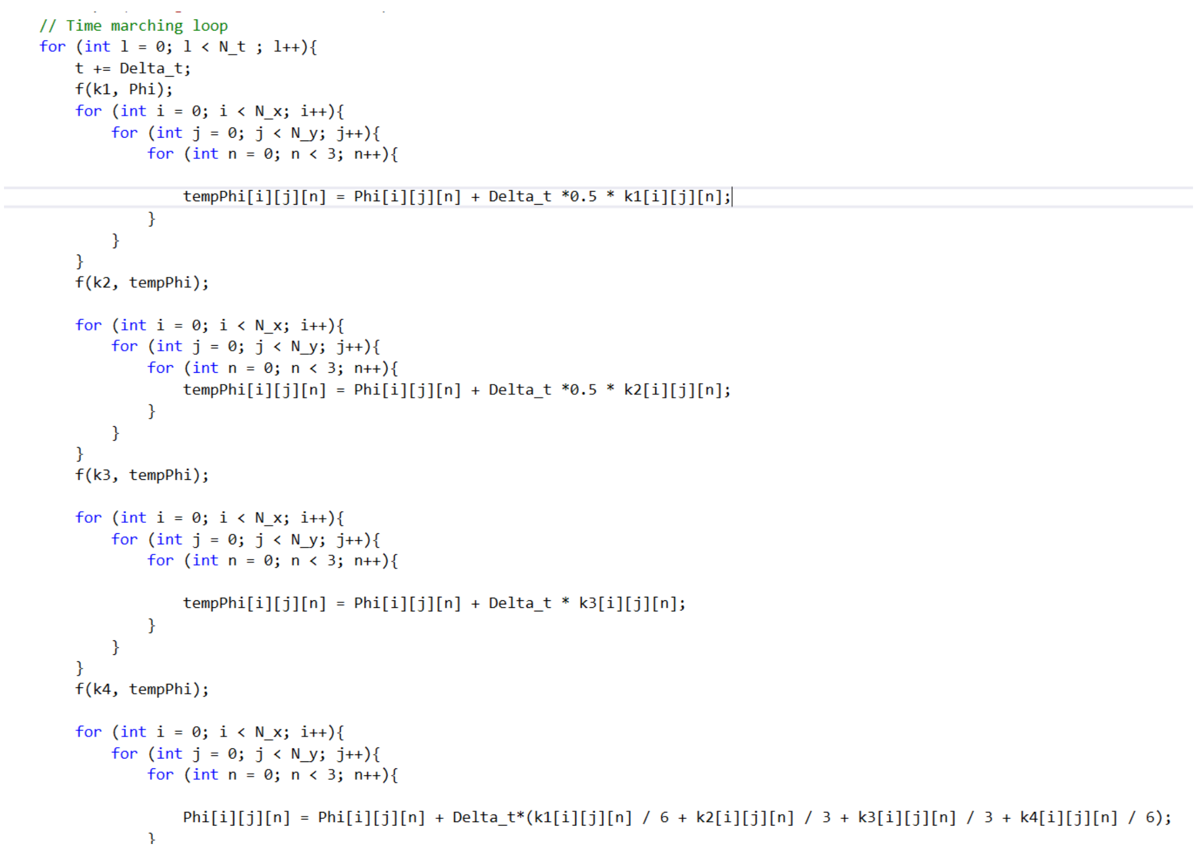
where whether in MATLAB or C++ program, we tend to build a 3D matrix $\phi(N_x \times N_y \times 3)$ to store the parameters v_x, v_y, h, and the example code for MATLAB is,



```
k(i,j,1)=-phi(i,j,1)/Delta_x*(-1/60*phi(iq3,j,1)+3/20*phi(iq2,j,1)+-3/4*phi(iq1,j,1)+3/4*phi(ip1,j,1)+-3/20*phi(ip2,j,1)+1/60*ph
         -phi(i,j,2)/Delta_y*(-1/60*phi(i,jq3,1)+3/20*phi(i,jq2,1)+-3/4*phi(i,jq1,1)+3/4*phi(i,jp1,1)+-3/20*phi(i,jp2,1)+1/60*ph
         -g/Delta_x*(-1/60*phi(iq3,j,3)+3/20*phi(iq2,j,3)+-3/4*phi(iq1,j,3)+3/4*phi(ip1,j,3)+-3/20*phi(ip2,j,3)+1/60*phi(ip3,j,3
k(i,j,2)=-phi(i,j,1)/Delta_x*(-1/60*phi(iq3,j,2)+3/20*phi(iq2,j,2)+-3/4*phi(iq1,j,2)+3/4*phi(ip1,j,2)+-3/20*phi(ip2,j,2)+1/60*ph
         -phi(i,j,2)/Delta_y*(-1/60*phi(i,jq3,2)+3/20*phi(i,jq2,2)+-3/4*phi(i,jq1,2)+3/4*phi(i,jp1,2)+-3/20*phi(i,jp2,2)+1/60*ph
         -g/Delta_y*(-1/60*phi(i,jq3,3)+3/20*phi(i,jq2,3)+-3/4*phi(i,jq1,3)+3/4*phi(i,jp1,3)+-3/20*phi(i,jp2,3)+1/60*phi(i,jp3,3
k(i,j,3)=-phi(i,j,3)/Delta_x*(-1/60*phi(iq3,j,1)+3/20*phi(iq2,j,1)+-3/4*phi(iq1,j,1)+3/4*phi(ip1,j,1)+-3/20*phi(ip2,j,1)+1/60*ph
         -phi(i,j,1)/Delta_x*(-1/60*phi(iq3,j,3)+3/20*phi(iq2,j,3)+-3/4*phi(iq1,j,3)+3/4*phi(ip1,j,3)+-3/20*phi(ip2,j,3)+1/60*ph
         -phi(i,j,3)/Delta_y*(-1/60*phi(i,jq3,2)+3/20*phi(i,jq2,2)+-3/4*phi(i,jq1,2)+3/4*phi(i,jp1,2)+-3/20*phi(i,jp2,2)+1/60*ph
         -phi(i,j,2)/Delta_y*(-1/60*phi(i,jq3,3)+3/20*phi(i,jq2,3)+-3/4*phi(i,jq1,3)+3/4*phi(i,jp1,3)+-3/20*phi(i,jp2,3)+1/60*ph
```

2.2 RK4 For time marching loop

At previous parts, we have already derived the expression of RK4, and we illustrate the C++ program for RK4 time marching loop:



```
// Time marching loop
for (int l = 0; l < N_t ; l++){
    t += Delta_t;
    f(k1, Phi);
    for (int i = 0; i < N_x; i++){
        for (int j = 0; j < N_y; j++){
            for (int n = 0; n < 3; n++){

                tempPhi[i][j][n] = Phi[i][j][n] + Delta_t *0.5 * k1[i][j][n];
            }
        }
    }
    f(k2, tempPhi);

    for (int i = 0; i < N_x; i++){
        for (int j = 0; j < N_y; j++){
            for (int n = 0; n < 3; n++){
                tempPhi[i][j][n] = Phi[i][j][n] + Delta_t *0.5 * k2[i][j][n];
            }
        }
    }
    f(k3, tempPhi);

    for (int i = 0; i < N_x; i++){
        for (int j = 0; j < N_y; j++){
            for (int n = 0; n < 3; n++){

                tempPhi[i][j][n] = Phi[i][j][n] + Delta_t * k3[i][j][n];
            }
        }
    }
    f(k4, tempPhi);

    for (int i = 0; i < N_x; i++){
        for (int j = 0; j < N_y; j++){
            for (int n = 0; n < 3; n++){

                Phi[i][j][n] = Phi[i][j][n] + Delta_t*(k1[i][j][n] / 6 + k2[i][j][n] / 3 + k3[i][j][n] / 3 + k4[i][j][n] / 6);
            }
```

2.3 Enforcement of periodic boundary condition for Various versions of the program and Data structure

In the shallow water equation problem, due to the periodic boundary condition, it is necessary to set up the enforced boundary conditions to deal with some grid points which are out of the current template module. There are four different program styles we have to consider, including MATLAB, C++, C++ with OpenMP parallel and C++ with MPI parallel programs. For first three programs, the method for boundary conditions enforcement are same. When the grid points needed were beyond the template, we would need to let the value of points come to the other sides, as the Figure.8 shown, to realize the boundary condition shift.

However, for MPI parallel program, things can be easier. For previous code, we always need to let the code to make an estimate about the position inside the grid. Boundary will be particularly enforced if is three elements away from the edge. But for MPI code, contributed by the exchange function, the 'edge' part in previous code now becomes part of the interior grid and can be treated in a same way as the interior grid of previous code.

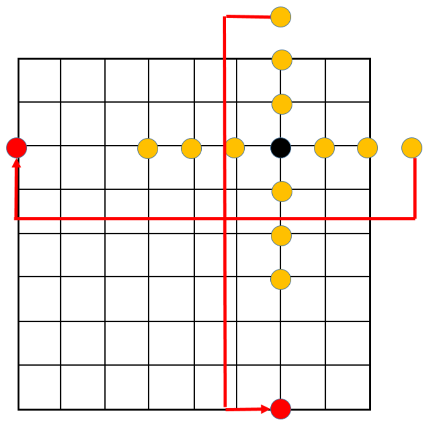
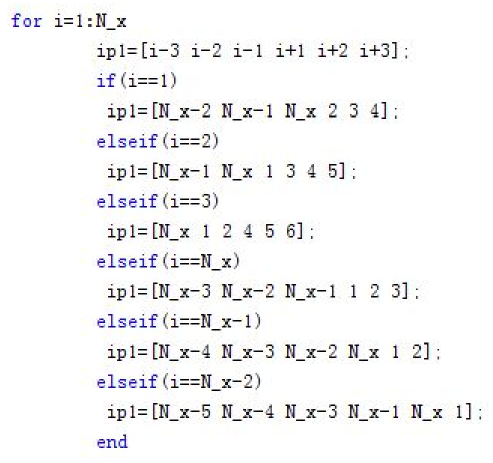


Figure 8: Diagram of Principle of MATLAB, C++ and C++ OpenMP Boundary condition enforce

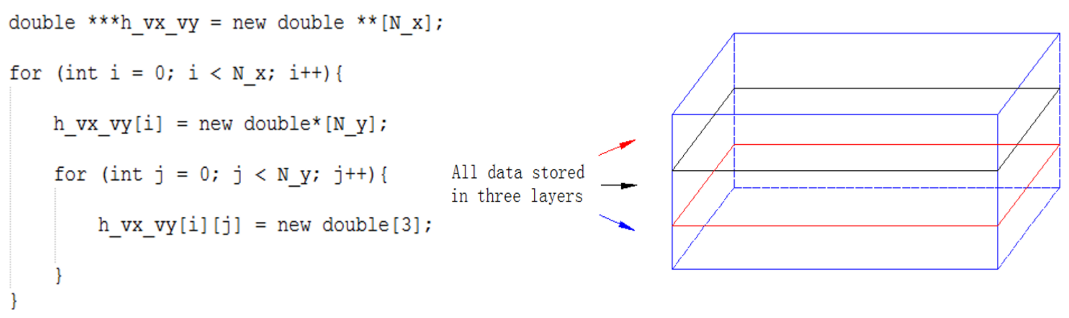
As for data structure, at the beginning, I was about to store all elements inside a $myNx + 6 \times 3 \times 3$ cuboid, which is easier for me to construct and easier for me to call them, but occupies more space obviously. I then changed the data structure into a stripe, which is actually storing all elements in a column. Not only this method enables us to call the elements with the same index as before, it also saves more space.

More specifically, this data structure can be illustrated by figure.12. Not only can we save more space by arranging in this way, also we can call them with exactly the same index as we used in previous method.



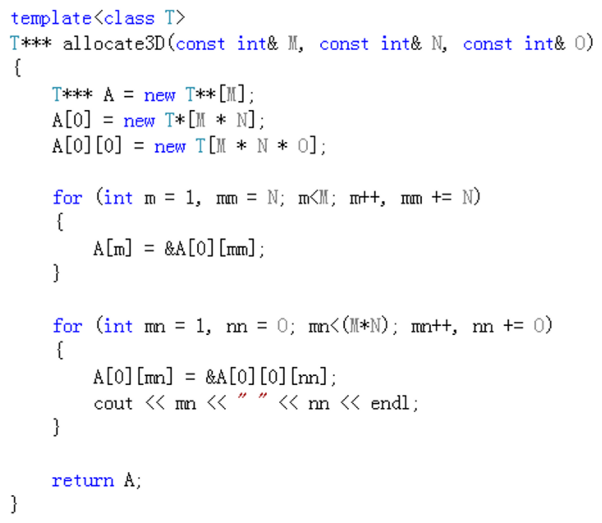
```
for i=1:N_x
        ip1=[i-3 i-2 i-1 i+1 i+2 i+3];
        if(i==1)
         ip1=[N_x-2 N_x-1 N_x 2 3 4];
        elseif(i==2)
         ip1=[N_x-1 N_x 1 3 4 5];
        elseif(i==3)
         ip1=[N_x 1 2 4 5 6];
        elseif(i==N_x)
         ip1=[N_x-3 N_x-2 N_x-1 1 2 3];
        elseif(i==N_x-1)
         ip1=[N_x-4 N_x-3 N_x-2 N_x 1 2];
        elseif(i==N_x-2)
         ip1=[N_x-5 N_x-4 N_x-3 N_x-1 N_x 1];
        end
```

Figure 9: realizing the boundary condition in MATLAB



```
double ***h_vx_vy = new double **[N_x];

for (int i = 0; i < N_x; i++){

    h_vx_vy[i] = new double*[N_y];

    for (int j = 0; j < N_y; j++){

        h_vx_vy[i][j] = new double[3];

    }
}
```

Figure 10: Storing all elements in a cuboid



```
template<class T>
T*** allocate3D(const int& M, const int& N, const int& O)
{
    T*** A = new T**[M];
    A[0] = new T*[M * N];
    A[0][0] = new T[M * N * O];

    for (int m = 1, mm = N; m<M; m++, mm += N)
    {
        A[m] = &A[0][mm];
    }

    for (int mn = 1, nn = O; mn<(M*N); mn++, nn += O)
    {
        A[0][mn] = &A[0][0][nn];
        cout << mn << " " << nn << endl;
    }

    return A;
}
```

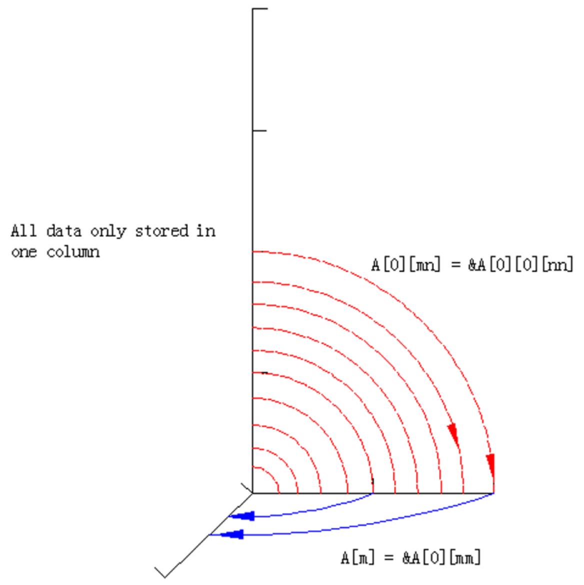


Figure 11: Storing all elements in a column

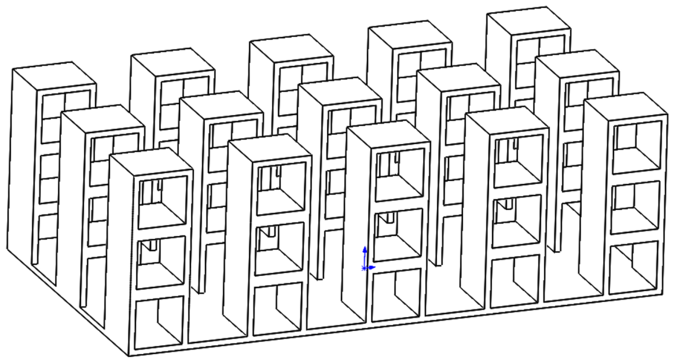


Figure 12: Storing all elements in a column and calling them by using their address relative the column

2.4 OpenMP application to parallelize solution

For our code, shallow water equation, since it requires large number of loop times, part of the loops is even nested. In order to achieve the highest efficiency, OpenMP directives thus need to be added to the most time-consuming part. As a result, we first add the 'for' directive before the time marching loop and each affiliated loops for collecting temporary RK4 results. We did not add it inside the RK4 function since that would create more threads and increase the running time, since the threads are already spread out. One thing need to be mentioned is that there will be a 'single' region inside the parallel region, which means that all threads need to stop here and wait for the write function and time calculation to proceed. If 'single' region not introduced, time and write function would be executed many times. Our algorithm is illustrated by a flow chart.

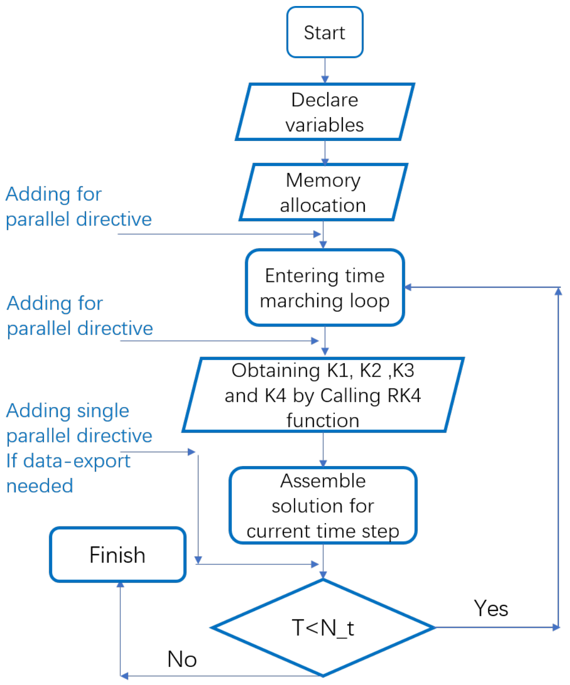
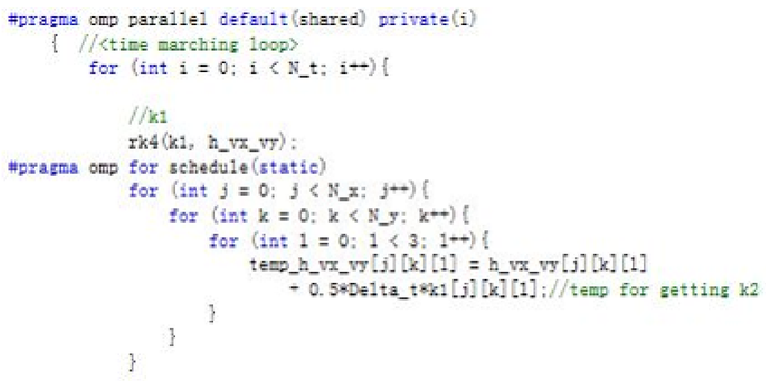

Start
Declare variables
Memory allocation
Adding for parallel directive
Entering time marching loop
Adding for parallel directive
Obtaining K1, K2 ,K3 and K4 by Calling RK4 function
Adding single parallel directive If data-export needed
Assemble solution for current time step
Finish
T<N_t
Yes
No

Figure 13: Algorithm of our OpenMp code

As illustrated by Figure.13 that we need to add 'for' directive before the time marching loop, sub threads also need to be created inside the time marching loop for the calculation of 'K' values of Rounge Kutta method. When four 'K' values are obtained, solution at one time step can be assembled. What the actual code structure looks like is showed by figure.14. If our code is running on our own laptop, 'write' function will be called to export the data. If not, 'write' will not be called as it's quite time-consuming, results of illustrating time consumed by file I/O are at a later section.



```
#pragma omp parallel default(shared) private(i)
    {  //<time marching loop>
        for (int i = 0; i < N_t; i++){

            //k1
            rk4(k1, h_vx_vy);
#pragma omp for schedule(static)
            for (int j = 0; j < N_x; j++){
                for (int k = 0; k < N_y; k++){
                    for (int l = 0; l < 3; l++){
                        temp_h_vx_vy[j][k][l] = h_vx_vy[j][k][l]
                            + 0.5*Delta_t*k1[j][k][l];//temp for getting k2
                    }
                }
            }
```

Figure 14: Part of our OpenMp code

2.5 MPI application to parallelize solution

For our shallow water equations, the usual MPI_Send/MPI_Rec may not be enough, since our data grid is not a 1-D stripe, but a 2-D grid instead. As a result, we need to apply other MPI functions in order to perform data exchange between different process among 2-D grid. We then need to make use of several build-in functions of MPI, such as MPI_Cart_create(), MPI_Comm_rank() and MPI_Cart_coords().
After using these functions, the number of grid after been divided into processes will be then be the total number of processes assigned. And the assignment of process to each grid will be like in figure below.

The process arrangement is illustrated by figure above, the blue region at the edge of each process is called halo region, since data stored there will not be exported into final solution but will be used for calculation. For the shallow water equation, the inner white part is a $myNx \times myNy$ square and the halo stripe has the dimension of $(myNx + 6) \times 3 \times 3$, 3 means three layers of data, h, Vx and Vy, another '3' corresponds to the periodic boundary condition. This kind of boundary condition enables us to present and maintain the effect of wave, which requires exchanging data between left and right, top and bottom edges. Halo region after MPI_Send/MPI_Rec will possess the value from the edge part of other process. Comparing with the flow chart of OpenMp, the major part of MPI code will be a little bit different.

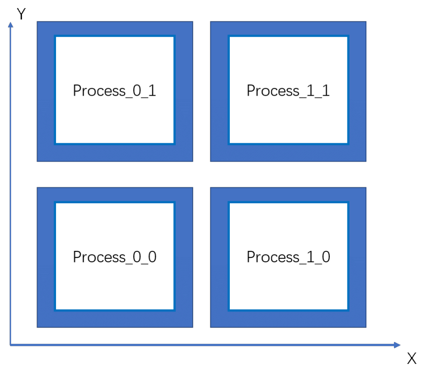


Figure 15: process allocation if four processes are assigned, White - data grid, Blue - halo region

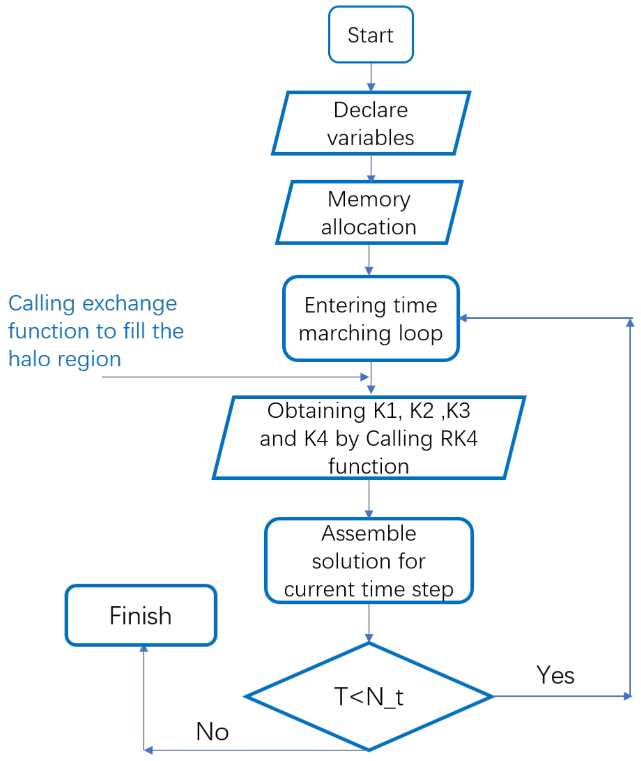
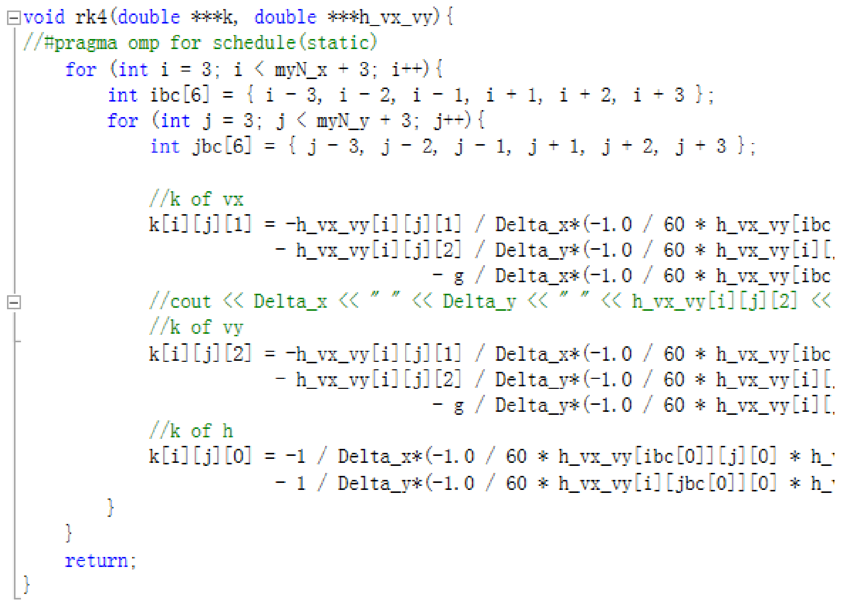

Start
Declare variables
Memory allocation
Entering time marching loop
Calling exchange function to fill the halo region
Obtaining K1, K2 ,K3 and K4 by Calling RK4 function
Assemble solution for current time step
Finish
T<N_t
Yes
No

Figure 16: Flow chart of MPI code

However, the RK4 function of MPI code can be shortened accordingly. Different from previous sections, periodic boundary conditions at edges can be enforced in a same way of the interior grid. Since the halo region is been filled with suitable values. For this case, the region with index from 3 to myNx+3 can be treated as interior region. illustrated by figure.17 and figure.18.



```
void rk4(double ***k, double ***h_vx_vy){
//#pragma omp for schedule(static)
    for (int i = 3; i < myN_x + 3; i++){
        int ibc[6] = { i - 3, i - 2, i - 1, i + 1, i + 2, i + 3 };
        for (int j = 3; j < myN_y + 3; j++){
            int jbc[6] = { j - 3, j - 2, j - 1, j + 1, j + 2, j + 3 };

            //k of vx
            k[i][j][1] = -h_vx_vy[i][j][1] / Delta_x*(-1.0 / 60 * h_vx_vy[ibc
                        - h_vx_vy[i][j][2] / Delta_y*(-1.0 / 60 * h_vx_vy[i][
                                       - g / Delta_x*(-1.0 / 60 * h_vx_vy[ibc
            //cout << Delta_x << " " << Delta_y << " " << h_vx_vy[i][j][2] <<
            //k of vy
            k[i][j][2] = -h_vx_vy[i][j][1] / Delta_x*(-1.0 / 60 * h_vx_vy[ibc
                        - h_vx_vy[i][j][2] / Delta_y*(-1.0 / 60 * h_vx_vy[i][
                                       - g / Delta_y*(-1.0 / 60 * h_vx_vy[i][
            //k of h
            k[i][j][0] = -1 / Delta_x*(-1.0 / 60 * h_vx_vy[ibc[0]][j][0] * h_
                        - 1 / Delta_y*(-1.0 / 60 * h_vx_vy[i][jbc[0]][0] * h_
        }
    }
    return;
}
```

Figure 17: Modification inside the RK4 function

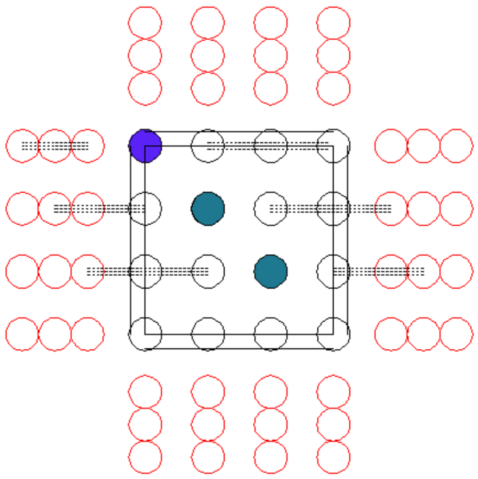
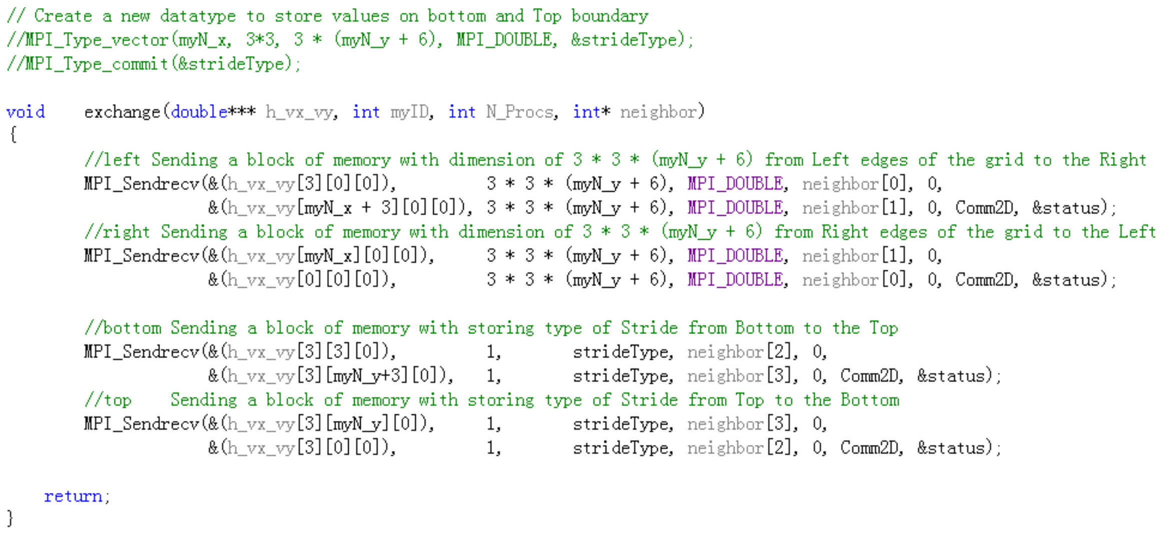


Figure 18: Boundary condition can be satisfied easily



```
// Create a new datatype to store values on bottom and Top boundary
//MPI_Type_vector(myN_x, 3*3, 3 * (myN_y + 6), MPI_DOUBLE, &strideType);
//MPI_Type_commit(&strideType);

void    exchange(double*** h_vx_vy, int myID, int N_Procs, int* neighbor)
{
        //left Sending a block of memory with dimension of 3 * 3 * (myN_y + 6) from Left edges of the grid to the Right
        MPI_Sendrecv(&(h_vx_vy[3][0][0]),         3 * 3 * (myN_y + 6), MPI_DOUBLE, neighbor[0], 0,
                     &(h_vx_vy[myN_x + 3][0][0]), 3 * 3 * (myN_y + 6), MPI_DOUBLE, neighbor[1], 0, Comm2D, &status);
        //right Sending a block of memory with dimension of 3 * 3 * (myN_y + 6) from Right edges of the grid to the Left
        MPI_Sendrecv(&(h_vx_vy[myN_x][0][0]),     3 * 3 * (myN_y + 6), MPI_DOUBLE, neighbor[1], 0,
                     &(h_vx_vy[0][0][0]),         3 * 3 * (myN_y + 6), MPI_DOUBLE, neighbor[0], 0, Comm2D, &status);

        //bottom Sending a block of memory with storing type of Stride from Bottom to the Top
        MPI_Sendrecv(&(h_vx_vy[3][3][0]),         1,        strideType, neighbor[2], 0,
                     &(h_vx_vy[3][myN_y+3][0]),   1,        strideType, neighbor[3], 0, Comm2D, &status);
        //top    Sending a block of memory with storing type of Stride from Top to the Bottom
        MPI_Sendrecv(&(h_vx_vy[3][myN_y][0]),     1,        strideType, neighbor[3], 0,
                     &(h_vx_vy[3][0][0]),         1,        strideType, neighbor[2], 0, Comm2D, &status);

    return;
}
```

Figure 19: Real code of Exchange() Function

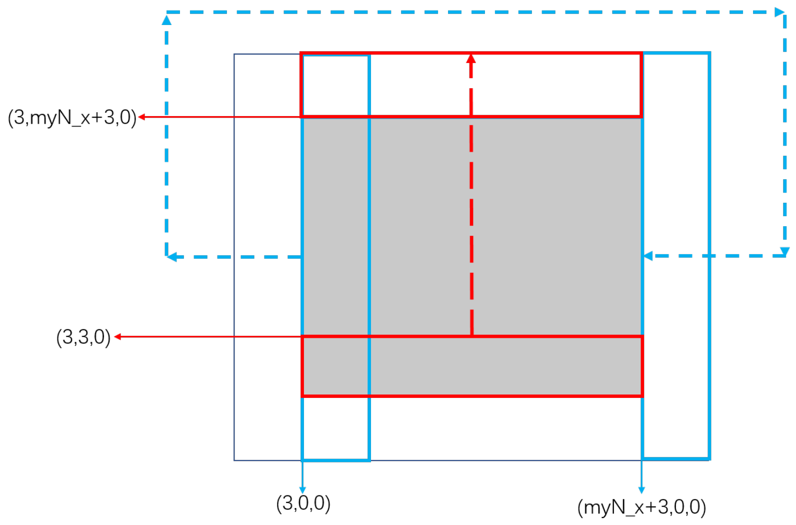


Figure 20: Graphical explanation of Exchange() function

Specifically, the exchange function is illusrated by Figure.19. Horizontal exchange based on big chunk of memory, vertical exchange will apply another kind of memory called strideType, which covers the whole $myNx \times 3 \times 3$ part of data. The exchange function helps to exchange four edges of the interior grid of a single process to other processes, which can be graphically illustrated as figure.20. When applying the boundary condition, halo points are all within the reach, RK4 function can thus be shortened accordingly.

The number of grids been decomposed is nˆ2, and I have tested 4 and 9 grids on my laptop, more grids/processes execution was experimented after uploaded on barcoo.

In terms of file I/O, previous code can all export data within only one process, which means only one set of data file will be generated and this file covers the data of the whole grid. Unlike them, MPI code runs on several processes and the number of set of data files exported equals the number of processes assigned, which contains only a portion of data. Files generated by these processes can eventually constitute the whole data grid, which is what we want and does not include the halo region.

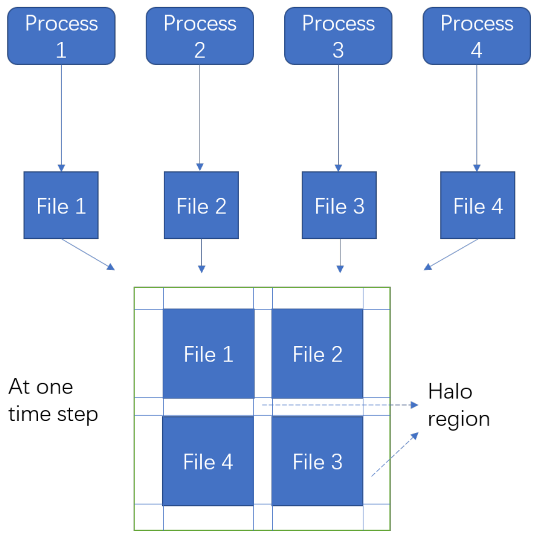
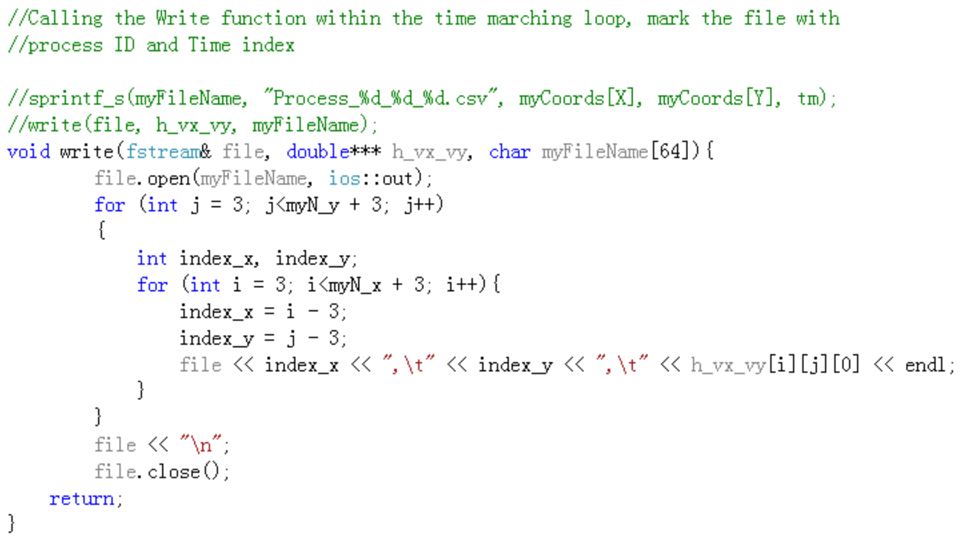


Figure 21: Files been exported do not include any halo region

Therefore, in order to combine all of the data file at the end, each exported file needs to be marked with their own processes ID and the index that can illustrate the time step. Screen shot of our code is illustrated by Figure.22.

As we can see from Figure.22 that we call the file-writing function within the time marching loop, with calling parameters myCoords-X, myCoords-Y and tm, which help to tag the file exported with process ID and the time index. We then need to open one file within the write function with this tagged name, write the data and close it. This is how we export file within one process at one time step. If I run it with 4 processes, there will be 4 set of '.csv' files generated eventually and can be visualized through Paraview.



```
//Calling the Write function within the time marching loop, mark the file with
//process ID and Time index

//sprintf_s(myFileName, "Process_%d_%d_%d.csv", myCoords[X], myCoords[Y], tm);
//write(file, h_vx_vy, myFileName);
void write(fstream& file, double*** h_vx_vy, char myFileName[64]){
        file.open(myFileName, ios::out);
        for (int j = 3; j<myN_y + 3; j++)
        {
            int index_x, index_y;
            for (int i = 3; i<myN_x + 3; i++){
                index_x = i - 3;
                index_y = j - 3;
                file << index_x << ",\t" << index_y << ",\t" << h_vx_vy[i][j][0] << endl;
            }
        }
        file << "\n";
        file.close();
    return;
}
```

Figure 22: Exporting file in actual code

3 Results

After explanation of part of the code and algorithm, results of four versions of code are posted in this section. Each of the post are the combined results of Shallow Water equation at different time step.

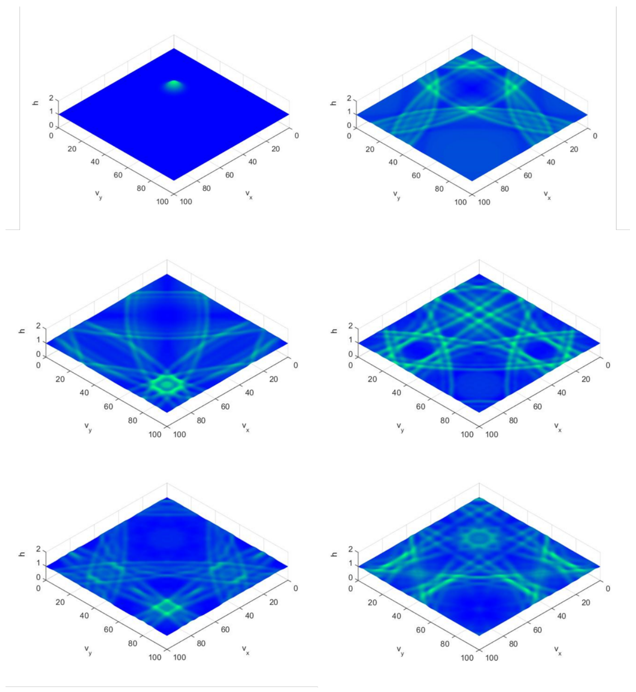


Figure 23: Screenshot of solution at different time step produced by matlab

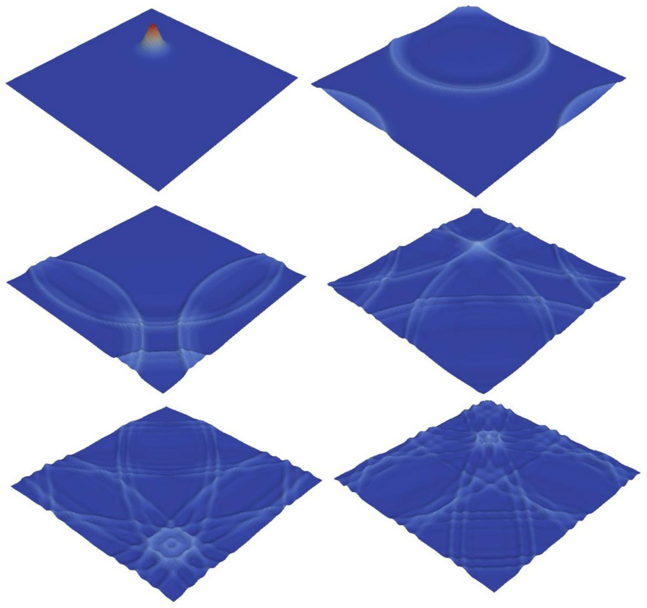


Figure 24: Screenshot of solution at different time step produced by cpp code

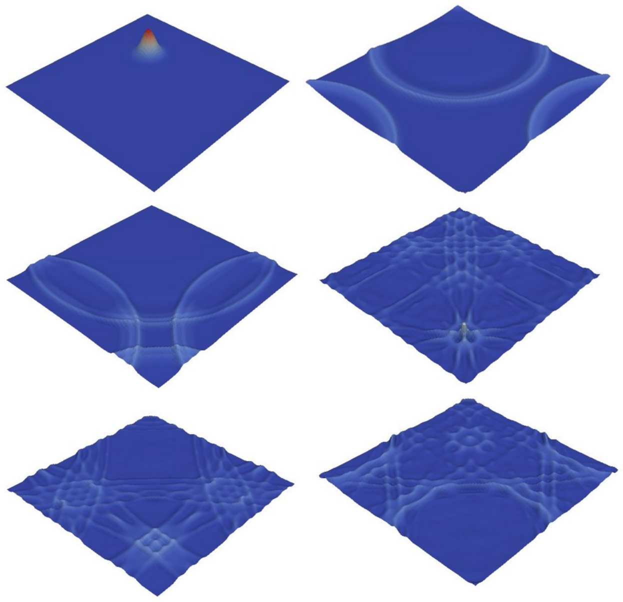


Figure 25: Screenshot of solution at different time step produced by OpenMp code with 4 threads

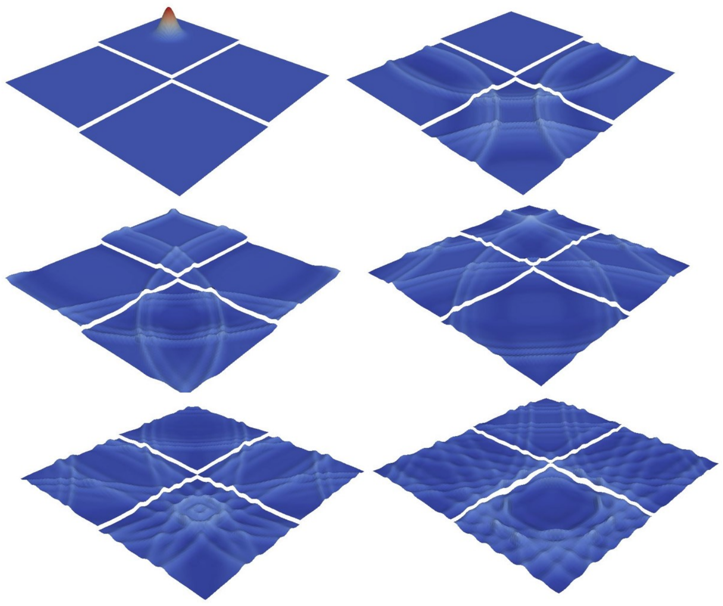


Figure 26: Screenshot of solution at different time step produced by MPI code with 4 processes

3.1 Scaling Results of OpenMP code

Table 1: Time required per integration step for OpenMp code when upload on barcoo

Threads	1	2	4	8	16	32	64
dx=dy=0.1,dt=0.02	0.652	0.327	0.164	0.0831	0.0462	0.0548	0.0542
dx=dy=1,dt=0.02	0.00610	0.00310	0.00160	0.00082	0.00048	0.000264	0.000314
dx=dy=1,dt=0.2	0.00621	0.00585	0.00549	0.00533	0.00542	0.0141	0.0251

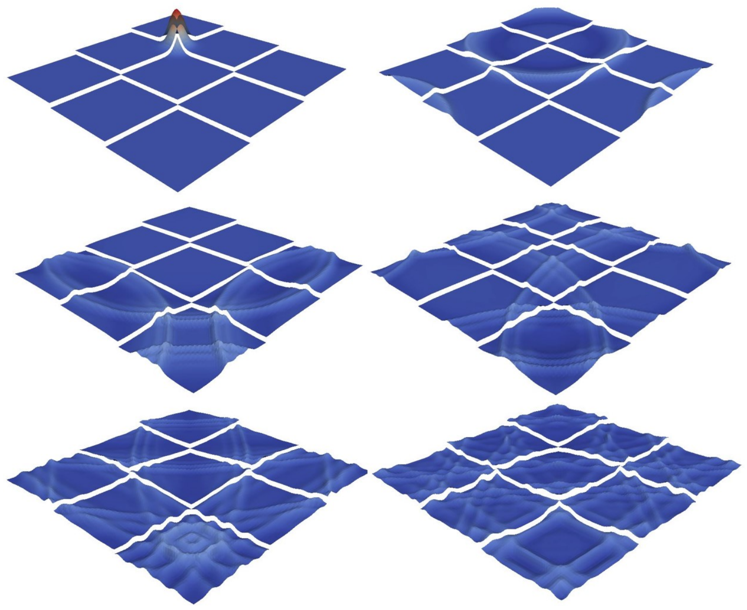


Figure 27: Screenshot of solution at different time step produced by MPI code with 9 processes

Table 2: Time required per integration step for OpenMp code with File I/O when upload on barcoo

Threads	1	2	4	8	16	32	64
dx=dy=1,dt=0.02	0.01057	0.01058	0.010554	0.01055	0.01054	0.01057	0.01056
dx=dy=1,dt=0.2	0.0113	0.01072	0.0106	0.010714	0.01066	0.010719	0.010737
dx=dy=0.1,dt=0.02	1.0935	1.0308	1.0303	1.0304	1.0923	1.0279	1.0288

As we can see form table.1 that scales time step produces relatively weak effect on time saving, but if we scales dx and dy, its effect on running time is highly observable. Except for scaling, all code present the best performance if 16 threads are used, we can imply a sharp rise from threads 1 to threads 16, running time per time step present a decreasing trend when using more threads. This phenomenon can be related by the nodes' property of barcoo, which has only 16 threads per node, communication between nodes will cost more time if more threads need to be introduced.

Table.2 illustrates the file I/O can have an impact on the running time. Since for each time step, a large amount of data will need to be exported, which cost for time. The total time therefore is longer than that without file exporting.

Graphically, effect of weak scaling for dx=dy=0.01 and dx=dy=0.1 is obvious after using 16 threads, as we can see form figure.xxx that when having more grid points, the decrease in speedup is larger than that with less grid points.

When we tried to make time step larger, the result is even more observable.

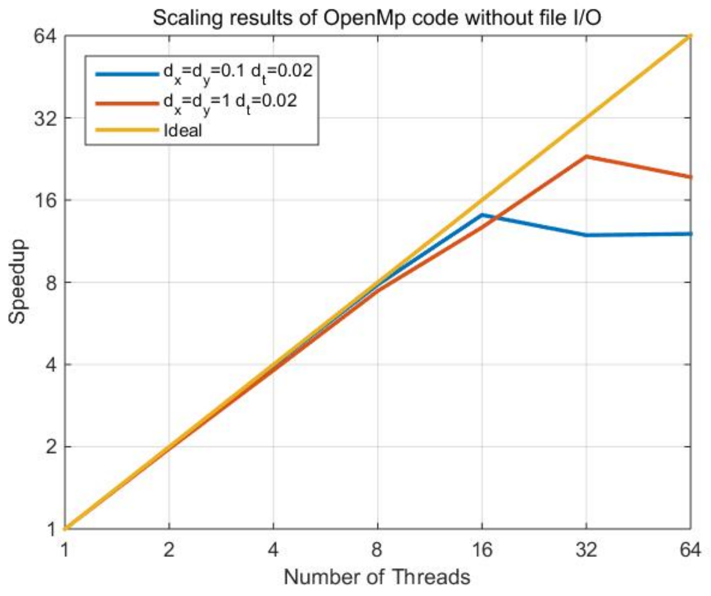


Figure 28: scaling results of increasing grid points by OpenMp code

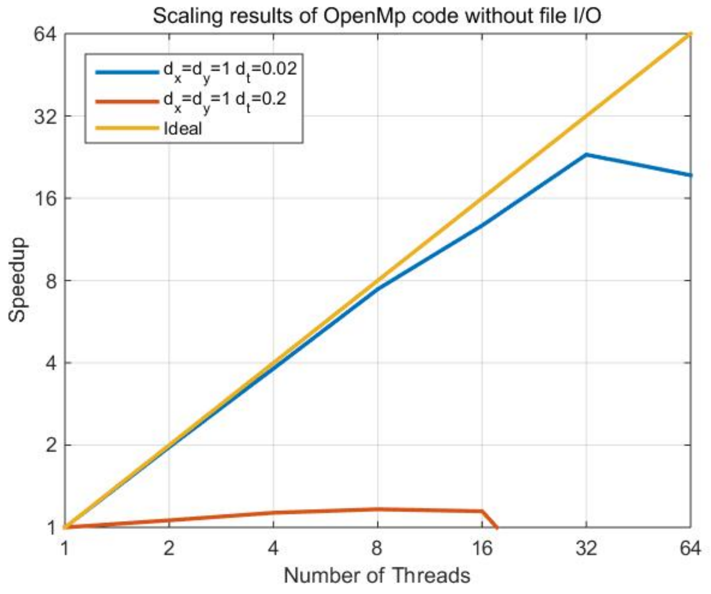


Figure 29: scaling results of increasing time step by OpenMp code

As we can see from Figure.29 that increased number of iteration made the power of speedup contributed by using more threads prolonged to 32 threads. This honestly is something we did not expected to happen. Since we thought what we compared was the average time cost per time step, which would not be affected by increasing or decreasing time steps. However, this is happening. What we can imply from it is that if OpenMp method is going to be applied, satisfying results can be produced by coding with smaller time step, ie more iterations. This phenomenon can be explained by efficiency managing. Since the speedup we plotted is a relative quantity, too many time steps may cost relative long processing time with one thread used. Once more threads participate into the execution, the whole iteration can be effectively divided into several portions. Based on shared memory, running time is reduced. That is why such phenomenon is not obvious for smaller iteration times, since it may be fast enough already with one threads. Admittedly, effective as it is, the parallel efficiency and speedup can experience a drop after using more than 16 threads. This is because barcoo can only provide us with 16 threads for one node. Using more threads means one node is not enough, informing other nodes is therefore inevitable but time-consuming.

Following this, we compared the speed up for same scaling code with file I/O. Results are as expected.

Comparing with previous results, file I/O greatly limited the speedup. Not only the file opening and closing takes time, adding single directive inside the parallel region also limit the speed. As the file exporting cannot be paralyzed, $\sharp pragma\_omp\_single$ means the part will only be executed by one single thread and would limit the potential of speedup.

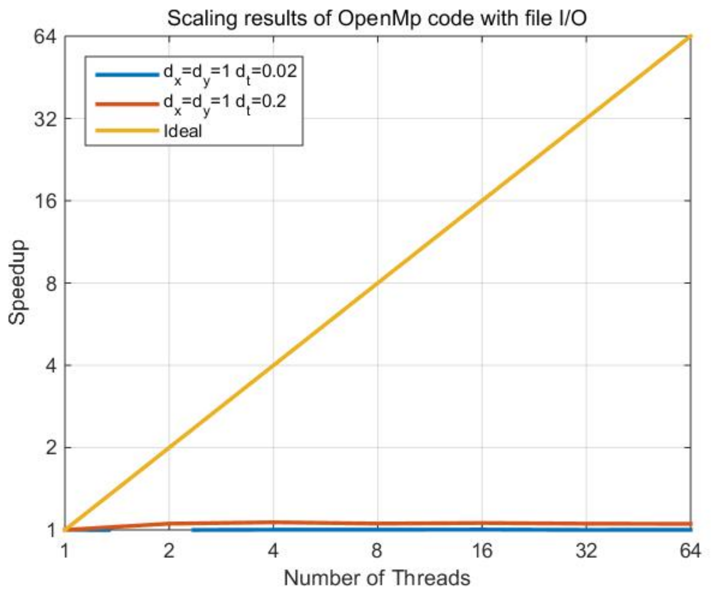


Figure 30: scaling results of increasing time step by OpenMp code with file I/O

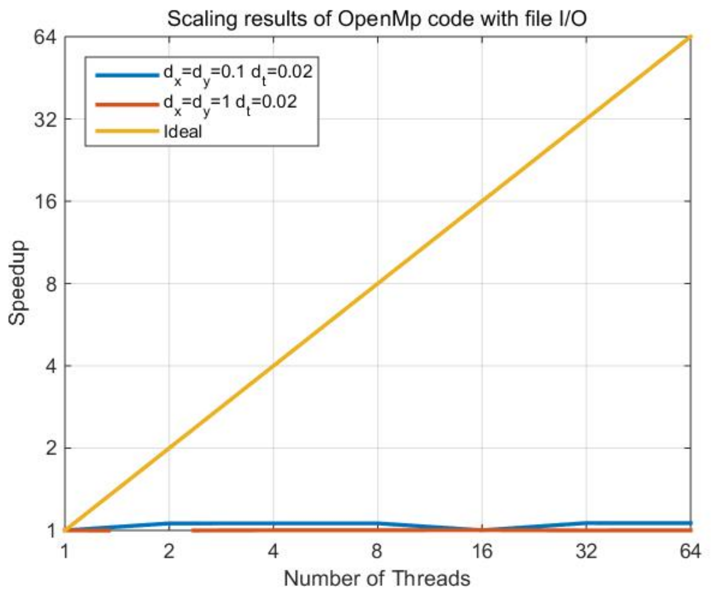


Figure 31: scaling results of increasing grid points by OpenMp code with file I/O

3.2 Scaling Results of MPI code

Table 3: Time required for MPI code without File I/O when running on barcoo

Processes	4	9	16	25	36
grid 1000 × 1000,dt=0.1	14.372	14.7925	16.3337	15.3714	15.4491
Processes	49	64	81	100	
grid 1000 × 1000,dt=0.1	17.7739	17.5863	17.6882	17.8928	

Table 4: Time required for MPI code without File I/O when running on barcoo

Processes	4	9	16	25	36
grid 100 × 100,dt=0.1	0.143928	0.193516	0.174702	0.184978	0.233391
Processes	49	64	81	100	
grid 100 × 100,dt=0.1	0.27448	0.206099	0.228061	0.21362	

Table 5: Time required for MPI code without File I/O when running on barcoo

Processes	4	9	16	25	36
grid 1000 × 1000,dt=0.2	7.00159	7.26787	8.59168	8.18284	8.97944
Processes	49	64	81	100	
grid 1000 × 1000,dt=0.2	8.99902	8.93141	9.04355	9.09262	

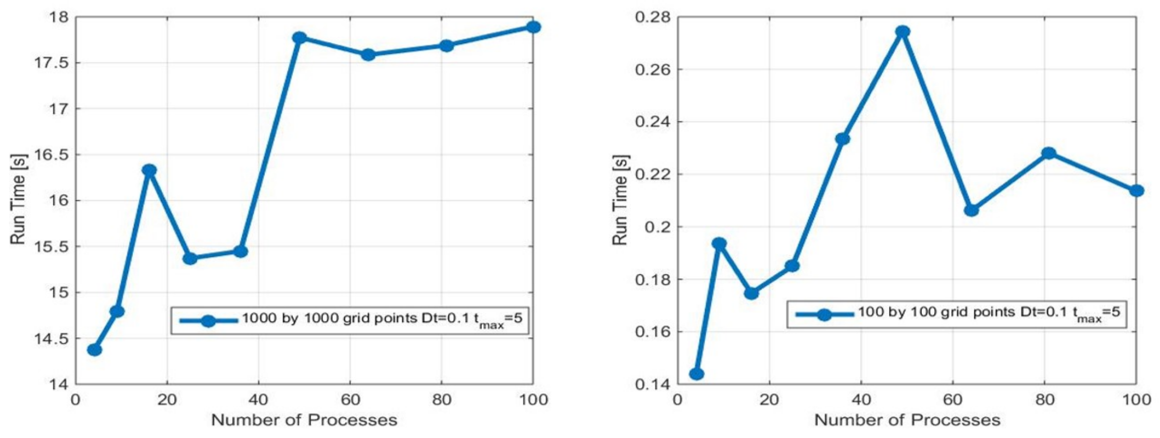
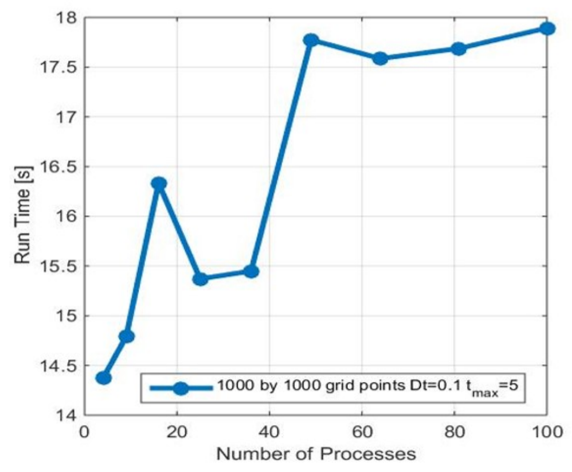


Figure 32: scaling results of changing grid points by MPI code without file I/O

Graphically, the scaling effects are very obvious above.

As we can imply from Figure.32 and Figure.33, Table.3 to Table.5, when we tried to decrease the grid points of each processes, running time dropped dramatically. Results of changing time step basically present two identical trends that running time rockets quickly by using or more than 25 processes. Based on this results, running time can be saved if applying appropriate grid points, larger time step and more carefully orchestrated number of processes, as parallel efficiency would negatively affected once too many processes are introduced.



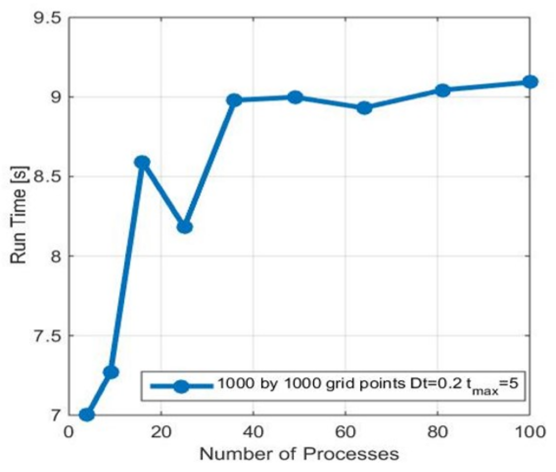


Figure 33: scaling results of changing total time steps by MPI code without file I/O

Speaking of parallel efficiency of MPI code, since the execution always includes the the communication between processes, which takes time, the running time thus can be increased due to the regular communication at each time step. When we use only 4 processes, level of communication is far less than that of 81 and 100 processes, that is why for our Shallow-Water-Equation, the efficiency drops at certain number of processes. Note that from both figures, running speed increased temporarily at 25 processes. This may be another optimum number of processes that balanced the slow communication time with short running time.

4 Discussion

As we can see from the scaling results that they are all less than ideal when more threads are used if more than 16 threads are used for OpenMp code. Parallel efficiency can be used to illustrate this point.

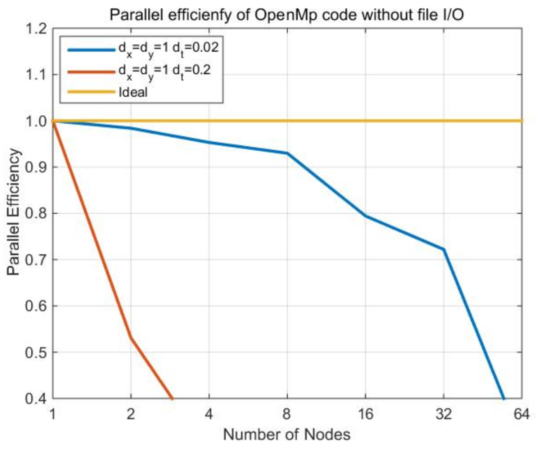


Figure 34: parallel efficiency of increasing time step by OpenMp code without file I/O

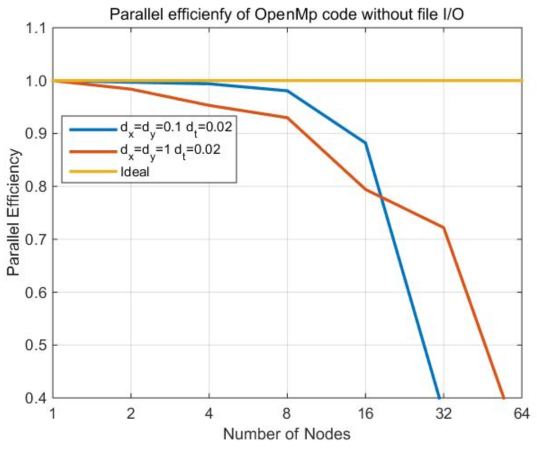


Figure 35: parallel efficiency of increasing grid points by OpenMp code without file I/O

In terms of OpenMp code, in order to reduce the running time and increase the efficiency, grid points should not be too many. The following figure illustrates the speedup and parallel efficiency produced by a 1000×1000 grid, time step is 0.1.

As we can see form figure.36 that more grid points and smaller time step results in lower potential of speedup and lower parallel efficiency. Therefore, one way to reduce running time is to use appropriate number of grid points.

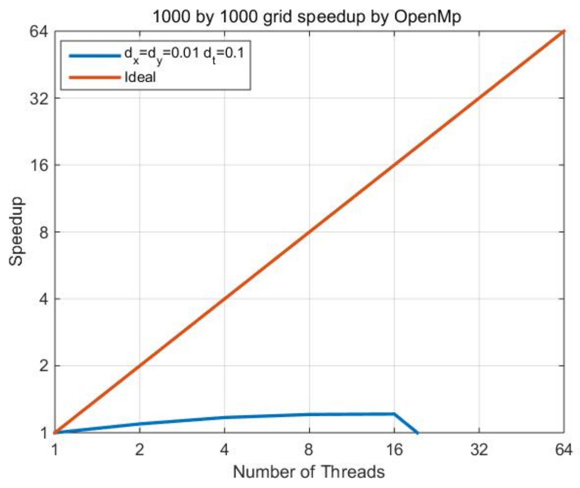


Figure 36: Speedup of OpenMp code with 1000 × 1000 grid

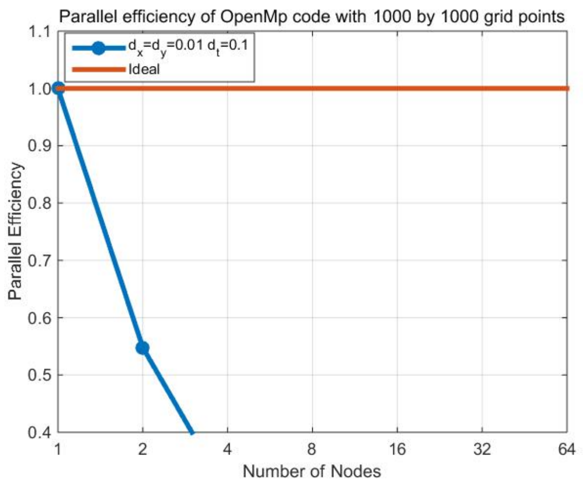


Figure 37: Parallel efficiency of OpenMp code with 1000 × 1000 grid

In terms of the optimization of our MPI code, we tried to embedded our MPI code with OpenMP directives, Which theoretically can help to create more threads for each assigned process. Different form the previous OpenMP code, we did not add OpenMP directive before the time marching loop, but we did put the for directive inside the time marching loop. Since things can go wrong if I let many threads performing the message passing.

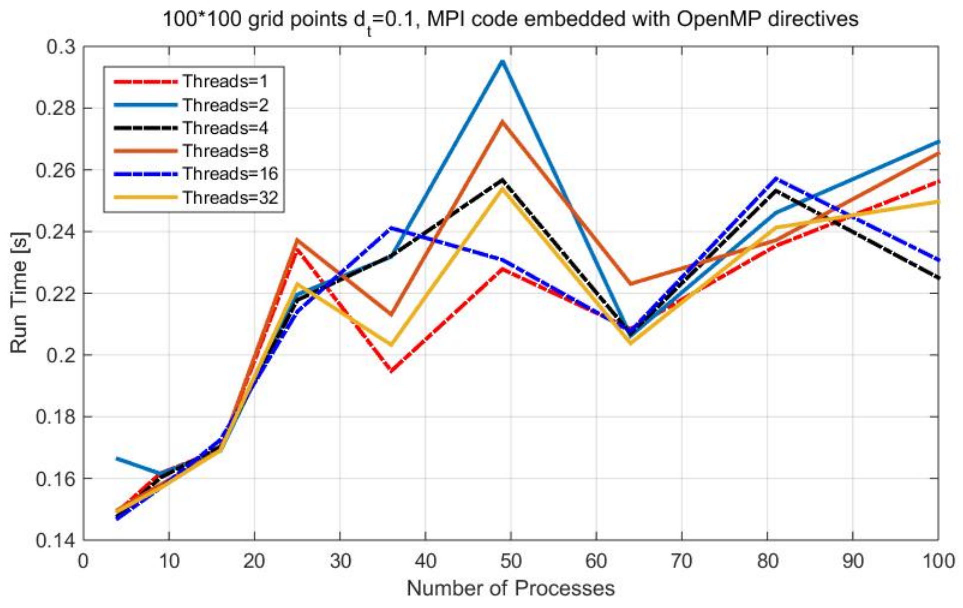


Figure 38: Comparing running time of hybrid parallel code with 100 $\times$ 100 grid with dt=0.1

Table 6: Results of total time required for hybrid parallel code

Time	P=4	P=9	P=16	P=25	P=36	P=49	P=64	P=81	P=100
T=1	0.1493	0.1616	0.1693	0.2342	0.1948	0.2278	0.2084	0.2353	0.2561
T=2	0.1663	0.1615	0.1689	0.2196	0.2318	0.2953	0.2062	0.2460	0.2690
T=4	0.1478	0.1602	0.1702	0.2177	0.2321	0.2566	0.2069	0.2532	0.2250
T=8	0.1498	0.1579	0.1691	0.2371	0.2131	0.2755	0.2230	0.2371	0.2653
T=16	0.1470	0.1569	0.1724	0.2140	0.2411	0.2308	0.2077	0.2571	0.2308
T=32	0.1493	0.1569	0.1690	0.2229	0.2032	0.2537	0.2037	0.2412	0.2496

What we can imply from Figure.38 is that at the beginning, they all present the similar trend. At 36 processes, code running with 1 thread is the fastest, which is followed by that of using 32 and 8 threads. By 49 processes, code with 2 threads is the slowest. However, interestingly, the running time at the 64 processes by basically all processes gathered together at roughly 0.2s. At the end, 100 processes, code running with 4 and 16 threads seems to be the most time-saving type.

For all of the scaling results, generally, I would say they are all less than ideal. Specifically, part of the scaling results may present a close-to ideal trend, but the trend is then heading to a negative way. Therefore, whether the outcome of our code is satisfying is up to the initial condition we used is appropriate or not, such as grid points and time steps. Communication among the hardware of the super computer is basically inevitable especially large number of threads or processes are to be introduced. How to balance the time spent on hardware and saved by parallel would be the key to high efficiency in MPI code, which is surely a bottleneck of parallel programming. Communication is essential in parallel programming when we need to use intermediate result. Rounge-Kutta method is quite typical for this, parallel slowdown can thus increase the running time when OpenMP method is used. Therefore, if we want to improve the efficiency of the algorithm, we can either reduce the level of hardware communication or the use of intermediate results.

5 Reference

Applied Numerical Method Steve More