Columbia University

APMAE9101: Research

MPI and OpenMP programming for solving 2D Advection Equations

Author: Sheng Zhang

Supervised by
Prof. Kyle Mandli

January 1, 2017

1 Introduction

Large-scale computational simulations were identified by Nobel laureate Ken Wilson as the third paradigm of scientific discovery joining the more traditional paradigms of the theory and experiment[8]. Computational simulations make it possible to obtain accurate results for experiments that are very costly or time-consuming to physical experiments. They can also be used to predict situations in real-time such as weather forecasting or to perform experiments that are impossible in real life such as the evolution of the universe. Computational simulations inspire many challenges. Besides the mathematical and numerical complications involved in creating accurate models and suitable numerical methods for simulating those models, many technical software engineering challenges are encountered when developing high performance simulation tools. The behavior of a phenomenon or a physical system that varies in both spatial and temporal dimensions is usually the subject of computational simulations. To conduct simulations with certain size, resolution and accuracy requirements, there might be a need for solving the system for a large spatial grid and for a large number of time steps. These requirements lead to the need for performing very expensive computations that require extensive computational resources. This introduces the need for high performance computing and the optimized scientific software tools.

During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures clearly show that parallelism is the future of computing. Parallel computing has been considered to be "the high end of computing", and has been used to model difficult problems in many areas of science and engineering. Today, commercial applications provide an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. Nevertheless, Parallelization can harm the code readability, generality and extensibility if it is not done in a careful manner.

The aim of this research report is to design, implement and test the message-passing and shared-memory parallelization by solving the 2D scalar advection equations. We first introduce the 2d advection equation and some numerical schemes to solve it using the finite volume method. Then we parallelize the serial code for solving the 2D advection equation using MPI and OpenMP programming, respectively. For each parallelization method, we also do some simple scalability studies to test the performance. Finally, according to the results we obtained, we give some plans for future improvements and applications.

2 The Advection Equation

2.1 Mathematics of Advection

The advection equation is the partial differential equation that governs the motion of a conserved scalar field as it is advected by a known velocity vector field. One easily visualized example of advection is the transport of ink dumped into a river. As the river flows, ink will move downstream in a "pulse" via advection, as the water's movement itself transports the ink. The advection equation for a conserved quantity described by a scalar field q is expressed mathematically by a continuity equation[1]:

$$\frac{\partial q}{\partial t} + \nabla \cdot (\mathbf{u}q) = 0 \tag{1}$$

where $\nabla\cdot$ is the divergence operator and $\mathbf{u}$ is the velocity vector field. Frequently, it is assumed that the flow is incompressible, that is, the velocity field satisfies:

$$\nabla \cdot \mathbf{u} = 0. \tag{2}$$

if so, the equation (1) can be rewritten as

$$\frac{\partial q}{\partial t} + \mathbf{u} \cdot \nabla q = 0. \tag{3}$$

For the purpose of this report we are going to work with the two-dimensional advection equation. If we take this case and assume the velocity of the flow in the x and y directions to be constant, the equation(3) can be rewritten as

$$\frac{\partial q}{\partial t} + \bar{u}\frac{\partial q}{\partial x} + \bar{v}\frac{\partial q}{\partial y} = 0. \tag{4}$$

Equation (4) is a scalar, linear, constant-coefficient PDE of hyperbolic type. The general solution of this equation is very easy to determine. Any smooth function of the form

$$q(x, y, t) = q_0(x - \bar{u}t, y - \bar{v}t) \tag{5}$$

satisfies the differential equation (4), as is easily verified, and in fact any solution to (4) is of this form for some $q_0(x, y)$. To find the particular solution to (4) of interest in a practical problem, we need more information in order to determine the particular function q_0 in (5): initial conditions and perhaps boundary conditions for the equation. Here we just consider the Cauchy problem (infinite spatial-domain) with initial condition $q(x, y, t_0) = q_0(x, y)$, and thus we can immediately write down the particular solution to (4) is

$$q(x, y, t) = q_0(x - \bar{u}(t - t_0), y - \bar{v}(t - t_0)). \tag{6}$$

2.2 Finite Volume Method for the 2D Scalar Advection

2.2.1 General Formulation and Dimensional Splitting[2]

In one space dimension, a finite volume method is based on subdividing the spatial domain into intervals (also called "grid cells") and keeping track of an approximation to the integral of q over each of these volumes. In each time step we update these values using approximations to the flux through the endpoints of the intervals. Given Q_i^n, the cell average at time t_n, we want to approximate Q_i^{n+1}, the cell averages at the next time t_{n+1} after a time step of length $\Delta t = t_{n+1} - t_n$. The general numerical formulation is of the form

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}(F_{i+1/2}^n - F_{i-1/2}^n) \tag{7}$$

where $F^n_{i-1/2}$ is some approximation to the average flux along $x = x_{i-1/2}$. Suppose that we can obtain $F^n_{i-1/2}$ based only on the values Q^n_{i-1} and Q^n_i, the cell averages on either side of the interface, then method (7) becomes

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}(\mathcal{F}(Q_i^n, Q_{i+1}^n) - \mathcal{F}(Q_{i-1}^n, Q_i^n))) \tag{8}$$

where $\mathcal{F}$ is some numerical flux function. The specific method obtained depends on how we choose the formula $\mathcal{F}$, but in general any method of this type is an explicit method.

For multidimensional hyperbolic equations, there are three of the most popular general approaches to obtaining numerical methods: fully discrete flux-differencing methods, semidiscrete methods with Runge—Kutta time stepping and dimensional splitting. We will only concentrate on dimensional splitting method for 2D advection because it is by far the simplest way to extend one-dimensional numerical methods to more space dimensions.

For our 2D scalar advection equation

$$\frac{\partial q}{\partial t} + \bar{u}\frac{\partial q}{\partial x} + \bar{v}\frac{\partial q}{\partial y} = 0,$$

we can split it into

$$\text{x-sweeps: } \frac{\partial q}{\partial t} + \bar{u}\frac{\partial q}{\partial x} = 0, \tag{9}$$

$$\text{y-sweeps: } \frac{\partial q}{\partial t} + \bar{v}\frac{\partial q}{\partial y} = 0. \tag{10}$$

In the x-sweeps we start with cell averages Q^n_{ij} at time t_n and solve one-dimensional problem $\frac{\partial q}{\partial t} + \bar{u}\frac{\partial q}{\partial x} = 0$ along each row of cells $\mathcal{C}_{ij}$ with j fixed, updating Q^n_{ij} to Q^*_{ij}:

$$Q^*_{ij} = Q^n_{ij} - \frac{\Delta t}{\Delta x}(F^n_{i+1/2,j} - F^n_{i-1/2,j}), \tag{11}$$

where $F^n_{i-1/2,j}$ is an approximate numerical flux for the one-dimensional problem between cells $\mathcal{C}_{i-1,j}$ and $\mathcal{C}_{ij}$. In the y sweeps we then use the Q^*_{ij} values as data for solving $\frac{\partial q}{\partial t} + \bar{v}\frac{\partial q}{\partial y} = 0$ along each column of cells with i fixed, which results in

$$Q^{n+1}_{ij} = Q^n_{ij} - \frac{\Delta t}{\Delta x}(G^*_{i,j+1/2} - G^*_{i,j-1/2}), \tag{12}$$

Fortunately, for scalar advection equation, there is no splitting error. Another benefit of this method is that fewer computations are required to solve the split system, thus reducing computational cost.Therefore, dimensional splitting gives a simple and relatively inexpensive way to extend one-dimensional high-resolution methods to two or three dimensions.

2.2.2 Simulation Study

To test the performance of the dimensional splitting method, we have set up a simulation study, which has been undertaken using Python programming language. Three schemes shall be under scrutiny in this section, upwind method,

Table 1: Error results for Gaussian wave

Scheme	ϵ_2	ϵ_∞
Upwind	0.0109	0.1663
Lax-Wendroff	0.0010	0.0136
Beam-Warming	0.0007	0.0088

Table 2: Error results for Square wave

Scheme	ϵ_2	ϵ_∞
Upwind	0.0644	0.7130
Lax-Wendroff	0.0530	0.8065
Beam-Warming	0.0554	0.8204

Lax—Wendroff method and Beam—Warming method. The study will consist of different initial conditions being setup on a box of size $0 \leq x \leq, 0 \leq y \leq 1$. Within this box it is possible to set up a grid mesh, for the purpose of this study we shall keep to a square grid mesh. Each scheme will then be used to transport the initial condition around the box. At this point it is possible to quantify the error in terms of the deviation from the exact solution. The 2-norm and infinity norm are two ways to quantify the error and are defined as

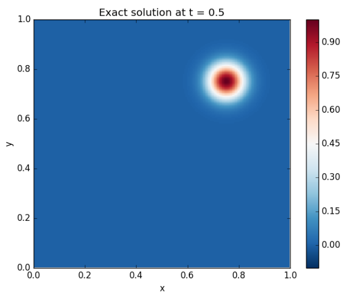
$$\epsilon_2 = \left(\frac{1}{MN} \sum_{j=1}^{M} \sum_{i=1}^{N} (Q_{ij}^{num} - Q_{ij}^{ex})^2 \right)^{1/2} \tag{13}$$

$$\epsilon_\infty = \max_{i,j} |Q_{ij}^{num} - Q_{ij}^{ex}| \tag{14}$$

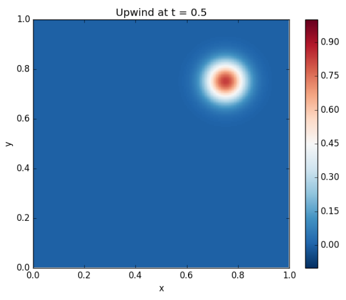
where Q_{ij}^{num} is the numerical solution and Q_{ij}^{ex} is the exact solution. The 2-norm takes into account the errors from all over the plane whereas the infinity norm is only concerned with the one point where the largest error occured. One scheme may create large errors on the wave being advected but small errors across the rest of the plane which leads to a small 2-norm, however this scheme may not necessarily be any better.

The simulation will also take into account two different types of initial condition. A curved wave such as that created by a Gaussian distribution will be the first type tested and it would be expected that the errors for this initial condition would be relatively small. There are no sudden changes from one grid point to another which can strongly affect the error. Secondly advecting a square wave will test how well each of the schemes deal with shock fronts. This type of wave can be the most difficult schemes to advect accurately, especially for schemes being tested since they are dispersive.

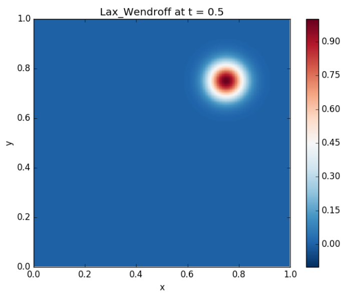
It is important to note that during this study the flow will be advecting with a positive unit velocity (i.e., $\bar{u} = 1.0, \bar{v} = 1.0$) in both directions. The grid mesh will also be set to a 200x200 grid and the Courant number is set to be 0.8.



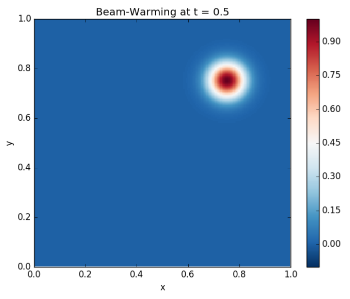
(a) True Solution at t = 0.5



(b) Upwind at t = 0.5

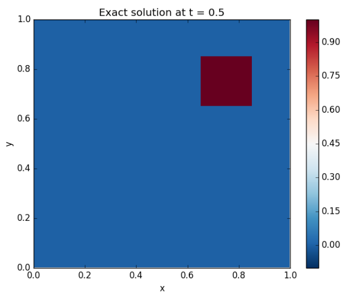


(c) Lax-Wendroff at t = 0.5

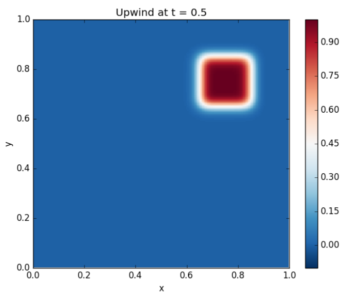


(d) Beam-Warming at t = 0.5

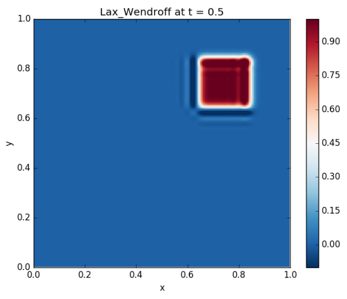
Figure 1: Gaussian wave advected



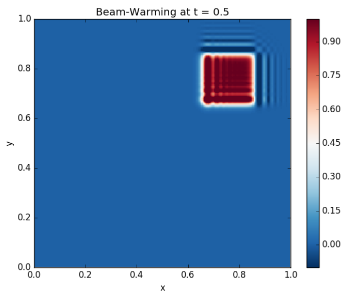
(a) True Solution at t = 0.5



(b) Upwind at t = 0.5



(c) Lax-Wendroff at t = 0.5



(d) Beam-Warming at t = 0.5

Figure 2: Square wave advected

The results obtained in table 1 and figure 1 show what occurs for a Gaussian wave. Excessive dissipation of the upwind method is evident. It is not surprising to see the upwind method is the worst among these three methods in terms of errors as it is first-order accurate while others are second-order accurate. The error results for the Lax-Wendroff schemes are close to that of Beam-Warming, but introduce slightly more error.

The error results for the Square wave are very interesting since they seem to suggest the upwind scheme is the best when it comes to reducing ϵ_∞. It is probably because the square wave gives rise to an oscillatory solution when the Lax-Wendroff and Beam-Warming methods are used. However, second-order accurate Lax-Wendroff and Beam-Warming methods are still working better from all over the domain. It may be intuitive that the second order method would give a more accurate solution, however the difference is not always that obvious.

Overall, the simulation study shows that the dimensional splitting method is indeed very easy to implement and computationally inexpensive. For smooth wave like Gaussian wave, it performs very well. Given that it is often hard to extend 1D numerical schemes into higher dimensions directly, dimensional splitting should be very useful in practice.

3 Solving the 2D Advection Equation using MPI

3.1 MPI, Python and MPI for Python[3, 5]

MPI, the Message Passing Interface, is a standardized and portable message-passing system designed to function on a wide variety of parallel computers. The standard defines the syntax and semantics of library routines and allows users to write portable programs in the main scientific programming languages (Fortran, C or C++). The paradigm of message-passing is especially suited for distributed memory architectures and is used in today's most demanding scientific and engineering applications related to modeling, simulation, design, and signal processing.

Python is an object-oriented scripting programming language. It has an extensive standard library and powerful open source third party package for scientific computing such as NumPy and SciPy. Python is powerful and efficient in working as a glue to easily integrate modules written in other languages such as Fortran, C and C++. It has a very clear, readable syntax that enables expressing the code structure, the control flow and the numerical computations very efficiently thus enhancing the programmer productivity. It supports full modularity of the code and hierarchical packages in addition to the powerful object oriented features. Therefore, Python enables a very clean, modular and extensible code design and implementation. In addition, Python has very efficient built-in support for tasks usually needed in conjunction with scientific computations such as processing files, creating graphical user interface (GUI), I/O management, text processing, network utilization and interfacing with shell commands. Also, available powerful visualization libraries such as matplotlib can be used.

MPI for Python is a general-purpose and full-featured package targeting the development of parallel applications in Python. It provides core facilities that allow parallel Python programs to exploit multiple processors. Sequential Python applications can also take advantages of MPI for Python by communicating through the MPI layer with external, independent parallel modules, possibly written in other languages like Fortran, C or C++. MPI for Python employs a back-end MPI implementation, thus being usable on any parallel environment supporting MPI.

3.2 Performance Metrics

In this section, we define the performance metrics that we have used to assess the parallel performance of our code for solving 2D scalar advection equation using MPI4Py. Those metrics are all based on the execution time of the code. To measure the execution time of parts of the running code, we have used timing-statement instrumentation. In the parallel case, the execution time measurements we have considered are those of process 0 (i.e., root process).

(1) **Speedup**

In general, the speedup gained by using a program A compared to using program B can be found by the following formula:

$$\text{Speedup of A over B} = \frac{\text{Execution time of B}}{\text{Execution time of A}}. \tag{15}$$

The performance gain obtained by a serial optimization is governed by Amdahl's Law, which states that the performance gain obtained by optimizing a portion of code is libmited by the fraction of execution time during which this portion is being executed.

In the parallel case, the definition of speedup can be specified as follows:

$$\text{Speedup} = \frac{\text{Serial execution time}}{\text{Parallel execution time}}, \tag{16}$$

for the same problem and input data. When running the program on P cores, the speedup is expected to be less than or equal to P. However, getting a speedup equal to P, a case known as linear speedup, rarely happens because of the communication overhead, process idle time and extra computations required for the parallel run[4]. In some cases, a superlinear speedup can be achieved, speedup $> P$.

(2) **Parallel Scalability and Efficiency**

Efficiency is a measure that shows resource utilization, in the very general case the total number of consumed CPU cycles over all processors participating in a simulation, in a parallel run compared to some base serial run. One formula for parallel efficiency is:

$$\text{Parallel efficiency} = \frac{\text{Speedup}}{\text{Number of cores, P}}, \tag{17}$$

In the ideal case, parallel efficiency will be equal to one, a case that indicates no overhead resulting from parallelization. However, this is most of the time not the case on parallel programs due to parallelization overhead.

Strong scalability studies demonstrate how well the code scales if we increase the number of cores while keeping the problem size constant. In the ideal case, a speedup equal to the number of cores is expected. We calculate the parallel efficiency in this case for a run on P_2 cores compared to a P_1 cores run as follows:

$$\text{Strong scaling efficiency for } P_2 \text{ cores run} = \frac{\text{Execution time on } P_1 \text{ cores}}{\text{Execution time on } P_2 \text{ cores}} \times \frac{P_1}{P_2}, \tag{18}$$

3.3 General Approach to Message-passing Parallelization

Finding numerical solutions for problems of hyperbolic PDEs mainly involves stencil operations. Restricting our discussion to the structured grid case, parallelizing a serial code for this kind of problem using message-passing parallelization model requires the following generic tasks[9]:

- Define the work load partitioning among processes.
- Perform the serial computations.
- Prepare the data vectors that need to be communicated.
- Communicate those vectors.
- Update the local data in each process using the communicated vectors.

To achieve the parallelization, we can introduce parallelizing statements, including MPI standard routines, throughout the original serial code wherever it is required. Those statements include the logic for partitioning workload and declaring vectors for communication. they also include copying required slices from the local data into the communication vectors, using MPI *send* and *receive* operations to communicate those vectors and updating the local data using the communicated vectors. The developer is responsible for all the mentioned parallelization tasks.

3.4 Parallel Solution 2D Solver for the Scalar Advection Equation using MPI4Py

In this section, we study the performance of our parallel program to solve the 2D advection equation (4) by conducting scalability experiments. We run these experiments on up to 16 cores using the Yeti High Performance Computing (HPC) Cluster at Columbia University.

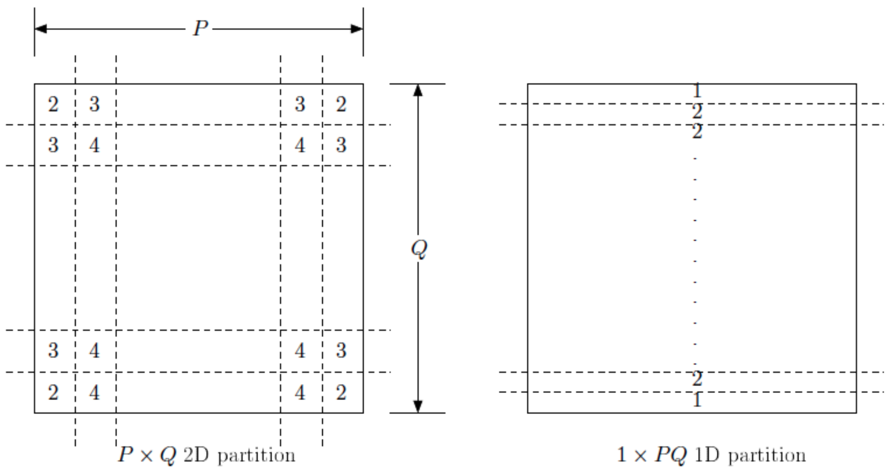


Figure 3: 1D and 2D partitions for 16 processors. The numbers indicate the number of communications with neighboring processors.

3.4.1 Work Load Partitioning

There are usually two ways to divide the computational domain: one dimensional partition and two dimensional partition. The first way is to divide the domain into a number of rectangles in x or y direction, while the second way is to divide the domain in both directions into a grid of subdomains. It would be interesting to compare the two partitions as both are natural to think of.

For general cases, assume that the $n_x \times n_y$ grid is partitioned to $P \times Q$ subgrids which can be mapped to $P \times Q$ processors. The estimated number of messages and the total amount of data for 2D $P \times Q$ partition and 1D $1 \times PQ$ partition are given below, respectively:

$$M_{P\times Q} = 4PQ - 2(P+Q), \quad D_{P\times Q} = 2(n_y P + n_x Q) - 2(n_x + n_y) \tag{19}$$

$$M_{1\times PQ} = 2(PQ-1), \quad D_{1\times PQ} = 2n_x(PQ-1) \tag{20}$$

It can be seen that the number of messages to send/receive for 2D partitioning is always more than 1D, that is, $M_{PQ} > M_{1\times PQ}$, and for the total amount of data to be transferred, unless $n_x Q < n_y$.

In our parallel program (in the github), we built a virtual topology of the size $P \times Q$ to a communicator that includes all processes, through which it is possible to build structure for the two domain partition ways we discussed above. We can take command-line arguments as input parameters to specify our Cartesian topology. Besides, building a virtual topology will also benefit us by increasing the execution performance since every node will communicate only with its virtual neighbor. For example, if the rank was randomly assigned, a message could be forced to pass to many other nodes before it reaches the destination. Beyond the question of performance, a virtual topology makes sure that the code is more clear and readable.

3.4.2 Updating and Communicating

The numerical scheme we used is upwind method with dimensional splitting. Note that before updating cell averages, we have to first update ghost cells in each process and do edge cells communications between neighboring subdomains. Moreover, For some grid structure like 3 by 3 or 4 by 4 domain partition, we even need to communicate cells from diagonal blocks.

3.4.3 Yeti Architecture

Here we provide some technical information about the architecture of the Yeti HPC Cluster, the platform where we conduct our experiments. The original Yeti cluster contained two submit nodes, a storage server, and 101 execute servers with a total of 1616 compute cores.These execute servers have dual 1.8 GHz Intel E5-2650L Xeon CPU with 8 cores each for a total of 16 cores per server. In February, 2015, 66 additional servers were added to the original Yeti cluster, bringing the total server count to 167 execute servers with a total of 2762 compute cores. The CPU on all expansion machines is dual 2.6 GHz Intel E5-2650v2 Xeon. We conducted our experiments on one 2nd generation node (dual 2.6 GHz Intel E5-2650v2 Xeon) with 16 cores and 16000mb memory.

3.4.4 Timing Approaches

To get timing measurements, we use the function *MPI.Wtime()* from the MPI4Py package. To get the total job time, we use the *time* command in the terminal. In the strong scalability of the advection problem, the grid size is fixed for the runs as 2000 by 2000. We evolve the solution until time $t = 0.5$. To focus on the scalability of the concurrent computations of the program, we temporarily only present the execution time of that part for each grid topology. Additionally, we experiment 5 times for each grid topology and take the average of them as our final execution time. For some number of processors with two possible grid topologies, we experiment both of them to figure out which partition would be better.

3.4.5 Results and Analysis

The results obtained in table 3 and figure 4 show the performance of the concurrent part of the parallel program using MPI4Py. We see that the efficiency degrades quickly as the number of cores increases. This degradation is probably due to the computational overheads and the reduction of the grid partition size per core as we increase the number of cores. Moreover, we could notice that two dimensional domain decomposition performs better than one dimensional partition for all comparisons. It is not so surprising because two dimensional partition has better load-balancing, meaning the number of grid cells along each axis is closer as opposed to the case in one dimensional partition.

Table 3: Performance of the parallel program using MPI4Py

Number of Cores	Cartesian Topology	Concurrent Computations (s)	Speedup	Efficiency
1	1×1	126.447	1.0	100%
2	1×2	69.824	1.81	90.5%
4	1×4	49.44	2.56	64.0%
4	2×2	45.723	2.76	69.1%
6	1×6	38.753	3.26	54.3 %
6	2×3	35.221	3.59	59.8 %
9	1×9	33.684	3.75	41.7%
9	3×3	31.609	4.00	44.4%
12	1×6	30.875	4.10	34.1 %
12	3×4	28.027	4.51	37.5%
16	1×16	26.823	4.71	29.4%
16	4×4	24.194	5.23	32.7%

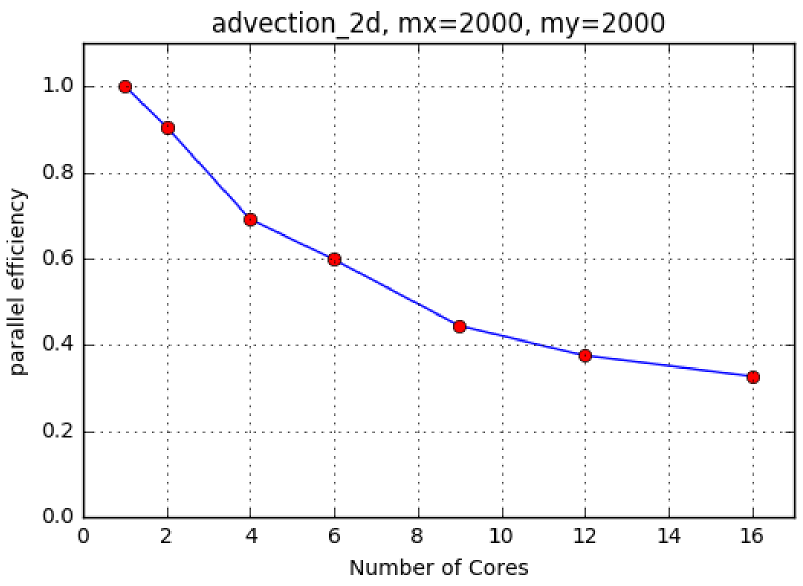


Figure 4: Strong scalability efficiency for the advection equation

4 Solving the 2D Advection Equation using OpenMP

In this section, we exploit Clawpack, Conservation Laws Package, to test the performance of the parallel solution of the 2D advection equation on annulus using OpenMP.

4.1 Clawpack[6]

Clawpack is a package that is developed to solve time-dependent linear and nonlinear systems of hyperbolic PDEs representing conservation laws in up to three space dimensions. It implements state-of-the-art high-resolution explicit Godunov-typ methods that are a class of finite volume methods. These methods require Riemann solvers to resolve the jump discontinuity at the interface between two grid cells into waves propagating into the neighboring cells. Clawpack is flexible in the sense that it can handle all kinds of hyperbolic PDEs. It can also solve hyperbolic problems that are not in the conservative form.

This package has been actively developed, maintained, extended and documented since 1994 by a number of developers. The source code of the package is available free of charge for research and instructional purposes. Clawpack has been used by many users to simulate a wide variety of phenomena and and physical systems related to industrial, engineering and technological problems, such as applications related to traffic flow, jet flow, tsunami modeling and semiconductor device simulations.

To set an application using Clawpack, the user provides the problem setup data and the required Riemann solver for the problem. The user also needs to provide any other problem-specific code for updating the boundary conditions of the solution q, initializing the solution, setting the system coefficients and /or adding any extra logic required at each time step.

4.2 OpenMP[7]

OpenMP is a shared-memory application programming interface (API) whose features are based on prior efforts to facilitate shared-memory parallel programming. Rather than a new programming language, it is notation that can be added to a sequential program in Fortran, C, or C++ to describe how the work is to be shared among threads that will execute on different processors or cores and to order accesses to shared data as needed. The appropriate insertion of OpenMP features into a sequential program will allow many, perhaps most, applications to benefit from shared-memory parallel architectures-often with minimal modification to the code. In practice, many applications have considerable parallelism that can be exploited.

OpenMP's directives let the user tell the compiler which instructions to execute in parallel and how to distribute them among the threads that will run the code. An OpenMP directive is an instruction in a special format that is understood by OpenMP compilers only. The first step in creating an OpenMP program from a sequential one is to identify the parallelism it contains. Basically, this means finding instructions, sequences of instructions, or even large regions of code that may be executed concurrently by different processors. The second step in creating an OpenMP program is to express, using OpenMP, the parallelism that has been identified. A huge practical benefit of OpenMP is that it can be applied to incrementally create a parallel program from an existing sequential code.

4.3 The advection equation on annulus

The PDE we are going to test is the advection equation for 2D solid body rotation in an annulus with period 1:

$$\frac{\partial q}{\partial t} + u(x,y)\frac{\partial q}{\partial x} + v(x,y)\frac{\partial q}{\partial y} = 0 \tag{21}$$

The velocity field is

$$u(x,y) = -2\pi y, \quad v(x,y) = 2\pi x \tag{22}$$

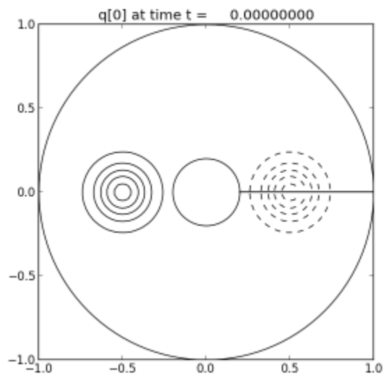
4.4 Adding OpenMP parallelism to the 2D solver

We can add OpenMP directives in the Clawpack Fortran Classic 2d solver for making use of multicore shared memory machines. In fortran codes step2.f90 and step2ds.f90, we can add OpenMP directives to the outer loop of the block where x-sweeps or y-sweeps initiates. Fox example, for x-sweeps, we can express the parallelism using *!$omp parallel do* directive together with *reduction* directive.That is because except for the operation to obtain the global CFL number by taking the largest CFL number of each loop iteration, other operations don't interact with their counterparts in other loop iterations. The parallelized step2.f90 and step2ds.f90 can be found in the github.

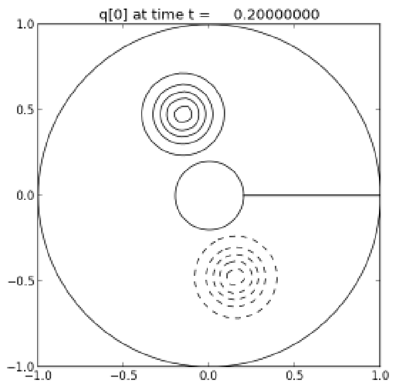
4.5 Scalability testing of the shared-memory parallelism

In this section, we study the performance of the parallel subrountines step2.f90 and step2ds.f90 by solving the PDE in the section 2.4.3. Again, we use the same performance metrics discussed in the section 2.3.2. Since the parallelized part is one component of the 2D solver, to test the scalability efficiency of the shared-memory parallelism we added, we have to time the section where OpenMP directives have been added. We use the function *omp_get_wtime()* from *omp-lib* library for timing purpose. To obtain the reasonable execution time, we sum all execution time over one frame and then divide it by the number of time steps contained in that frame to get the execution time per time step. To test the strong scalability, we fix the grid size as 400 × 1200.

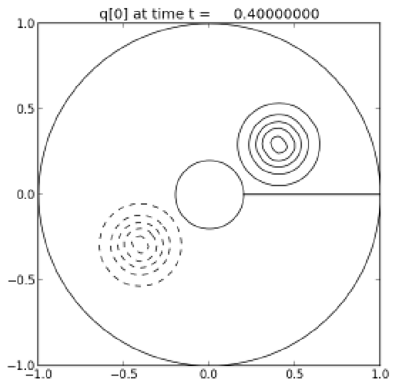
The experiment is also conducted on the Yeti HPC cluster. But It is important to note that rather than running jobs in batch mode, we run jobs in



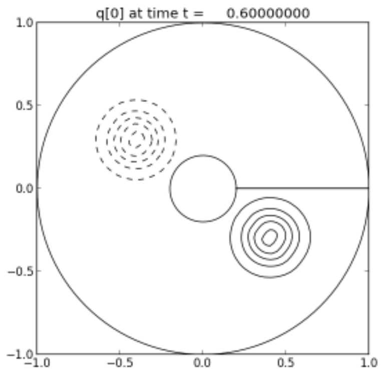
(a) Initial data



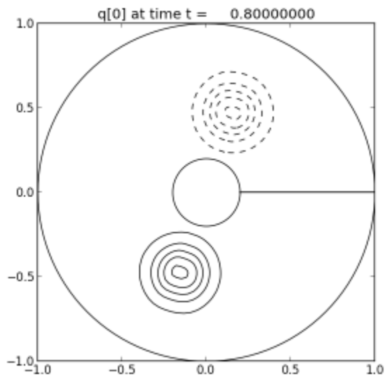
(b) Numerical solution at t = 0.2



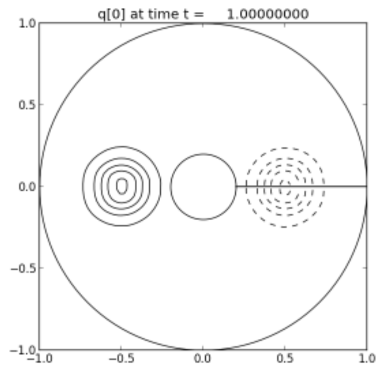
(c) Numerical solution at t = 0.4



(d) Numerical solution at t = 0.6



(e) Numerical solution at t = 0.4



(f) Numerical solution at t = 1.0

Figure 5: 6 plots of numerical solution of the advection equation on annulus in Clawpack examples

Table 4: Performance of the parallel part using OpenMP

Number of Threads	Execution Time (s/per time step)	Speedup	Efficiency
1	0.204	1.0	100%
2	0.107	1.90	95.0%
3	0.072	2.83	94.3%
4	0.055	3.71	92.7%
5	0.045	4.53	90.6 %
6	0.038	5.36	89.3%
7	0.033	6.18	88.3%
8	0.029	7.03	87.9%

interactive mode on Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz with memory 16000mb.

The results obtained in table 4 and figure 6 show the parallelism we expressed has gained good speedup as we increase the number of threads, although we can notice a small drop in efficiency, which should be caused by computational overheads when more threads are used.

5 Conclusions and Future Directions

In this section, we summarize the key conclusions of this report and give plans for further improvements and applications.

5.1 Concluding Remarks

(1) The dimensional splitting method is a very useful method for extending 1D schemes to multidimensional schemes in practice. But we still need to be careful about the potential errors it may cause when problems become complicated.

(2) To use MPI for Python package directly to achieve parallelization may not be able to gain good speedup as low-level parallelization may lead to large computational overheads.

(3) The parallelism we introduce to the 2D solver in Clawpack has gained good speedup.

5.2 Future Directions

(1) Rather than using MPI4Py, we could instead use PETSc4Py to achieve better parallelization by avoiding low-level parallelization details. PETSs employs the MPI standards for parallelism. It is intended to be used in

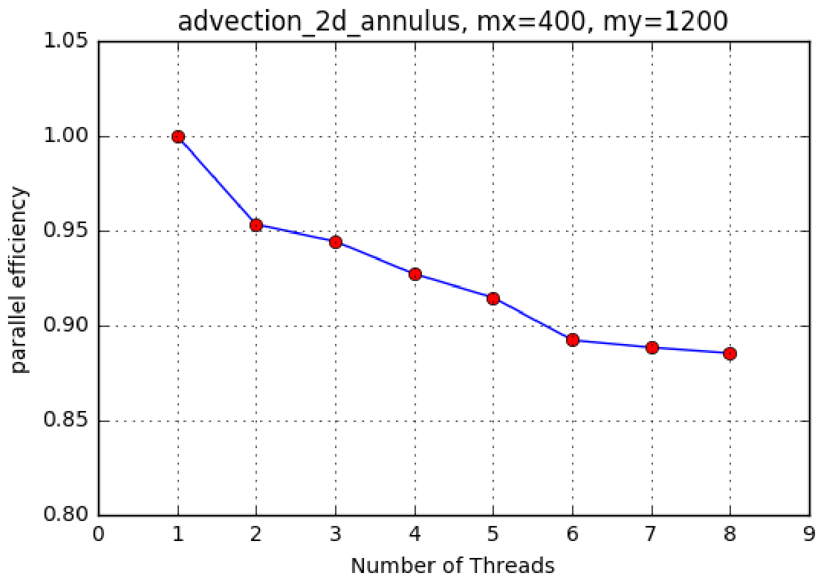


Figure 6: Strong scalability efficiency for the parallel part using OpenMP

large-scale computational applications and has proved to be scalable in several projects.

(2) As there is no parallel 2D solver in Pyclaw, we could add the shared-memory parallelism to the 2D solver in Pyclaw.

References

[1] “Mathematics of advection,” [accessed 12/27/2016], [Online]. Available: http: https://en.wikipedia.org/wiki/Advection

[2] R. J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, Cambridge: Cambridge University Press, 2002.

[3] L. Dalcin, P. Kler, R. Paz, and A. Cosimo, *Parallel Distributed Computing using Python*, Advances in Water Resources, 34(9):1124-1139, 2011.

[4] P. S. Pacheco, *Parallel Programming with MPI*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[5] Lisandro Dalcin, *MPI for Python 2.0.0 documentation*, [Online]. Available: http://pythonhosted.org/mpi4py/usrman/index.html

[6] R. J. Leveque, “Clawpack 5.3.1 documentation,” [accessed 12/27/2016], [Online]. Available: http://www.clawpack.org/contents.html

[7] Barbara Chapman; Gabriele Jost; Ruud van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, Scientific Programming, MIT Press 2007.

[8] C. G. Bell, “’The future of high performance computers in science and engineering,’ *Commun.ACM*, vol. 32, pp. 1091-1101, September, 1989.

[9] X. Cai, H. Petter, and H. Moe,“On the performance of the Python programming language for serial and parallel scientific computations,” *Scientific Programming*, vol. 13, no 1, pp. 31-56, 2005.