The MiniWeather Mini App

Matt Norman, Oak Ridge National Laboratory

Contents

1 Introduction

The miniWeather code is a self-contained compressible fluid code that mimics the basic dynamics seen in atmospheric weather and climate. The dynamics themselves are dry compressible, stratified, non-hydrostatic flows dominated by buoyant forces that are relatively small perturbations on a hydrostatic background state. The equations simulated in this code form the backbone of pretty much all fluid dynamics codes, and this particular flavor forms the base of all weather and climate modeling.

With only about 500 total lines of code (and only about 200 lines that you care about), it serves as an approachable place to learn parallelization and porting using MPI + X, where X is OpenMP, OpenACC, CUDA, or potentially other approaches to CPU and accelerated parallelization.

The code defaults to a domain of 20km $\times$ 10km in the x- and z-directions, but you can feel free to change that if you wish. Beware, though, that the vertical length can only get so large before the hydrostatic initialization gives invalid values. Also, the troposphere doesn't extend much beyond 10km. So it's probably best to leave the domain as it is.

The code uses periodic boundary conditions in the x-direction and solid wall boundary conditions in the z-direction.

2 Compiling and Running the Code

2.1 Required Libraries

To run everything in this code, you need MPI, parallel-netcdf, ncview, and an OpenACC compiler (preferably PGI 17+) to be installed. The serial code still uses MPI because I figured it was best to start to parallel I/O in place so that it's ready to go when the MPI implementation is finished.

You can download parallel-netcdf from:
`https://trac.mcs.anl.gov/projects/parallel-netcdf`
Also, you can download a free version of PGI Community Edition from:
`https://www.pgroup.com/products/community.htm`
You'll probably also want to have "ncview" installed so you can easily view your NetCDF files:
`http://meteora.ucsd.edu/~pierce/ncview_home_page.html`

2.2 Compiling the Code

To compile the code, first edit the Makefile and change the flags to point to your parallel-netcdf installation as well as change the flags based on which compiler you are using. There are four versions of the code: serial, mpi, mpi+openmp, and mpi+openacc. The filenames are clear which file is associated with which programming paradigm. To make all of these at once, simply type "make". To make them individually, you can type "make serial", "make mpi", "make openmp", and "make openacc".

2.3 Altering the Code's Configurations

There are four aspects of the configuration you can edit easily, and they are clearly labeled in the code as "USER-CONFIGURABLE PARAMETERS". These include: (1) the number of cells to use "nx_glob" and "nz_glob"; (2) the amount of time to simulation "sim_time"; (3) the frequency to output data to file "output_freq"; and (4) the initial data to use "data_spec_int".

2.4 Running the Code

To run the code, simply call "mpirun -n [# ranks] ./mini_weather_[version]". Since parameters are set in the code itself, you don't need to pass any parameters. Some machines use different tools instead of mpirun.

2.5 Viewing the Output

The file I/O is done in the netCDF format: (`https://www.unidata.ucar.edu/software/netcdf/`). To me, the easiest way to view the data is to use a simple tool called "ncview" (`http://meteora.ucsd.edu/~pierce/ncview_home_page.html`). To use it, you can simply type "ncview output.nc", making sure you have X-forwarding enabled in your ssh session.

3 Parallelizing the Code

This was somewhat designed to parallelize MPI first and then OpenMP or OpenACC next, but you can always parallelize with OpenMP or OpenACC without MPI if you want. But it is rewarding to be able to run it on multiple nodes at higher resolution for sharper eddies in the dynamics.

As you port the code, you'll want to change relatively little code at a time, re-compile, re-run, and look at the output to see that you're still getting the right answer. There are advantages to using a visual tool to check the answer, as it can sometimes give you clues as to why you're not getting the right answer.

Note that you only need to make changes to the first 450 lines of code. Everything below that is simply initialization and I/O code that doesn't need to be parallelized (unless you want to). There's a note in the serial code that tells you at what point the rest of the code will remain the same.

3.1 MPI

This code was designed to use domain decomposition, where each MPI rank "owns" a certain set of cells. You'll notice that the data structure, "state" has made room for a number of "halo". The Fortran code has the fluid state dimensioned from 1-hs:nx+hs in the x-direction, for instance. This means that the cells from 1:nx are owned by that MPI rank. But the cells from 1-hs:0 are owned by the left-neighboring MPI rank, and nx+1:nx+hs are owned by the right-neighboring MPI rank. The domain is only decomposed in the x-direction and not the z-direction. IMPORTANT: Please be sure to set "nranks", "myrank", "nx", "i_beg", “left_rank”, and “right_rank”. You can set more variables, but these are used elsewhere in the code (particularly in the parallel file I/O).

To parallelize with MPI, there are only two places in the code that need to be altered. The first is the initialization (subroutine named “init”), where you must determine the number of ranks, you process’s rank, the beginning index of your rank’s first cell in the x-direction, the number of x-direction cells your rank will own, and the rank IDs that are to your left and your right. Because the code is periodic in the x-direction, your left and right neighboring ranks will wrap around. For instance, if your are rank zero, your left-most rank will be nranks-1.

The second place is in the routine that sets the halo values in the x-direction. In this routine, you need to:

1. Create MPI data buffers to hold the data that needs to be sent and received, allocate them in the init() routine, and deallocate them in the finalize() routine.
2. Pack the data you need to send to your left and right MPI neighbors
3. Send the data to your left and right MPI neighbors
4. Receive the data from your left and right MPI neighbors
5. Unpack the data from your left and right neighbors and place the data into your MPI rank’s halo cells.

Once you complete this, the code will be fully parallelized in MPI. Both of the places you need to add code for MPI are marked in the serial code, and there are some extra hints in the “set_halo_values_x()” routine as well.

3.2 OpenMP

For the OpenMP code, you basically need to decorate the loops with OpenMP and pay attention to any variables you need to make “private” so that each thread has its own copy. There are no race conditions in this particular code, so there is no need for critical regions, though that is something you often need in an OpenMP code. Keep in mind that OpenMP works best on “outer” loops rather than “inner” loops. Also, for sake of performance, there are a couple of instances where it is wise to use the “collapse” clause because the outermost loop is not very large.

3.3 OpenACC

The OpenACC approach will differ depending on whether you’re in Fortran or C. Just a forewarning, OpenACC is much more convenient in Fortran when it comes to data movement because in Fortran, the compiler knows how big your arrays are, and therefore the compiler can (and does) create all of the data movement for you. All you have to do is optimize the data movement after the fact.

3.3.1 Fortran Code

The OpenACC parallelization is a bit more involved when it comes to performance. But, it's a good idea to just start with the kernels themselves, since the compiler will generate all of your data statements for you on a per-kernel basis. You need to pay attention to private variables here as well.

Once you're getting the right answer with the kernels on the GPU, you can look at optimizing data movement. I recommend putting data statements for the five main arrays as well as the MPI buffers around the main time stepping loop. Then, you simply need to move the data to the host before sending MPI data, back to the device once you receive MPI data, and to the host before file I/O.

You might also find some benefit from using the async() clause to reduce kernel launch overheads. This will only matter if you have relatively little data per node.

3.3.2 C/C++

To be written.

3.4 CUDA

To be written.

4 Numerical Experiments

A number of numerical experiments are in the code for you to play around with. You can set these by changing the "data_spec_int" variable.

4.1 Rising Thermal

data_spec_int = DATA_SPEC_THERMAL
sim_time = 1000
This simulates a rising thermal in a neutral atmosphere, which will look something like a "mushroom" cloud (without all of the subsequent violence).

4.2 Colliding Thermals

data_spec_int == DATA_SPEC_COLLISION
sim_time = 600
This is similar to the rising thermal test case except with a cold bubble at the model top colliding with a warm bubble at the model bottom to produce some cool looking eddies.

4.3 Mountain Gravity Waves

data_spec_int == DATA_SPEC_MOUNTAIN
sim_time = 1500

This test cases passes a horizontal wind over a faked mountain at the model bottom in a stable atmosphere to generate a train of stationary gravity waves across the model domain.

4.4 Turbulence

data_spec_int == DATA_SPEC_TURBULENCE
sim_time = 1000
This test case creates a neutrally stratified atmosphere with randomly initialize winds. After a few time steps, the random noise is locally averaged out to lower values. Keep in mind that this is not classic turbulence in any sense because it's only 2-D, and it's stratified.

4.5 Density Current

data_spec_int == DATA_SPEC_DENSITY_CURRENT
sim_time = 600
This test case creates a neutrally stratified atmosphere with a strong cold bubble in the middle of the domain that crashes into the ground to give the feel of a weather front (more of a downburst, really).

5 Physics and Numerical Approximation

While these numerics are certainly cheap and dirty, they are probably the fastest way to get the job done at a moderate order of accuracy. For instance, on 16 K20x GPUs, you can perform a simulation with 5 million grid cells (3200 × 1600) in just a few minutes.

5.1 The 2-D Euler Equations

This app simulates the 2-D inviscid Euler equations for stratified fluid dynamics, which are defined as follows:

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u\\ \rho w\\ \rho\theta\end{bmatrix}+\frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uw\\ \rho u\theta\end{bmatrix}+\frac{\partial}{\partial z}\begin{bmatrix}\rho w\\ \rho wu\\ \rho w^2+p\\ \rho w\theta\end{bmatrix}=\begin{bmatrix}0\\ 0\\ -\rho g\\ 0\end{bmatrix} \tag{1}$$

$$\rho_H=-\frac{1}{g}\frac{\partial p}{\partial z}$$

where ρ is density, u, and w are winds in the x-, and z-directions, respectively, θ is potential temperature related to temperature, T, by $\theta = T\left(P_0/P\right)^{R_d/c_p}$, $P_0 = 10^5\,\mathrm{Pa}$, $g = 9.8\ \mathrm{m\,s^{-2}}$ is acceleration due to gravity, $p = C_0\left(\rho\theta\right)^{\gamma}$ is pressure, $C_0 = R_d^{\gamma}p_0^{-R_d/c_v}$, $R_d = 287\,\mathrm{J\,kg^{-1}\,K^{-1}}$, $\gamma = c_p/c_v$, $c_p = 1004\,\mathrm{J\,kg^{-1}\,K^{-1}}$, $c_v = 717\,\mathrm{J\,kg^{-1}\,K^{-1}}$, and $p_0 = 10^5\,\mathrm{Pa}$. This can be cast in a more convenient form as:

$$\frac{\partial \mathbf{q}}{\partial t}+\frac{\partial \mathbf{f}}{\partial x}+\frac{\partial \mathbf{h}}{\partial z}=\mathbf{s} \tag{2}$$

5.2 Maintaining Hydrostatic Balance

The flows this code simulates are relatively small perturbations off of a "hydrostatic" balance, which balances gravity with a difference in pressure:

$$\frac{dp}{dz} = -\rho g$$

Because small violations of this balance lead to significant noise in the vertical momentum, it's best not to try to directly reconstruct this balance but rather to only reconstruct the perturbations. Therefore, hydrostasis is subtracted from (1) to give:

$$\frac{\partial}{\partial t}\begin{bmatrix} \rho' \\ \rho u \\ \rho w \\ (\rho\theta)' \end{bmatrix} + \frac{\partial}{\partial x}\begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u w \\ \rho u \theta \end{bmatrix} + \frac{\partial}{\partial z}\begin{bmatrix} \rho w \\ \rho w u \\ \rho w^2 + p' \\ \rho w \theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\rho' g \\ 0 \end{bmatrix} \tag{3}$$

where a "prime" quantity represents that variable with the hydrostatic background state subtracted off.

5.3 Dimensional Splitting

This equation is solved using dimensional splitting for simplicity and speed. The equations are split into x- and z-direction solves that are, respectively:

$$x: \qquad \frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{f}}{\partial x} = \mathbf{0}$$

$$z: \qquad \frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{h}}{\partial x} = \mathbf{s}$$

Each time step, the order in which the dimensions are solved is reversed, giving second-order accuracy overall.

5.4 Finite-Volume Spatial Discretization

A Finite-Volume discretization is used in which the PDE in a given dimension is integrated over a cell domain, $\Omega_i \in \left[x_{i-1/2}, x_{i+1/2}\right]$, where $x_{i\pm1/2} = x_i \pm \Delta x$, x_i is the cell center, and Δx is the width of the cell. The integration is the same in the z-direction. Using the Gauss divergence theorem, this turns the equation into (using the z-direction as an example):

$$\frac{\partial \overline{\mathbf{q}}_{i,k}}{\partial t} = -\frac{\mathbf{h}_{i,k+1/2} - \mathbf{h}_{i,k-1/2}}{\Delta z} + \overline{\mathbf{s}}_{i,k} \tag{4}$$

where $\overline{\mathbf{q}}_{i,k}$ and $\overline{\mathbf{s}}_{i,k}$ are the cell-average of the fluid state and source term over the cell of index i, k.

To compute the update one needs the flux vector at the cell interfaces and the cell-averaged source term. To compute the flux vector at interfaces, fourth-order-accurate polynomial interpolation is used using the four cell averages surrounding the cell interface in question.

5.5 Runge-Kutta Time Integration

The equation (4) still has a time derivative that needs integrating, and a simple low-storage Runge-Kutta ODE solver is used to integrate the time derivative, and it is solved as follows:

$$\mathbf{q}^{\star} = \mathbf{q}^{n} + \frac{\Delta t}{3} RHS\left(\mathbf{q}^{n}\right)$$

$$\mathbf{q}^{\star\star} = \mathbf{q}^{n} + \frac{\Delta t}{2} RHS\left(\mathbf{q}^{\star}\right)$$

$$\mathbf{q}^{n+1} = \mathbf{q}^{n} + \Delta t RHS\left(\mathbf{q}^{\star\star}\right)$$

5.6 “Hyper-viscosity”

The centered fourth-order discretization in Section 5.4 is unstable as is and requires extra dissipation to damp out small-wavelength energy that would otherwise blow up the simulation. This damping is accomplished with a scale-selective fourth-order so-called “hyper”-viscosity that is defined as:

$$\frac{\partial \mathbf{q}}{\partial t} + \frac{\partial}{\partial x}\left(-\kappa \frac{\partial^3 \mathbf{q}}{\partial x^3}\right) = \mathbf{0}$$

and this is also solved with the Finite-Volume method just like above. The hyperviscosity constant is defined as:

$$\kappa = -\beta \left(\Delta x\right)^4 2^{-4} \left(\Delta t\right)^{-1}$$

where $\beta \in [0, 1]$ is a user-defined parameter to control the strength of the diffusion, where a higher value gives more diffusion.