How to Parallelize Your Code Using EZ_PARALLEL

November 19, 2019

1 What is EZ_PARALLEL?

EZ_PARALLEL will help you parallelize your serial code! It lets you do this by *adding only about 10 new lines to your code!* Beyond those 10 new lines, additional MPI code is also needed, but it has already been written for you!

As a starting point, we expect the flow of a *serial* finite difference code to be as follows:

1. Initialize simulations parameters, e.g., the size of the array containing the domain (with allocating memory to that array), setting physical constants, etc.
2. Initializing the domain grid. This includes allocating memory to the array containing the domain and setting the initial condition.
3. Step forward in time. This includes calculating each subsequent time step and outputting the domain grid to a file.

This is not to say that every code must be in the format to use EZ_PARALLEL, but it is the basic form that we expect any basic serial code to be in.

2 Basic Instructions to Use EZ_PARALLEL

1. **Obtaining and Compiling EZ_PARALLEL.** Download the EZ_PARALLEL directory, and place it in the same directory as your serial code. Open the file named *makefile* with a text editor (e.g. Notepad or equivalent). If running on Linux or Mac, delete the "#" symbols from the lines

   ```
   #FC = mpifort
   #INPTHS =
   #INLIBS =
   #LKLIBS =
   ```

   If running on Windows, delete the "#" symbols from the lines

   ```
   #FC = gfortran
   #INPTHS = -IC:\lib\mpi\include
   #INLIBS = -LC:\lib\mpi\lib
   #LKLIBS = -lmsmpi
   ```

   In your command window, navigate to the EZ_PARALLEL directory and run the MAKE command. This will compile the library and allow you to use it in your code.

   If you need to re-compile EZ_PARALLEL at a future time: If running on Linux/Mac, remove the "#" symbol from the line

```
# rm -f -r *.o *.a *.so
```

under the lines

```
clean:
# Linux
```

If running on Windows, remove the "#" symbol from the line

```
# del *.o *.a *.so
```

under the lines

```
clean:
# Linux
#     rm -f -r *.o *.a *.so
# Windows
```

Then run MAKE CLEAN followed by MAKE to re-compile EZ_PARALLEL.

2. **Ensuring Proper Installation.** In your code, add USE MPI before the IMPLICIT NONE statement. Add CALL INIT_MPI immediately after declaring your variables, and FIN_MPI immediately before END PROGRAM. This will initialize and terminate MPI.

   Compile your code by adding three flags to the compilation command: For Linux/Mac (with MPIFORT), use the following command

   ```
   mpifort -o out.exe my_mpi_program.f90 -I.\EZ_PARALLEL -L.\EZ_PARALLEL -lEZ_PARALLEL
   ```

   For Windows, use the following command (on a single line)

   ```
   gfortran -o out.exe my_mpi_program.f90 -I.\EZ_PARALLEL -L.\EZ_PARALLEL
   -lEZ_PARALLEL -IC:\lib\mpi\include -LC:\lib\mpi\lib -lmsmpi
   ```

   where you replace OUT.EXE with the desired name of your executable file and MY_MPI_PROGRAM.F90 with the name of your .F90 file.

   To run/execute your code, the execution commands are slightly different in different settings.

   - If you compile the code using mpifort (Linux, Mac), then the execution command is probably

     ```
     mpirun -np 2 your_executable.exe
     ```

     where the number 2 is included to specify that you want to use 2 processors (or cores). If you want 1 core, then change the 2 to a 1. If you want 4 cores, then change the 2 to a 4.
   - If you compile the code using gfortran with the MPI library (Windows), then the execution command is probably

```
mpiexec -np 2 your_executable.exe
```

where the number 2 is included to specify that you want to use 2 processors (or cores). If you want 1 core, then change the 2 to a 1. If you want 4 cores, then change the 2 to a 4.

Run your code with one processor (using a small grid/small number of timesteps so it runs quickly). You should receive a message before your program runs that states "EZ_PARALLEL: Only one processor is in use. All subroutines in EZ_PARALLEL will be skipped over."

Run your code with two processors (using a small grid/small number of timesteps so it runs quickly). Your entire code will be run by two processors, which means all PRINT and WRITE statements will appear twice.

3. **Parallelizing Your Code.** For the purpose of discussion, we will assume that your code is in the general form of the following psuedocode

```
DECLARE VARIABLES/INITIALIZE PARAMETERS

INITIALIZE GRID

TIME STEP
```

DECLARE VARIABLES/INITIALIZE PARAMETERS includes declaring variable types and values, and reading in parameter values from an input file, e.g., a NAMELIST. INITIALIZE GRID allocates memory to the simulation grid and fills in the initial condition using a function and a reference point. For example, in a code for numerically solving the heat equation, the grid could contain the temperature at each point. TIME STEP progresses the simulation forward in time and writes the grid to an output file every so often.

In short, the lines we will add to the code are

```
USE MPI

DECLARE VARIABLES/INITIALIZE PARAMETERS

CALL INIT_MPI
CALL DECOMP_GRID(y_len, overlap)
CALL IDENTIFY_REF_POINT(y_len, y_ref, dy, overlap)

INITIALIZE GRID

TIME STEP

CALL FIN_MPI
```

where the USE and CALL lines are verbatim, calling subroutines from EZ_PARALLEL.

In greater detail:

- USE MPI will allow your code to use MPI.
- CALL INIT_MPI allows us to use MPI commands later in our code.

- CALL GRID_DECOMP(Y_LEN, OVERLAP) splits up the simulation grid into several subgrids, one for each processor. Y_LEN is the length of the y-direction (second dimension) of the grid, while OVERLAP reflects the number of points the timestepping scheme requires. For example, the centered-difference scheme

$$U_i^{n+1} = \frac{1}{h^2}\left(U_{i+1}^n - 2\,U_i^n + U_{i-1}^n\right)$$

  utilizes one point U_{i+1}^n to the right of U_i^n (or one point U_{i-1}^n to the left of U_i^n), and so we would use OVERLAP $= 1$.
- CALL IDENTIFY_REF_POINT(Y_LEN, Y_REF, DY, OVERLAP) calculates the reference point for each subgrid. Y_REF is the y-coordinate of the original reference point and DY is the grid spacing in the y-direction.
- Within INITIALIZE GRID, add CALL SHARE_SUBGRID_BOUNDARIES(X_LEN, Y_LEN, OVERLAP, GRID) after filling in the initial condition and boundaries. X_LEN is the length of the grid in the x-direction (first dimension), and GRID is the simulation grid. This ensures that the subgrids are properly initialized.

  Your original (serial) code would look like

```
ALLOCATE MEMORY TO GRID
FILL IN INITIAL CONDITION
FILL IN BOUNDARY CONDITIONS
```

  Your modified (parallel) code would look like

```
ALLOCATE MEMORY TO GRID
FILL IN INITIAL CONDITION
FILL IN BOUNDARY CONDITIONS
CALL SHARE_SUBGRID_BOUNDARIES(x_len, y_len, overlap, grid)
```

- Within TIME STEP, add CALL SHARE_SUBGRID_BOUNDARIES(X_LEN, Y_LEN, OVERLAP, GRID) after calculating the values on GRID for the new timestep.

  Your original (serial) code would look like

```
DO WHILE STEPPING FORWARD IN TIME
  TAKE TIME STEP

  IF WANT TO OUTPUT GRID
    OUTPUT GRID
  END IF
END DO
```

  Your modified (parallel) code would look like

```
DO WHILE STEPPING FORWARD IN TIME
  TAKE TIME STEP
  CALL SHARE_SUBGRID_BOUNDARIES(x_len, y_len, overlap, grid)

  IF WANT TO OUTPUT GRID
    OUTPUT GRID
  END IF
END DO
```

Additionally, you will have to modify your OUTPUT subroutine, such that each processor will output their subgrid to a file with a unique name. For example, in a serial code the output file name FNAME may be assigned using

```
WRITE(fname,"(A,I0.8,A)") "./output_data/out_", step, ".dat"
```

where STEP is the timestep number. To give each processor a unique file name, we can create an integer variable PROC_ID to store the processor's ID number. To obtain this value, add CALL GET_ID(PROC_ID) to your code. The output file name in the parallel code could then be

```
WRITE(fname,"(A,I0.8,A,I0.4,A)") "./output_data/out_", step, "_", proc_id, ".dat"
```

- CALL FIN_MPI finalizes MPI and stops all but a single processor. For example, if you wish to output the time it took for your code to run, add CALL FIN_MPI before you write the execution time to the screen so that it only outputs a single time.

Your code should now be ready to compile and run.

To run/execute your code, the execution commands are slightly different in different settings.

- If you compile the code using mpifort (Linux, Mac), then the execution command is probably

  ```
  mpirun -np 2 your_executable.exe
  ```

  where the number 2 is included to specify that you want to use 2 processors (or cores). If you want 1 core, then change the 2 to a 1. If you want 4 cores, then change the 2 to a 4.
- If you compile the code using gfortran with the MPI library (Windows), then the execution command is probably

  ```
  mpiexec -np 2 your_executable.exe
  ```

  where the number 2 is included to specify that you want to use 2 processors (or cores). If you want 1 core, then change the 2 to a 1. If you want 4 cores, then change the 2 to a 4.