RAYLEIGH TUTORIAL

MODULE 3: RUNNING THE CODE & BEST PRACTICES

IN THIS MODULE:

- Changing Run Resolution
- Changing Process Count and Layout
- Controlling Run Timing
- Checkpointing

BEFORE WE BEGIN:

- Prepare a run directory named module2

```
$ mkdir ~/module2
$ cd ~/module2
$ ln –s ~/Rayleigh-1.2.0/bin/rayleigh.opt    .
$ cp ~/Rayleigh-1.2.0/input_examples/c2001_case0_input main_input
```

- Edit main_input (and save; nano, ctrl+o , ctrl+x):

  DELETE LINE: benchmark_mode = 1

  EDIT LINE: max_iterations = 1000 (was 100000)

  EDIT LINES: nprow = 2 & npcol = 2 (were 16 & 32)

GRID RESOLUTION AND DOMAIN BOUNDS

- N_theta OR l_max control # of angular gridpoints
- N_r controls # of radial gridpoints
- Radial domain bounds controlled by

  { rmin, rmax } OR { shell_depth , aspect_ratio }
- Access these via the problemsize_namelist

```
&problemsize_namelist
 n_theta = 96
 n_r = 64
 rmin = 9.0
 rmax = 10.0
/
```

THE SAME

```
&problemsize_namelist
 l_max = 63
 n_r = 64
 shell_depth = 1.0
 aspect_ratio = 0.9
/
```

A NOTE ABOUT NAMELISTS

- Namelists override default values in the code
- Throughout this tutorial, we will be editing many namelist values, while leaving others untouched.
- Only modify indicated values. This means:

You hear/read, "set these : "

```
&problemsize_namelist
 n_theta = 96
 n_r = 64
 rmin = 9.0
 rmax = 10.0
/
```

You see:

```
&problemsize_namelist
 n_theta = 192
 n_r = 32
 rmin = 2.0
 rmax = 10.0
 nprow = 2
 npcol = 4
/
```

You need:

```
&problemsize_namelist
 n_theta = 96
 n_r = 64
 rmin = 9.0
 rmax = 10.0
 nprow = 2
 npcol =4
/
```

i.e., leave nprow and npcol alone in this example.
Omission does not equate to deletion!

```
&problemsize_namelist
 n_theta = 96
 n_r = 64
 rmin = 9.0
 rmax = 10.0
 nprow =8
 npcol =8
/
```

```
&problemsize_namelist
 l_max = 63
 n_r = 64
 shell_depth = 1.0
 aspect_ratio = 0.9
/
```

Exercise 1:

Try both combinations above :

```
$ mpiexec –np 4 ./rayleigh.opt
```

Verify that the preamble is the same.

Exercise 2:

How does the reported iter/sec change if:

- You halve the number of radial points?
- You halve the number of theta points?

GRID POINTS

- Both radial and angular grids are de-aliased
- Angular grid:

$$\ell_{max} + 1 = \frac{2}{3} n_\theta$$

$$n_\emptyset = 2 \times n_\theta$$

- Radial grid:

$$n_{max} + 1 = \frac{2}{3} n_r$$

- Best practices for n_r and n_theta:

  SHOULD have small-prime factorization (e.g., 2 , 3 , 5)

  MUST be even

PROCESS COUNT AND LAYOUT

- Rayleigh's MPI ranks are arranged in process rows and columns
- Total MPI Ranks –> from command line
- NPROW -- number of processes in a row
- NPCOL -- number of processes in a column
- NPROW * NPCOL = Total MPI Ranks– always!
- Control these via main_input

Make changes to main_input:

```
&problemsize_namelist
 nprow = 2
 npcol  = 4
/
```

PROCESS COUNT AND LAYOUT

```
&problemsize_namelist
 nprow = 2
 npcol   = 4
/
```

Try these commands
- what happens?

```
$ mpiexec –np 8 ./rayleigh
```

np MUST equal nprow x npcol

```
$ mpiexec –np 9 ./rayleigh
```

Can also specify nprow and npcol via command-line :

```
$ mpiexec –np 4 ./rayleigh –nprow 2 –npcol 2
```

… overrides main_input values

DETERMINING PROCESS LAYOUT

- NPROW – determines how θ and m are distributed

$$\text{nprow} = \frac{1}{N}\frac{n_\theta}{3}$$

$$N \geq 1$$

- NPCOL – determines how ℓ and r are distributed

$$\text{npcol} = \frac{n_r}{M}$$

$$M \geq 1$$

Note: Due to an unpatched bug, nprow and npcol must each be at least 2!

PROCESS LAYOUT: BEST PRACTICES

For ideal load balancing:

- N should be a factor of $\frac{n_\theta}{3}$
- M should be a factor of n_r

For balanced communication:

- nprow and npcol should agree to within a factor of 2 or 4 (minimizes message count)

RUN TIMING: CALLING IT QUITS

- Specify the number of time steps and/or walltime
- Lowest one “wins”

```
&temporal_controls_namelist
 max_iterations = 5
 max_time_minutes = 30.0
/
```

Try these two combinations:

```
&temporal_controls_namelist
 max_iterations = 5000
 max_time_minutes = 1.0
/
```

Command-line flag for max_iterations:

```
$ mpiexec –np 4 ./rayleigh –niter 2
```

RUN TIMING: TIME-STEP SIZE

- Time-stepping is controlled through the same namelist
- Time-stepping is adaptive

Increase dt iff:
dt < CFL*cflmin

```
&temporal_controls_namelist
 cflmin = 0.4
 cflmax = 0.6
 max_time_step = 1.0d-4
 min_time_step = 1.0d-13
/
```

CFL safety factor

Never take step larger than this

Halt if time step becomes this small

RUN TIMING: EXERCISES

Set the grid parameters to :

```
n_theta = 48
n_r = 64
rmin = 9.0
rmax = 10.0
```

Exercise 1:
Force a time step change
(run for 10 iterations)

```
&temporal_controls_namelist
 max_time_step = 1.0d-2
 min_time_step = 1.0d-13
/
```

Exercise 2:
Force a time-step “crash”
(try running for 10…)

```
&temporal_controls_namelist
 max_time_step = 1.0d-2
 min_time_step = 1.0d-3
/
```

CHECKPOINTING

Rayleigh created a Checkpoints directory – have a look:

```
$ ls Checkpoints
```

- checkpoint_log : list of all checkpoints written so far
- last_checkpoint : last checkpoint written
- numbered directories : dedicated directory for each checkpoint

- Checkpoint format is independent of process configuration
- Checkpoints can be copied between machines of same Endian-ness
- Resolution can be upgraded when starting from a checkpoint
- Main_input can be modified/updated when starting from a checkpoint

CHECKPOINTING

Rayleigh created a Checkpoints directory – have a look:

```
$ ls Checkpoints/00000010
```

Each subdirectory contains everything needed to resume a run:

- grid_etc : simulation grid parameters
- X : 3-D array of variable “X” at timestep 10
- XAB : Nonlinear terms for variable “X” equation
- boundary_conditions : boundary values for all state variables
- equation_coefficients : constant and non-constant PDE coefficients
- main_input : a “copy” of the simulation main_input file

CHECKPOINTING

- Checkpointing controlled via checkpoint_interval
- Sets # of time steps between checkpoints
- Clear Checkpoints before EACH exercise:

```
$ rm Checkpoints/*
```

Exercise 1:

Use values to the right.

Run for 10 time steps.

Check directory contents

```
&temporal_controls_namelist
 checkpoint_interval = 2
 max_time_step = 1.0d-4
 min_time_step = 1.0d-13
/
```

Exercise 2:

Same, but checkpoint every 3rd timestep

Check directory contents… what's odd?

CHECKPOINTING

If a model runs for the specified number of time steps OR the specified walltime, the final time step is saved.

To disable this feature, add the following line to your temporal_controls namelist:

```
&temporal_controls_namelist
 save_last_timestep = .false.
/
```

Exercise:

Clear your Checkpoints directory contents again
Add/set this flag to .false. and rerun previous exercise
Verify that time step 10 was not saved

CHECKPOINTING: QUICKSAVES

- Checkpoints can take up a LOT of space
- But we often want to checkpoint FREQUENTLY
- Solution: rotating checkpoint slots (“quicksaves”)
- Idea: save often, but overwrite most saved data

```
$ rm Checkpoints/*
```

Exercise:
Clear checkpoint directory
Set values to the right
Run the code for 10
time steps.

```
&temporal_controls_namelist
 checkpoint_interval = 5
 quicksave_interval = 2
 num_quicksaves = 3
 max_iterations = 14
 save_last_timestep = .true.
/
```

CHECKPOINTING: QUICKSAVES

- So what happened?

```
$ ls Checkpoints
```

- We have several files of the form quicksave_XX_Y
- Quicksaves were written every other time step.
- The fourth quicksave overwrote quicksave_01
- The fifth quicksave overwrote quickave_02 etc.
- Normal checkpoints take precedence
- So what's in the quicksave slots?

CHECKPOINTING: QUICKSAVES

- All checkpoints are logged:

```
$ more Checkpoints/checkpoint_log
```

How do we read this?

1: Numbered checkpoints are not indented.

```
 00000002 01
 00000004 02
000000005
 00000006 03
 00000008 01
00000010
```

Standard checkpoints (every 5th time step)

CHECKPOINTING: QUICKSAVES

- All checkpoints are logged:

```
$ more Checkpoints/checkpoint_log
```

How do we read this?

2: Quicksaves are indented and numbered

```
 00000002 01
 00000004 02
000000005
 00000006 03
 00000008 01
00000010
 00000012 02
0000000014
```

Quicksave 1 contained:

time step 2 … until…

… time step 8

CHECKPOINTING: QUICKSAVES

- All checkpoints are logged:

```
$ more Checkpoints/checkpoint_log
```

How do we read this?
2: Quicksaves are indented and numbered

```
 00000002 01
 00000004 02
000000005
 00000006 03
 00000008 01
00000010
 00000012 02
0000000014
```

Quicksave 2 contained:

time step 4 … until…
… time step 12

Quicksave 2 was “due” at time step 10.
Numbered checkpointing took precedence.

CHECKPOINTING: QUICKSAVES

- All checkpoints are logged:

```
$ more Checkpoints/checkpoint_log
```

How do we read this?
2: Quicksaves are indented and numbered

```
 00000002 01
 00000004 02
000000005
 00000006 03
 00000008 01
00000010
 00000012 02
0000000014
```

Quicksave 3 contained:
time step 6 … until…

Quicksave 3 was “due” at time step 14.
Final checkpoint took precedence.

CHECKPOINTING: RESTARTS

- Why bother with all of this?
  So we can restart the code as desired!
- To restart, we specify two flags in a new namelist:

Tells Rayleigh to read a checkpoint

Tells Rayleigh which checkpint to use

```
&initial_conditions_namelist
init_type = -1
restart_iter = A NUMBER
/
```

Try these possiblities….

```
restart_iter = 5   :  restarts from 00000005
restart_iter = -2  : negative -> restart from quicksave_02
restart_iter = 0   : restarts from last checkpoint written
                     (could be either type)
```

CHECKPOINTING: BEST PRACTICES

- Checkpoint often with quicksaves
- Conserve disk space by checkpointing sparingly with checkpoint_interval
- How often? Use your judgment, but every 30 minutes is good rule of thumb.

Note: We can use this instead of quicksave_interval:

```
quicksave_minutes = 30.0
```

QUESTIONS?