Meteorology 5344, Fall 2017
Computational Fluid Dynamics
Dr. M. Xue

Computer Problem #l: Optimization Exercises

Due Thursday, September 19
Updated in evening of Sept 6th.

Exercise 1.

This exercise is designed to acquaint you with the basics of the OSCER Schooner Linux Supercomputer (Schooner .oscer.ou.edu) for running programs in single processor and shared-memory parallel (SMP) mode. You will also experience and learn fundamental techniques of code optimization.

For system related problems, including general questions about the compiler, you can contact support@oscer.ou.edu.

Schooner is made up of dual 10-core CPU nodes. Additional details on the system can be found at http://ou.edu/content/oscer/resources/hpc.html

Log onto Schooner (`ssh schooner.oscer.ou.edu -l your_login`), then enter

It is assumed that your default shell is csh or tcsh. If not, enter command after you login:

```
   tcsh
```

Load software modules that you will need to use (you may need to do this everything you log on):

```
   module purge
   module load hdf4/4.2.10/intel
   module load mpi/openmpi/1.10.3/intel
```

(You will need to run the above commands everything you login or enter a new shell environment – such as tcsh – to have access for commands such as Intel Fortran compiler ifort, man pages of ifort, MPI libraries etc.)

Copy files needed from /home/mxue and untar the tar file.

```
   cd
   cp /home/mxue/cfd2017.tar.gz .
   gunzip cfd2017.tar.gz
   tar xvf cfd2017.tar
```

Go into cfd2017 directory and compile Fortran program hw1.f90 using Intel Fortran compiler ifort, with the following sets of options separately:

```
   cd cfd2017
   cd hw1
   ifort -O0 -o hw1_O0.exe hw1.f90 dummy_function.f90
   ifort -O1 -o hw1_O1.exe hw1.f90 dummy_function.f90
   ifort -O2 -o hw1_O2.exe hw1.f90 dummy_function.f90
   ifort -O3 -o hw1_O3.exe hw1.f90 dummy_function.f90
   ifort -O3 -fno-inline-functions -o hw1_O3_noinlining.exe hw1.f90
                                   dummy_function.f90
```

Consult the man pages of ifort (man ifort) for information on the compiler options.

Run the following commands. The program will print out CPU and wall clock times used by various sections of code in the program. The CPU times used should be the total of all cores used when multiple cores are used.

```
   ./hw1_O0.exe > hw1_O0.output
   ./hw1_O1.exe > hw1_O1.output
   ./hw1_O2.exe > hw1_O2.output
   ./hw1_O3.exe > hw1_O3.output
   ./hw1_O3_noinlining.exe > hw1_O3_noinlining.output
```

Recompile hw1.f90 with automatic share-memory parallelization turned on. Run the job using 1, 2, 4, 8 and 16 CPU cores (or number of threads).

```
   ifort -O3 -parallel -o hw1_smp.exe hw1.f90

   setenv OMP_NUM_THREADS 1
   ./hw1_smp.exe > hw1_smp_1thread.output
   setenv OMP_NUM_THREADS 2
   ./hw1_smp.exe > hw1_smp_2thread.output

   setenv OMP_NUM_THREADS 4
   ./hw1_smp.exe > hw1_smp_4thread.output

   setenv OMP_NUM_THREADS 8
   ./hw1_smp.exe > hw1_smp_8thread.output

   setenv OMP_NUM_THREADS 16
   ./hw1_smp.exe > hw1_smp_16thread.output
```

To run the above executable within a batch queue with exclusive access to a node, a batch queue script `test_openMP_hw1_sbatch` is provided in cfd2017 directory. You can **enter the following** to submit the batch job.

```
    sbatch test_openMP_hw1_bsub
```

Enter

```
    squeue -u your_use_id   (i.e., cfdxx)
```

to check the status of your jobs in the batch queue.

For more information on submitting jobs to the batch queue, see http://www.ou.edu/content/oscer/support/running_jobs_schooner.html. sbatch

After the batch job described by script test_openMP_hw1_bsub is finished, examine the CPU and wall clock timings within the output files (files named *.output). Examine carefully the structure and content of the Fortran code sections and the similarities and differences among codes that do the same or very similar things. Discuss the timing results in the context of possible optimizations such as superscalar operations, pipelining, vectorization, (automatic) shared-memory parallelization, memory access pattern, cache utilization, subroutine inlining, and any other observations that you feel important or interesting. The manual pages of ifort compiler will provide a lot of useful information about the compiler optimization and associated options. Enter **`man ifort`** will list the manual pages (if the manual pages fail to find,

Exercise 2.

This exercise is designed to help you gain some hands-on experience running a large atmospheric prediction model, the Advanced Regional Prediction System (ARPS, http://www.caps.ou.edu/ARPS), in single CPU/core, and multi-CPU/core SMP and DMP modes, on a super-scalar DSM parallel system with multi-core/multi-CPU shared-memory nodes, and to help you understand certain optimization and parallelization issues.

The MPI version of ARPS uses the horizontal domain decomposition strategy discussed in class. The shared-memory parallelization relies on Intel compiler's automatic parallelization capability, which performs loop-level parallelization by analyzing the code.

Step 1: Log onto Schooner . Copy ARPS source code package into your home directory, unzip and untar the package.

```
ssh Schooner .oscer.ou.edu -l your_login

cd
cd cfd2017
```

More recent versions of the ARPS package can be downloaded from the ARPS web site at http://www.caps.ou.edu/ARPS. The ARPS model User’s Guide are also available at the website.

Step 2: Compile and build several versions of ARPS executable.

```
cd arps5.2.11
./makearps clean        ! clean off existing object codes and executables if any.
                          'makearps help' lists a set of options for makearps.
./makearps arps         ! builds arps executable using default (usually
                         high)optimization level. Watch and note the
                         compilation to see what compiler options are used.
                         'man ifort tells you what those options mean. The
                         executable is bin/arps.
mv bin/arps bin/arps_highopt   ! rename the arps executable

./makearps clean        ! clean off existing object codes
./makearps -opt 0 arps  ! builds arps executable with minimum optimization
mv bin/arps bin/arps_noopt   ! rename the arps executable
```

```
./makearps clean        ! clean off existing object codes
./makearps -p arps      ! builds arps executable with automatic shared-memory
                          parallelization. Again what the compilation to see
                          what compiler options are used and compare with those
                          used by the first compilation with default compilation
                          level. Note the main difference.
mv bin/arps bin/arps_omp  ! rename the arps executable

./makearps clean        ! clean off existing object codes
./makearps arps_mpi    ! builds the distributed-memory parallel version of ARPS,
                          using MPI. The executable is bin/arps_mpi.
                        ! Please note that if ./makearps clean is run again,
                          bin/arps_mpi will be removed then your MPI jobs will
                          fail. Do ./makearps arps_mpi as the last step.
```

Now all executables you need are built. Do '`ls -l bin`' to be sure.

You can do the above compilations by running shell script `do_arps_compilations`, by entering `./do_arps_compilations`.

Step 3: Go to directory test that contains prepared batch scripts (test_*), arps input (*.input) and sounding data (may20.snd) files for running ARPS at various parallel configurations using either OpenMP or MPI:

```
cd
cd cfd2017
cd test
```

Inside directory test, you will find files named *.input which are the input files contains ARPS configuration parameters. These files are configured to make identical simulations of a supercell thunderstorm for 2 hour, using a 67x67x35 computational grid (set by nx, ny and nz in *.input). For MPI runs, the domain decomposition is specified by parameters nproc_x and nproc_y. For example, in arps_mpi4cpu.input, nproc_x and nproc_y are set to 2, i.e., the computational domain is divided into 2x2 subdomains and distributed over 4 processors. nproc_x=1 and nproc_y=4 or nproc_x=4 and nproc_y=1 should also work although the efficiency may be different because the innermost loops (usually for i index in the x direction) have different length.

Step 4: Examine and submit batch scripts

Various batch scripts are provided in directory test called test*.bsub*, that run different ARPS jobs, using the ARPS executables built earlier; there were compiled with different optimization/ parallelization options, and they will use different number of processors, in single-CPU, shared-memory or distributed-memory MPI mode.

Enter

```
sbatch test_openMP_bsub
```

to submit a batch job that contains the following ARPS jobs using a single node, with single or up to 16 CPU cores/threads in shared-memory-parallel mode, with different levels of optimization.

```
#Run ARPS executable compiled with automatic shared-memory (OpenMP)
parallelization using different number of shared-memory threads

setenv OMP_NUM_THREADS 16
date; ~/cfd2017/arps5.2.11/bin/arps_omp < ~/cfd2017/test/arps_omp16core.input >
~/cfd2017/test/arps_omp16core.output; date

setenv OMP_NUM_THREADS 8
date; ~/cfd2017/arps5.2.11/bin/arps_omp < ~/cfd2017/test/arps_omp8core.input >
~/cfd2017/test/arps_omp8core.output; date

setenv OMP_NUM_THREADS 4
date; ~/cfd2017/arps5.2.11/bin/arps_omp < ~/cfd2017/test/arps_omp4core.input >
~/cfd2017/test/arps_omp4core.output; date

setenv OMP_NUM_THREADS 2
date; ~/cfd2017/arps5.2.11/bin/arps_omp < ~/cfd2017/test/arps_omp2core.input >
~/cfd2017/test/arps_omp2core.output; date

setenv OMP_NUM_THREADS 1
date; ~/cfd2017/arps5.2.11/bin/arps_omp < ~/cfd2017/test/arps_omp1core.input >
~/cfd2017/test/arps_omp1core.output; date
```

Enter

```
sbatch test_nonParallel_bsub

#Run ARPS executable compiled without automatic shared-memory parallelization
and with no compiler optimization

date; ~/cfd2017/arps5.2.11/bin/arps_noopt <
~/cfd2017/test/arps_noopt_1core.input > ~/cfd2017/test/arps_noopt_1core.output;
date

#Run ARPS executable compiled without automatic shared-memory parallelization
but with a high level of single CPU/thread optimizations

date; ~/cfd2017/arps5.2.11/bin/arps_highopt <
~/cfd2017/test/arps_highopt_1core.input > ~/cfd2017/test/arps_highopt_1core.output;
date
```

Enter

```
sbatch test_mpi_bsub_1_1node
```

to submit a batch job that contains that following ARPS job using 1 core on a single node:

```
mpirun -np 1 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps1x1.input >
~/cfd2017/test/arps_mpi1x1.output
```

Enter

```
sbatch test_mpi_bsub_2_1node
```

to submit a batch job that contains that following ARPS jobs using 2 cores on a single node:

```
mpirun -np 2 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps2x1.input >
~/cfd2017/test/arps_mpi2x1.output

mpirun -np 2 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps1x2.input >
~/cfd2017/test/arps_mpi1x2.output
```

Enter

```
sbatch test_mpi_bsub_4_1node
```

to submit a batch job that contains that following ARPS jobs using 4 cores on a single node with different domain decomposition configurations:

```
mpirun -np 4 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps4x1.input >
~/cfd2017/test/arps_mpi4x1.output

mpirun -np 4 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps1x4.input >
~/cfd2017/test/arps_mpi1x4.output

mpirun -np 4 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps2x2.input >
~/cfd2017/test/arps_mpi2x2.output
```

Enter

```
sbatch test_mpi_bsub_8_1node
```

to submit a batch job that contains that following ARPS jobs using 8 cores on a single node with different domain decomposition configurations:

```
mpirun -np 8 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps8x1.input >
~/cfd2017/test/arps_mpi8x1.output

mpirun -np 8 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps1x8.input >
~/cfd2017/test/arps_mpi1x8.output

mpirun -np 8 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps2x4.input >
~/cfd2017/test/arps_mpi2x4.output
```

Enter

```
sbatch test_mpi_bsub_16_1node
```

to submit a batch job that contains that following ARPS jobs using 16 cores on a single node with different domain decomposition configurations:

```
mpirun -np 16 ~/cfd2017/arps5.2.11/bin/arps_mpi <
~/cfd2017/test/arps16x1.input > ~/cfd2017/test/arps_mpi16x1.output

mpirun -np 16 ~/cfd2017/arps5.2.11/bin/arps_mpi <
~/cfd2017/test/arps1x16.input > ~/cfd2017/test/arps_mpi1x16.output
```

```
mpirun -np 16 ~/cfd2017/arps5.2.11/bin/arps_mpi < ~/cfd2017/test/arps4x4.input >
~/cfd2017/test/arps_mpi4x4.output
```

Enter

```
sbatch test_mpi_bsub_16_2nodes
```

to submit a batch job that contains that following ARPS job using 16 cores with 8 cores/node therefore requiring 2 nodes to run the job:

```
mpirun -np 16 ~/cfd2017/arps5.2.11/bin/arps_mpi <
~/cfd2017/test/arps1x16_2node.input > ~/cfd2017/test/arps_mpi1x16_2node.output
```

Enter

```
sbatch test_mpi_bsub_16_4nodes
```

to submit a batch job that contains that following ARPS job using 16 cores with 4 cores/node therefore requiring 4 nodes to run the job:

```
mpirun -np 16 ~/cfd2017/arps5.2.11/bin/arps_mpi <
~/cfd2017/test/arps1x16_4node.input > ~/cfd2017/test/arps_mpi1x16_4node.output
```

Enter

```
sbatch test_mpi_bsub_16_8nodes
```

to submit a batch job that contains that following ARPS job using 16 cores with 2 cores/node therefore requiring 8 nodes to run the job:

```
mpirun -np 16 ~/cfd2017/arps5.2.11/bin/arps_mpi <
~/cfd2017/test/arps1x16_8node.input > ~/cfd2017/test/arps_mpi1x16_8node.output
```

Enter

```
sbatch test_mpi_bsub_16_16nodes
```

to submit a batch job that contains that following ARPS job using 16 cores with 1 core/node therefore requiring 16 nodes to run the job:

```
mpirun -np 16 ~/cfd2017/arps5.2.11/bin/arps_mpi <
~/cfd2017/test/arps1x16_16node.input > ~/cfd2017/test/arps_mpi1x16_16node.output
```

For the above ARPS batch job submissions, you can write a script to submit all jobs at once. You can achieve this within test directory by entering `./submit_batchfiles.` Then enter `squeue –u your_id` to check the status of your batch jobs.

Step 5: Examine timing statistics in the output file and discuss the results

After the batch jobs complete, look into the output file created by each run (called *.output – see an example given below). At the end of the file, there are timing statistics like the following. Put the total CPU time and wall clock time for all runs into a table and discuss and try to explain the timing results.

```
grep "Entire model" *.output
```

to get a listing of the total CPU and Wall clock times used by each job after you have completed all the runs (when all *output files are generated).

Specifically, do the following:

Compare the single core jobs (arps_noopt_1core, arps_highopt_1core, arps_omp1core, arps_mpi1x1), discuss the impacts of optimization, etc and explain the behaviors based on your knowledge about the compiler optimization and hardware architecture.

Compare the timing statistics of shared memory parallel (OpenMP/omp) jobs using different number of cores/threads (arps_omp*core), and discuss the parallelization efficiency/speedup factors.

Compare the timing statistics of distributed memory parallel (MPI) jobs using different number of cores (up to 16) on a single node, and with different domain decomposition configurations and discuss the parallelization efficiency/speedup factors.

Compare the timing statistics of MPI jobs using 16 cores but different number of nodes, and discuss the parallelization efficiency/speedup factors, and possible reasons for your findings.

Compare the spend up factors of OpenMP and MPI jobs with up to 16 cores on a single node.

Provide some general discussion/recommendation for running similar ARPS jobs.

Use graphs (e.g., histograms) to show the timings. Pay attention to the CPU, core and level-2 cache configurations of the Schooner compute nodes when discussing your results.

'ls –l runname.*' (where runname is arps1x1 etc.) will show output files runname.hdf000000, runname.hdf003600 and runname.hdf007200 (7200 here is 7200 s or 2 hours). The interval between the creation times of the 2 and 0 hour files (output at initial and end times of model run) is pretty much the wall clock time used by the job and should be close to that at the end of runname.output (e.g., arps_highopt_1cpu.input) file.

```
Example of Time Statistics Printed at the end of ARPS output file (they were
obtained on a different computer so your numbers will be different):

ARPS CPU Summary:

 Process             CPU time             WALL CLOCK time (Processor mean for
MPI job)
 ----------------   ---------------------     ---------------------
 Initialization  :   1.551763E+00s    0.06%     2.680500E-01s    0.18%
```

```
Data output     :   6.702171E+01s    2.78%     4.281781E+00s    2.83%
Wind advection  :   3.289250E+01s    1.37%     2.087100E+00s    1.38%
Scalar advection:   1.176103E+02s    4.88%     7.433331E+00s    4.92%
Coriolis force  :   0.000000E+00s    0.00%     0.000000E+00s    0.00%
Buoyancy term   :   1.957366E+01s    0.81%     1.227869E+00s    0.81%
Misc Large tstep:   4.031898E+01s    1.67%     2.562375E+00s    1.70%
Small time steps:   6.853007E+02s   28.45%     4.326842E+01s   28.63%
Radiation       :   2.799463E-02s    0.00%     8.687500E-04s    0.00%
Soil model      :   0.000000E+00s    0.00%     0.000000E+00s    0.00%
Surface physics :   0.000000E+00s    0.00%     0.000000E+00s    0.00%
Turbulence      :   3.649897E+02s   15.15%     2.287119E+01s   15.14%
Comput. mixing  :   1.111839E+02s    4.62%     6.964956E+00s    4.61%
Rayleigh damping:   1.448061E+01s    0.60%     9.221437E-01s    0.61%
TKE src terms   :   9.704794E+01s    4.03%     6.079481E+00s    4.02%
Gridscale precp.:   6.999552E-03s    0.00%     5.500000E-04s    0.00%
Cumulus (NO    ):   0.000000E+00s    0.00%     0.000000E+00s    0.00%
Microph (warmra):   1.433780E+02s    5.95%     8.984076E+00s    5.95%
Hydrometero fall:   7.225069E+01s    3.00%     4.520113E+00s    2.99%
Bound.conditions:   2.420235E+01s    1.00%     1.474463E+00s    0.98%
Message passing :   5.158498E+02s   21.41%     3.184401E+01s   21.07%
Miscellaneous   :   1.011852E+02s    4.20%     6.317975E+00s    4.18%


Entire model    : 2.408868E+03s              1.511087E+02s

Without Init/IO : 2.340294E+03s              1.465589E+02s
```

The total CPU time used by the entire model (2.408868E+03s above) is the sum of CPU time used by all processors.

WALL CLOCK time used by the entire model (1.511087E+02s above) is the wall clock time from the start to end the model excitation. This is the number that you should focus your discuss on.