Meteorology 5344, Fall 2003
Computational Fluid Dynamics
Dr. M. Xue

Computer Problem #l: Optimization Exercises on the IBM Regatta (sooner.oscer.ou.edu)

Distributed on Friday, August 29, 2003
Due on Friday, September 12, 2003

Exercise 1.

This problem set is designed to acquaint you with the basics of the OSCER IBM shared-memory parallel supercomputer (sooner.oscer.ou.edu), as well as fundamental techniques of code optimization. The IBM machine consists of 32 Power4 superscalar processors.

On Sooner, copy program hw1.f90 from /home/mxue/cfd2003. Compile and run the program using the following three sets of options:

```
xlf90 -qsuffix=f=f90 -o hw1.exe –O1 hw1.f90
xlf90 -qsuffix=f=f90 -o hw1.exe -O3 hw1.f90
xlf90 -qsuffix=f=f90 -o hw1.exe –qipa=inline –O5 hw1.f90
```

You can consult the man pages of xlf90 for information on the compiler options.

Run hw1.exe. The program will print out CPU times used by various sections of code in the program.

Make a table of the CPU times used by all the sections. You may want to run several times and take the average. Examine carefully the structure and content of the sections of code and the similarities and differences among codes that do the same things. Discuss the timing results in the context of general optimization, memory access pattern, pipelining, effects of subroutine calls inside loop and compiler optimization levels and options, and any other observations that you feel important or interesting.

Exercise 2.

This exercise is designed to help you gain some handson experiences running the Advanced Regional Prediction System (ARPS, http://www.caps.ou.edu/ARPS), a sophisticated mesoscale weather prediction model, in various modes, on a modern super-scalar shared-memory parallel platform, the IBM Regatta p690 (sooner.oscer.ou.edu), and to help you understand certain optimization and parallelization issues.

You may find sections of “The Power4 Processor Introduction and Tuning Guide” as well as Dr. Henry Neeman’s Seminar presentation linked at our class’s web site helpful for you to understand the timing statistics. The MPI version of ARPS uses the horizontal domain decomposition strategy discussed in class. The shared-memory parallelization relies on IBM compiler's automatic parallelization capability, which determines if loops can be parallelized by analyzing the code.

Step 1: Log onto sooner.oscer.ou.edu using ssh. Copy ARPS source code package into your home directory, unzip and untar the package.

```
ssh sooner.ou.edu -l cfd##  (where cfd## is your account)
cd $HOME
cp /home/mxue/cfd2003/arps5.0.0IHOP_6.tar.gz .
gunzip arps5.0.0IHOP_6.tar.gz
tar xvf arps5.0.0IHOP_6.tar
cd arps5.0.0IHOP_6
```

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

```
cd arps5.0.0IHOP_6
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 xlf95' 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_smp  ! 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.
```

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

Step 3: Copy a directory containing example batch scripts (*.cmd), arps input (*.input) and sounding data (may20.snd) files, into your ARPS directory:

```
cp –rp /home/mxue/cfd2003/test .
cd test
```

Do the above inside your ARPS directory. 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 1 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. If you are familiar with ARPS plotting program, arpsplt, you can plot and visualize the output data (in file called *.hdf003600 for the end time, see RAEDME file in the ARPS root directory for simple instructions. http://www.caps.ou.edu/ARPS/arpsqg/ has a quick guide to various programs in the ARPS package).

The *.cmd files are the example LoadLeveler batch scripts that you need to edit to change the paths of input, output files and executables.

Step 4: Prepare and submit batch scripts

The eight example batch scripts listed below are provided for you to run eight ARPS timing tests. arps_noparallel*.cmd use arps executables built without parallelization, arps_smp*.cmd use the shared-memory auto-parallel version (arps_smp), and arps_mpi*.cmd run distributed-memory version of ARPS (arps_mpi). Comments in the *smp*.cmd scripts explain the meanings of the batch queue parameters. We will call strings such as arps_noparallel_highopt as runname, which is specified inside input files such as arps_noparallel_highopt .input and used to construct the names of output files by ARPS.

```
arps_noparallel_highopt.cmd
arps_noparallel_noopt.cmd
arps_smp1cpu.cmd
arps_smp2cpu.cmd
arps_smp4cpu.cmd
arps_mpi1cpu.cmd
arps_mpi2cpu.cmd
arps_mpi4cpu.cmd
```

You need to edit the above batch scripts to change the paths to file names and executables. Absolute path has to be used. Compare the batch scripts to see how parameters such as the number of CPU and/or threats are set.

To submit a batch job to the LoadLeveler batch system, using the first script as an example,

```
llsubmit arps_noparallel_highopt.cmd
```

To check the LoadLeveler batch queues:

```
     llq -x
```

To cancel a LoadLeveler batch job:

```
    llcancel <JobID>
```

Note that <JobID> is an integer that appears as part of the leftmost column of the output of llq.

To get detailed information on the status of a job:

```
   llq -s <JobID>
```

Not that there are often many long-running jobs in the queue, so you may have to wait a while for your jobs to run. The jobs should take between a few minutes to less than 2 hours wall clock time to finish.

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

Look into the output file created by each run (called *.output). 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 the timing results. Compare the shared and distributed-memory parallel performance (speed up) relative to each other and relative to the single processor run. Discuss the impact of optimization, etc. Use graphs when they are helpful. 'ls –l runname.*' (with runname substituted by the actual run name) will show output files runname.hdf000000 and runname.hdf003600. The interval between the creation times of these two 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 file.

```
Example of Time Statistics Printed at the end of ARPS output file:

ARPS CPU Summary:

 Process              CPU time                    WALL CLOCK time
 -----------------    ----------------------      ----------------------
 Initialization  :    5.900000E-01s    0.02%      3.900000E-01s    0.03%
 ...
 Message passing :    1.820433E+00s    0.07%      1.590000E+00s    0.11%

 Miscellaneous   :    5.122986E+01s    2.09%      3.066000E+01s    2.15%


 Entire model    : 2.456460E+03s              1.426440E+03s

 Without Init/IO : 2.425460E+03s              1.407960E+03s
```

The total CPU time is the sum of CPU time used by all processors. Please note that the timing results can be affected by the load of machine at the time your job was run. I have seen jobs requesting more processors taking longer (wall clock time) to complete than single processor job. If the timing appears obviously wrong, you may want to rerun the job. Fell free to note and discuss any abnormal behaviors.