PCSE OpenMP Homework 1, part 2 (hw1_part2)

(updated 3/7/2017 – see green text, pg 4 – Fortranner use f90_setaffinity)

Overview

This is part2 of 2 parts for hw1. You will parallelize an existing the rb.c or rb.F90 serial program using OMP directives and report scaling information.

Work with either C or Fortran codes—NOT both. DO THIS ASSIGNMENT on Stampede.

Resources:

Untar the file omp_hw1_part2.tar into a directory with the following commands:

```
cd
tar    xvf   ~train00/omp_hw1_part2.tar
cd     hw1/rb
```

General:

You can use the login node testing and parallelizing codes, but use ***idev*** to interactively run an executable when benchmarking. You can check where the threads are running by checking the load on each core with the ***top*** utility.

```
top      # then hit the 1 character
```

The basic steps are.

1. Look over the serial code.
2. Copy the serial code to another file (with a specific name) and parallelize it using OpenMP directives. Compile the code
3. Run the code
4. **Plot and analyze results, as requested.**

To make this a bit easier, scripts have been made for you.
For compiling execute:

./compile my_prog.c or **./compile my_prog.F90**

The ***compile*** script will make an executable named ***a.out_my_prog***.

To interactively run an executable with various number of threads, execute:

./dothis # (will run parallel code for different # of threads)

FYI: The timer.c and affinity.c codes are compiled with the "-c" option in the ***compile*** script and creates just "object" files (timer.o, affinity.o). These are "loaded" into your code on the compile line like this:

```
ifort -qopenmp timer.o affinity.o prb_d.F90 -o a.out_prb_d
icc   -qopenmp timer.o affinity.o prb_d.c   -o a.out_prb_d
```

(You can always type these on the command line instead of using ***compile***.)

Instructions hw1 – part 2

1. The red-black (rb) program is an example of an update mechanism that might be encountered in various algorithms. In a red-black algorithm, red or black elements of an array are updated separately. Two red-black loops update array "a" with other elements of "a". In this red-black algorithm the "red" elements of the "a" array, {1,3,5,7,...}, are updated, and then the "black" elements of the array, {0,2,4,...}, are updated. (Here, we could have simply called them odd and even, respectively. (But, of course, in 2-D or 3-D algorithms, associating color with a pattern makes much more sense for visualization.)

   The objective here is to first parallelize loops in a simple manner (part a), and then reduce the number of parallel forks in part b. In part c you will also try to optimized the code and check the performance of binding (affinity) schemes.

   You will create 3 different parallel codes from the same serial code source, parallelize them in slightly different ways, and then benchmark and analyze them. In the Report show the OpenMP changes to the pseudo code (below), and include any loop changes in the optimization section (part c). Create required plots, and discuss the behavior and/or parallelizability where specified. Report times for the while loop only, as shown in the serial code, rb.c.

```
Begin Program  Pseudo Code
<any new variables here>

#pragma omp parallel … or !$omp parallel
 nt = omp_get_num_threads(); WARMUP,initial parallel


#pragma omp ... or !$omp ...
  Initialization:

  Start timer
  while[

     #pragma omp ...  or !$omp ...
     1st red-black loop (if you change

     #pragma omp ...  or !$omp ...
     2st red-black loop

     #pragma omp ...   or !$omp ...
     error var initialization

     #pragma omp ...  or !$omp ...
     error loop

  ]
  End timer

  Report time

END
```

Use the ***schedule(runtime)*** clause on the work sharing loops in the while loop so that they can run with different scheduling algorithms (static and dynamic).

Use the ***compile*** and ***dothis*** scripts for compiling code and executing them. Execute the ***dothis*** script interactively on a compute node – use idev.

---

a. Parallelize the code in the most obvious/simple (and correct) way—by separately parallelizing each loop (with **parallel do** or **parallel for**) within the timed region. General Hint: be careful with initializing the error var).

    i. Copy rb.c/F90 to prb_a.c or prb_a.F90
    ii. Parallelize all loops in with "parallel for" or "parallel do", EXCEPT for the initialization.
    iii. Compile: **./compile prb_a.c** (creates a.out_prb_a)
    iv. Execute **./dothis** to execute a.out_prb_a (various # of threads and schedules).Times for 1-8 threads are reported for 4 types of scheduling. Report: Do the various Scheduling methods make much difference? How good/bad is the scaling up to 8 threads?
    v. Change code to now parallelize the initialization loop. DO NOT include a schedule clause—let it default to static scheduling. Compile with ***compile prb_a.c*** and execute ***dothis***. Report: How much better the scaling is? Why is it better? Do the various Scheduling methods now make much difference?
    vi. Watch where the threads are executing with top. (On the node in another window execute "top" and then hit the "1" key to see the loads. Type s and then 1 to have top update every second.) Report: Is there any consistency to where the threads are running in iv & v?

    Report: Show coding changes. Plot the scaling for the STATIC case (without a chunk size) with and without the parallelization of the initialization loop (label as ser_init and par_init in plot).

b. Using prc_a.c, reduce the number of parallel regions (directives).

    i. copy prb_a.c/F90 to prb_b.c or prb_b.F90
    ii. Modify directives in the while loop so that there is only one parallel directive in the *while loop*. (Hint: Think "single" for the error and niter vars.)
    iii. Compile: use ***compile prb_b.c***; and execute: ***dothis.***

    Report: Show coding changes. Does the scheduling make any difference? Does the code scale better or worse than the runs in part a; does it run faster? Give a reason why they "should" be about the same, even though the number of forking requests has been significantly reduced.

c. There are 2 optimizations to be performed here: 1.) Use a single parallel region (directive) outside the while loop; 2.) If the 2 red-black loops can be combined, fuse them. We use16 threads here.

General Instructions: Copy rb.c/F90 to prb_c.c or prb_c.F90 [or copy from prb_b.c/F90], parallelize, compile with ***compile prb_c.c/F90***; use ***dothis*** to execute and run on a compute node.

i. Once you have constructed a single parallel directive outside of the while loop, compile and run multiple times with ***dothis***, save output. (Include the initialization of the **a** array in the parallel region.)
ii. Now combine red-black loops, compile and run multiple times with ***dothis*** (save output).

`Review the contents of` ***`dothis`*** `and` ***`compile`*** `scripts.`