PCSE OpenMP Homework 1, part 1 (hw1_part1)

Overview

This is part1 of 2 parts for hw1. You will parallelize the code that you have written in homework 0 using OMP directives and report scaling information.

DO THIS ASSIGNMENT on:
Stampede (preferred) or on Maverick (only if Stampede2 is unavailable.

Resources:

Your own code written for homework 0

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. Create a new folder named hw1
2. Create a subfolder part1
3. Copy your serial code from your hw0 folder to the hw1/part1 folder and parallelize your code using OpenMP directives.
4. Debug, compile, and run the code
5. **Plot and analyze results, as requested.**

Machine selection
This exercise works best on Stampede2. Only use maverick if Stampede becomes unavailable

Instructions on compiling
Compile your code with the flags **-O2 -xHost –qopenmp** (-**openmp** on Maverick, see below)

Instructions: hw1 – part 1

1. Your code has 3 time consuming kernels
    a. Initialization if array x with random numbers
    b. Smoothing, i.e. calculating array **y** from array **x**
    c. Counting of elements below a threshold
2. At the beginning of your code (i.e. before you initialize the first array with random numbers) add a dummy parallel region that only prints a message from each thread. Having this 'dummy' parallel region will prevent your actual timings from being corrupted by extra overhead from starting the very first parallel region.
3. Pseudo code would read like this (please fill in the blanks for your favorite language):

```
a. omp parallel
b. print 'This is thread', omp_get_thread_num()
c. end parallel
```

4. Parallelize the kernels b. and c. Do **not** parallelize the first kernel that initializes one array with random numbers
5. Add 2 distinct parallel regions to your code, i.e. one for each kernel under consideration
6. Employ the appropriate work-sharing constructs for the 2 kernels
7. Enhance the output of your code so that the number of threads is printed in the well-organized output created at the end of the code.
8. Once you have finished modifying and testing the code increase the number of elements in both directions to 32768+2. This will increase the run-time but also the memory footprint. It is absolutely essential that you are testing and running this larger case in an idev session, if you have not already done so in the first place. You will be using a lot of resources on a node. If you run the tests on the login nodes you will be caught and blocked by the sys-admin. Moreover your timings will be wrong since you will be sharing the login node with other users.
9. Throughout this assignment (unless otherwise instructed) execute your code with
    numactl --cpunodebind=0 --preferred=0 ./a.out (insert your executable name here)

Assignment 1: This assignment focuses on the kernel that counts the number of elements in an array.

- Make sure that your code is properly debugged. Compare the number of elements below the threshold in your serial and parallel version.
- Report 3 times:
    - Serial execution of the parallel code: compile without the -qopenmp flag
    - Parallel execution with 1 thread: compile with the -qopenmp flag, set the number of threads to 1 (OMP_NUM_THREADS variable)
    - Parallel execution with 8 threads
    - Discuss the difference in execution time between the serial execution and the parallel execution with one thread. The results are surprising. Speculate what may be going on.

Assignment 2a: This assignment focuses on the 'smoothing' kernel. Report only times for this kernel in assignment 2. Only discuss the timings for the 'smoothing' kernel in this assignment.

- Don't forget to use the numactl command.
- Run tests with static scheduling without a specific chunk size. Create a plot (speedup v. thread count) for 1, 2, 4, 8, 16, 32 and 64 threads. Add also the ideal speed-up in either plot.

- Discuss the plots. What setup performs the best? Discuss the deviation from the ideal speed-up.
- Add the output of your code for the run with 8 threads to your report.
- In your opinion: Is the scaling good or bad?

Assignment 2b: Again, only the 'smoothing' kernel is of concern here.

- Create a table with run-times for a number of different scheduling settings. Use 8 threads for these tests. Include static and dynamic scheduling in your table and chunk sizes of 100, 1000, and 10000. So in total you should have
  2 x 3 entries, i.e. 2 schedules and 3 chunk-sizes.
- Discuss the timing differences between the scheduling types (static v. dynamic)
- Discuss the timing differences between the different chunk sizes.

Assignment 2c:

- Again use 8 threads. Create a table with run-times for the 'smoothing' kernel. Experiment with different **numactl** settings
  i. ./a.out (no numactl)
  ii. numactl --cpunodebind=0 --preferred=0 ./a.out
  iii. numactl --cpunodebind=0 --interleave=0,1 ./a.out
  iv. numactl --cpunodebind=0 --preferred=1 ./a.out
- Discuss the results.
- Speculate why the execution speed differs. Base your speculation of the fact that a thread that executes on a specific socket can access data faster that is allocated on the same socket than data that is allocated on the other socket. In your table add 2 columns indicating where the memory is allocated and where the threads are running in the 4 experiments.
- Look at the man page for numactl. Execute the command
  numactl --hardware
- Describe what does the option **--hardware** do? Refer to the man page.
- How many nodes are being shown by the command?
- Add to your report the total amount of memory for the available nodes. Results in MB or GB are accepted.

Assignment 3:

- Inspect the man page of your compiler on Stampede2 (either icc or ifort). Explain briefly what the option **-xHost** instructs the compiler to do. Speculate why it is important to use the 'highest instruction set available'.

Summary:

- Parallelize 2 kernels in your code
- Investigate the 'counting' kernel (assignment 1)
- Investigate the 'smoothing' kernel (assignment 2)
- Answer all the questions from the assignments. All text highlighted in green warrants a response in your report.
- Answer the **-xHost** question
- Do not forget to add the compiler options **–xHost, -O2, and –qopenmp**
- Special remark: On Maverick the OpenMP compiler option is -openmp