CSE 491-Section 2
Parallel Processing

Homework 2

Due in class, October 14, 2015

Important Notes:

- *For your questions about the homework, please use the D2L discussion forum for the class so that answers by the TA, myself (or one of your classmates) can be seen by others.*
- *Assuming you have a compiler with OpenMP support (recent versions of GCC has), you may develop your programs locally on your computer, but the performance data for problems 2 and 3 must be collected on HPCC's intel14 and intel14-phi clusters.*
- *Note that jobs may wait in the queue to be executed for a few hours on a busy day, thus plan accordingly and do not wait for the last day of the assignment.*
- *In problems 2 and 3, make sure that you take at least 3-5 measurements for running times. Pick the minimum running time among those (which is the best case scenario).*

Problem 1 (20 points)

a) Suppose the run-time of a serial program is given by $T_{serial} = n^2$, where the units of the run-time are in microseconds. Suppose that a parallelization of this program has run-time $T_{parallel} = n^2/p + log_2(p)$. Write a program that finds the speedups and efficiencies of this program for various values of n and p. Run your program with n = 10, 20, 40, …, 400, and p = 1, 2, 4, 8, …, 128.
- What happens to the speedups and efficiencies as p is increased and n is held fixed?
- What happens when p is fixed and n is increased?
Plot speedups and efficiencies to help you understand the scaling behavior of the problem for different values of n & p. Explain your answers with some example plots (*don't include all possible plots*).

b) Suppose that $T_{parallel} = (T_{serial}/p) + T_{overhead}$ and also suppose that we fix p and increase the problem size.
- Show that if $T_{overhead}$ grows more slowly than T_{serial}, the parallel efficiency will increase as we increase the problem size.
- Show that if, on the other hand, $T_{overhead}$ grows faster than T_{serial}, the parallel efficiency will decrease as we increase the problem size.

Problem 2 (30 points)

Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and its area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation:

number in circle / total number of tosses = π / 4,

since the ratio of the area of the circle to the area of the square is π / 4. We can use this formula to estimate the value of π with a random number generator. This is called a "Monte Carlo" method, since it uses randomness (the dart tosses).

```
number_in_circle = 0;
for (toss = 0; toss < number of tosses; toss++) {
    x = random* double between -1 and 1;
    y = random* double between -1 and 1;
    distance_squared = x * x + y * y;
    if (distance_squared <= 1) number_in_circle++;
}
pi estimate = 4 * number_in_circle/((double) number_of_tosses);
```

a) Using the skeleton code provided (pi_skltn.c), first implement a sequential code to estimate π. Then create two OpenMP parallel versions of your code using atomics and reduction to estimate the total number of darts hitting inside the circle.

b) Plot the speedup and efficiency curves for your OpenMP parallel versions (e.g. using 1, 2, 4, 8, 10, 16 and 20 threads) on the **intel14** cluster for different problem sizes, i.e. number of tosses = 100,000, 10 million, and 1 billion. How do the two different OpenMP versions compare against each other? How do speedup and efficiency change with problem size? Why?
Important: Note that C's *rand*() function is not thread safe! Use the simpler thread-reentrant verion, `int rand_r(unsigned int*)` in your programs. For good randomness among threads, you must make sure that each thread starts with a different seed! For example, thread ids…
int rand_r(unsigned int*): http://linux.die.net/man/3/rand_r

Problem 3 (50 points)
Count sort is a simple serial sorting algorithm that can be implemented as follows:

```
void count_sort(int a[], int n) {
  int i, j, count;
  int *temp = malloc(n*sizeof(int));

  for (i = 0; i < n; i++) {
    count = 0;
    for (j = 0; j < n; j++)
      if (a[j] < a[i])
        count++;
      else if (a[j] == a[i] && j < i)
        count++;
    temp[count] = a[i];
  }
  memcpy(a, temp, n*sizeof(int));
  free(temp);
} /* Count sort */
```

The basic idea is that for each element `a[i]`, we count the number of elements in `a` that are less than `a[i]`. Then we insert `a[i]` into the `temp` list using the subscript determined by `count`. There's a slight problem with this approach when the list contains equal elements, since they could get assigned to the same slot in `temp`. The code deals with this by incrementing `count` for equal elements on the basis of the subscripts. If both `a[i] == a[j]` and `j < i`, then we count `a[j]` as being "less than" `a[i]`. After the algorithm has completed, we overwrite the original array with the temporary array using the string library function `memcpy`. A serial implementation is provided on D2L (count_sort_skltn.c) that should be used for validation and speedup.

a) Implement two different OpenMP parallel versions of the count sort algorithm: first by parallelizing the `i`-loop, and second one by parallelizing the `j`-loop. *In both versions, modify the code so that the* `memcpy` *part can also be parallelized.*

b) Compare the running time of your OpenMP implementations on multiple threads (e.g. 1, 2, 4, 8, 10, 16 and 20 threads) on a single **intel14** node at HPCC. Plot total running time vs. number threads (include curves for both the i-loop and j-loop versions in a single plot) for a fixed value of *n*. **A good choice for *n* would be in the range 50,000 to 100,000.**

c) Now do the same comparison and plots in B on a Xeon Phi card on the **intel14-phi** cluster. Use the *offload* pragma for running the sorting part on the Xeon Phi (see instructions at the bottom). Run your program with 10, 20, 40, 80, 120, 160, 200 and 240 threads.

d) How does your best performing implementation (`i`-loop vs. `j`-loop) on **intel14** or **intel14-phi** compare to the serial `qsort` library function's performance in C? Why? Explain your observations.

Compiling and running your programs on Xeon Phi: Xeon Phis at HPCC can be used in *offload* mode only. For compiling and running your programs, you need to use the Intel programming environment on HPCC:

```
- module swap GNU Intel
- icc -O3 -openmp –o count_sort_iloop.mic  count_sort_iloop_mic.c
```

Note that "module swap GNU Intel" is required not only for compiling your codes, but also for running your executables. For example, you created your executable and want to test it. In your job script or when you start an interactive session, you still need to do a "module swap GNU Intel" so that required libraries get loaded.

Further help regarding how to program with Xeon Phis can be found on pages 17-18 of the following document:

https://software.intel.com/sites/default/files/managed/ee/4e/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf

Measuring your execution time properly: The omp_get_wtime() command will allow you to measure the timing for a particular part of your program (see the skeleton codes). However, on the dev-nodes there will be several other programs running simultaneously, and your measurements will not be accurate. After you make sure that your program is bug-free and executes correctly on the dev-nodes, the way to get good performance data for different programs and various input sizes is to i) use a job script for your runs, or ii) use the interactive queue. Suggested options for starting an interactive queue on **intel14** and **intel14-phi** are as follows:

```
qsub -I -l nodes=1:ppn=20,walltime=00:30:00,feature=intel14 -N myjob
qsub -I -l nodes=1:ppn=20,walltime=00:30:00,feature=phi -N myjob
```

Sometimes getting access to a node interactively may take very long. In that case, we recommend you to create a job script with the above options, and submit it to the queue (this may still take a couple hours, but at least you do not have to sit in front of the computer). Note that you can execute several runs of your programs with different input values in the same job script – this way you can avoid submitting and tracking several jobs.

The options above will allow exclusive access to a node for 30 minutes. If you ask for a long job, your job may get delayed. Note that default memory per job is 750 MBs, which should be plenty for the problems in this assignment. But if you will need more memory, you need to specify it in the job script.

Turn in instructions: You will return a hardcopy homework, including report and interpretation of your results for the programming assignments. You will turn in your source codes and output files by emailing them to the TA.