CSE/CMSE 822: Parallel Computing
Fall 2017, Homework 4
Due 11:59 pm, October 18, 2017

*This homework is 15% of your **homework grade**.*

***Important note:** Please use a word processing software (e.g., MS Word, Mac Pages, Latex, etc.) to type your homework. Follow the submission instructions at the end to turn in an electronic copy of your work.*

1) [50 pts] Estimation of the value of π

One way to estimate the value of π is the "Monte Carlo" method which uses randomness to generate an approximation. In this method, 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. Assuming we always hit the square, and the points that are hit by the darts are uniformly distributed (which is the default behavior of any random number generator), then the number of darts that hit inside the circle should approximately satisfy the equation:

$$\frac{\#darts\ in\ the\ circle}{total\ \#tosses} = \frac{\pi}{4}$$

because the ratio of the area of the circle to the area of the square is π / 4.

This method is outlined in the code segment below.

```
number_in_circle = 0;
for (toss = 0; toss < number of tosses; toss++) {
    x = a random number (of type double) between -1 and 1;
    y = a random number (of type 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);
```

i) [25 pts *for correctness*] 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. Your first OpenMP parallel version must use atomic updates (at each iteration), and the second version must use reduction to resolve the race conditions among threads in estimating the total number of darts hitting inside the circle.

ii) [25 pts *for performance and interpretation*] On the **intel16** cluster, plot the speedup and efficiency curves for your OpenMP parallel versions using 1, 2, 4, 8, 14, 20 and 28 threads for three different problem sizes, a) 100,000, b) 10 million, and c) 1 billion tosses. Analyze how do the two different OpenMP versions compare against each other with respect to problem size, i.e. how do their relative speedup and efficiencies compare? Explain your observations.

Important: C's *rand*() function is not thread safe! Therefore, in your programs, use the simpler thread-reentrant version:

```
int rand_r(unsigned int*)
```

Since this function is memoryless, the sequence of random numbers it generates is not as high quality as the rand() function. To ensure good randomness among threads, make sure that each thread starts with a different seed! For example, the thread id of each thread is a good seed to start

with. You can read more about rand_r() function through its man page, or on the following website http://linux.die.net/man/3/rand_r

2) [60 pts] Parallel Count Sort

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.

i) [30 pts] Implement two different OpenMP parallel versions of the count sort algorithm. First version should parallelizing `i`-loop, and second version should parallelize the `j`-loop. *In both versions, modify the code so that the `memcpy` part can also be parallelized.*

ii) [20 pts] On the **intel16** cluster, compare the running time of your OpenMP implementations using 1, 2, 4, 8, 14, 20 and 28 threads. 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.*

iii) [10 pts] How does your best performing implementation (`i`-loop vs. `j`-loop) compare to the serial `qsort` library function's performance in C? Why? Explain your observations.

Instructions:

- **Accessing the git repo.** You can pull the skeleton codes from the git repo. Please refer to *"Homework Instructions"* under the *"Reference Material"* section on D2L.

- **Submission.** Your submission will include:
  - A pdf file named "HW4*_yourMSUNetID*.pdf", which contains your answers to the non-implementation questions, and report and interpretation of performance results. *If your assignment is written using a document processing software such as word, please export the document to PDF format for your homework submission: do not submit the *.doc file.*
  - All source files used to generate the reported performance results. Make sure to use the exact files names listed below:
    - sequential.c (for problem 1)
    - atomic.c (for problem 1)
    - reduction.c (for problem 1)
    - countsort_paralleli.c (for problem 2)
    - countsort_parallelj.c (for problem 2)

    These are the default names in the git repository – essentially, do not change the directory structure, file names, or form of the function declarations.

  To submit your work, please follow the directions given in the "Homework Instructions" under the "Reference Material" section on D2L. Make sure to strictly follow these instructions; otherwise you may not receive proper credit.

- **Discussions.** For your questions about the homework, please use the Slack group for the class so that answers by the TA, myself (or one of your classmates) can be seen by others.

- **Compilation and execution.** You can use any compiler with OpenMP support. The default compiler environment at HPCC is GNU, and you need to use the –fopenmp flag to compile OpenMP programs properly. Remember to set the OMP_NUM_THREADS environment variable, when necessary.

- **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). *Make sure to collect at least 3 measurements and take their averages while reporting a performance data point.*

- **Executing your jobs.** 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 use the interactive or batch execution modes. *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.*

  i) **Interactive queue.** Suggested options for starting an interactive queue on the **intel16** cluster is as follows:

  ```
  qsub -I -l nodes=1:ppn=28,walltime=00:30:00,feature=intel16 -N myjob
  ```

  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.

ii) **Batch job script.** 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. An example job script is provided in the file *job_script_pi.qsub*.