CSE 491 – Section 2: High Performance Computing
Spring 2017, Homework 3

Due 11:59 pm, Friday, March 24th

This homework is 20% of your ***homeworkgrade*** *(not your total 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) [20 pts] Loop parallelization and data dependencies

Suppose A is a matrix of dimensions K by M, which is updated as follows:

```
for (int i = 1; i <K-1; i++)
      for (int j = 0; j <M; j++) {
            int a1 = A[i-1][j];
            int a2 = A[i+1][j];
            A[i][j] = (a1+a2)/2;
      }
```

i) [7 pts] Determine the data dependencies in these nested for-loops.
ii) [7 pts] Draw the iteration space dependency graph for this code.
iii) [6 pts] Is this code OpenMP parallelizable? If so, give its OpenMP parallel version. If not, explain why?

2) [40 pts] Estimation of the value of π

One way to estimate the value of πis 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 πsquare 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) [20 pts] 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, 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) [20 pts] On the **intel16** cluster, plot the speedup and efficiency curves for your OpenMP parallel versions using1, 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

3) [40 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) [20 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) [15 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) [5 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.

BONUS) [40 pts] Parallel Weight Balancing with OpenMP Tasks

Suppose you have a set of weights $w_1, w_2, \ldots, w_n$, where each weight is a real number. You are asked to divide the weights into two sets such that the difference of the sum of weights in these two sets is minimum. For example, assume that {1.2, 2.3, 4.5, 10.2} is the set of given weights. This set can be divided into two sets as {1.2, 2.3, 4.5} and {10.2}, which minimizes the difference of the sum of weights between them. (10.2 - (1.2+2.3+4.5) = 3.2 is the minimum possible difference).

This is an NP-complete problem, so the only way to find the exact solution is to enumerate all possible combinations and choose the one which gives the minimum difference between the two sets. Therefore you need to create all subsets of the given set of weights. The pseudocode for generating subsets using backtracking method is given below (as well as in the skeleton code provided):

```
S = input set
index = index in set S from where the current subset should start

GenerateSubset(index, subset, S) {
   if (index == S.size) { //then this subset is done
update if it is a better solution than what we have so far
     return
}

subset = subset U S(index) //add the index element to the subset
GenerateSubset(index+1, subset, S)  // generate subsets with the
                                   // index element in them
   subset = subset - S(index) // remove the added element from the subset
GenerateSubset(index+1,subset,S)    // generate subsets without
                        // the index element
}
```

Using the skeleton code provided, first implement a sequential version of the program. Then implement an OpenMP parallel version using tasks — which are ideally suited for parallelizing recursive programs. Each recursive call to the GenerateSubset can be denoted as a new OpenMP task.

Reporting Performance

i) Experiment with different number of weights such as 15, 20, 25 and 30.
ii) For your OpenMP parallel version, experiment with different #threads, i.e. 2, 4, 8, 14, 20, 28.
iii) Plot the speedups for your parallel code with respect to the serial code. Explain your observations.
iv) Note: The skeleton code provided expects the #weights and #threads to be provided as command line arguments. For the serial version, you still need to provide the number of threads, but you will not be using it.

Hints:

- Keep in mind that there will be 2^n number of subsets for a set of n elements. So you should refrain from saving all subsets, as this will require huge memory space. Instead you can keep a current best solution, and update your solution when you identify a partitioning which is better the current one. For initialization, you can start by setting the best solution to be empty and the global minimum to an arbitrarily large value.
- How can you update your current solution? Assume that the sum of all given weights is SUM. Note that the best solution will have the minimum difference between the sum of its weights and SUM/2. So, for each generated subset, you can simply compute the difference between your generated subset's sum and (SUM/2) and update your solution if it is less than your current best.

Instructions:

- **Obtaining files from the git repo & submission.** You will clone the skeleton code and the testing input files through the instructor repository on the Gitlab server. Assuming that you have already cloned the instructor repository, you will need to pull the most recently committed files for HW3 and move them over into your individual repository:

  ```
  cd cse491-instructor
  git pull
  cd ../cse491-${USER}/homework
  cp -r ../../cse491-instructor/homework/3 .
  ```

  Then complete the homework and submit it using your own personal repository. Your submission will include exactly two files:

  - Your source code files:
    - prob2/parallel_pi.c
    - prob3/count_sort_ipar.c
    - prob3/count_sort_jpar.c
    - bonus/subset_tasks.c
  - A PDF file named "HW3_yourMSUNetID.pdf", which contains your answers to the non-implementation questions of the assignment

  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 D2L discussion forum 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 5 measurements and take their averages while reporting a performance data point.*

- **Executing your jobs.** 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:

```
#!/bin/bash -login
#PBS -l walltime=00:15:00,nodes=1:ppn=28,feature=intel16
#PBS -j oe
#PBS -N subset

# change to the working directory where your code is located
#cd ~/cse491-${USER}/homework/3/bonus
cd $PBS_O_WORKDIR

# call your executable with different no. of threads
./subset_tasks  15 1
./subset_tasks  15 2
./subset_tasks  15 4
# list more jobs here…
# all output will be written to the default job stdout.
# if desired, you can redirect individual results to
# a file. i.e., ./subset_tasks  15 4> subset_w15_t4.out
```