CISC 811: High Performance Computing: Take-home final

Set April 23rd 8 am due April 23rd 6 pm
All solutions should be developed on the HPCVL machine sfnode0. For each question provide both an electronic (via email) and hardcopy of your code. **All solutions should be generally applicable and you may NOT make assumptions about the distribution of input keys to aid solution. Any solution that does so will be heavily penalized.**
When the number of possible key values in a sort is much less than the total number of keys a "bucket sort" is a very efficient way of sorting.
Q1. (4 marks) Firstly, construct a sequential version of the algorithm at the bottom of the page and test it using 180×10^6 keys, distributed on the interval [1:16], so that there are 16 buckets. For simplicity and speed of generation, distribute the possible key values in a cyclic fashion within the `keyin` array (*i.e.* `keyin` should be a sequence 1 2 ... 16 1 2 ... 16 ...). Check for the correct output, and use compiler flags to achieve the fastest execution time. Timing should be performed by placing timing tests around the sort section of the code, so as to avoid counting the start-up costs involved with setting `keyin`. Provide details of the compilation flags you used, and execution time in seconds. How much speed increase did optimization produce and is there a specific feature of the algorithm that leads to this answer?
Q2. (6 marks) Parallelize the bucket sort algorithm using OpenMP (parallel regions are the most effective way to parallelize this algorithm) and again use compiler optimization. Think carefully about which parts of the algorithm need to be parallelized and which do not, and also about how to deal with the race conditions in the algorithm. (HINT: small shared 1-dimensional arrays are often made 2-dimensional to avoid race conditions.) Produce a table that includes the execution times for the single processor code and the parallel code run on 1,2,4,8 threads. As for the serial code, only time the sort itself. Graph speed up for the parallel version of the code.
Q3. (10 marks) Construct an MPI version of the code and use compiler flags to optimize performance. To make coding more simple, distribute the `keyin` values across processors so that the values may be generated locally, and each processor holds $180 \times 10^6/(\#$ of processor) keys. `keyout` should be held solely on the root processor, although it will be useful for each processor to use an auxiliary array (of size $180 \times 10^6/(\#$ of processor)) to store its locally sorted values. There will be two main communication steps, one to *gather* the details of the global `bucket` array, and a second one to transfer the sorted data to the root process. As in the previous question produce a table of execution times for 1,2,4,8 processors and graph overall speed-up. Comment on the scalability of this algorithm if the `keyout` array is held on the root process.

```
integer n,nbuckets,i
integer keyin(n),keyout(n),bucket(nbuckets)
do i=1,nbuckets
  bucket(i)=0
end do
do i=1,n
  bucket(keyin(i))=bucket(keyin(i))+1
end do
do i=2,nbuckets
  bucket(i)=bucket(i)+bucket(i-1)
end do
do i=1,n
  keyout(bucket(keyin(i)))=keyin(i)
  bucket(keyin(i))=bucket(keyin(i))-1
end do
```