Programming Assignment 4

Due May 16, 2012 12:50 PM

In this programming assignment you will implement two parallel sorting algorithms, and will learn about OpenMP, an API which simplifies parallel programming on shared memory CPUs.

1 OpenMP

OpenMP is an API which enables simple yet powerful multi-threaded computing on shared memory systems. To link the OpenMP libraries to your C++ code, you simply need to add -fopenmp to your compiler flags. You can then specify the number of threads to run with from within the program, or set environment variables:

```
export OMP_NUM_THREADS=4 (on icme-gpu1)
setenv OMP_NUM_THREADS 4 (on leland machines)
```

We will cover OpenMP in lecture. You can learn more about OpenMP at the official website:
`http://openmp.org/`.

If you find yourself struggling, there are many excellent examples at:
`https://computing.llnl.gov/tutorials/openMP/exercise.html`

See also the documents uploaded on piazza in Resources.

2 Sorting

This assignment asks you to implement two different sorting algorithms, merge sort and radix sort, in parallel. We will use recursion in both implementations. We will only be interested in sorting unsigned integers, to simplify our implementations.

2.1 Merge Sort

Merge Sort is a comparison based sort. The algorithm uses a divide and conquer strategy:

1. Split the array in half
2. Recursively sort each half independently
3. Merge the two halves together by iteratively selecting the smallest element from either list.

We see that the "divide" step of the algorithm is very naturally parallelized, as we can assign different threads to sort the different halves independently since they are sorted independently. We can also parallelize the "conquer" step by merging the two halves from multiple different threads. Part of your task is to understand how we can split these arrays such that we merge different parts of them together in parallel.

2.2 Radix Sort

Radix sort is a non-comparison based sort that sorts based on digit or more usefully for computers, bits. It places each element into a different bucket based on a specific digit (i.e., every number in the 800s is larger than every number in the 700s), and then sorts each bucket independently.

Our approach to implementing this algorithm will be the following:

1. Find the distribution histogram (the number of elements that will be sorted into each bucket)

2. Partition the data by bucket
3. Place the elements into the appropriate bucket
4. Sort each bucket

We will use a Least Significant Digit implementation, which means that we will start sorting with the *rightmost* digit. While this seems counterintuitive, remember that this is a *non-comparative* sort.

To implement this algorithm in parallel, we will first compute the histogram in parallel, performing a reduction to get a single distribution. We can also perform the sorting of the buckets independently.

3 What we give you

- `mergesort.cpp` : This executes the serial C++ STL sort, and shell code for a parallel implementation of mergesort. Do not modify the function headers, but instead only implement the bodies of the functions. We specify thresholds beyond which we call serial sorting algorithms. You should modify this file.
- `radixsort.cpp` : This contains a serial implementation of lsd radix sort, and shell code for a parallel implementation of radix sort. Do not modify the function headers, but instead only implement the bodies of the functions. You should modify this file.
- `Makefile` : This compiles both mergesort and radixsort by running the command `make`. You do not need to modify this file.

By default, these files should successfully compile. This will build the mergesort and radixsort executables:

- `mergesort` : Running this executable will run the serial sort algorithm available in the standard template libraries. You must specify as command line arguments the number of elements to support, and the thresholds for parallelization. Set the last argument as 1 to run the serial code, and 0 to only run the parallel code.
  `./mergesort sortThreshhold mergeThreshhold numElements serial`
- `radixsort` : Running this executable will run our implementation of a serial sorting algorithm. You should modify this file so that it can optionally take the number of elements as its only input.
  `./radixsort`

Both of these executables will check to see if the serial sort produces the same answer as the parallel sort. Initially, this check will fail (as you have not yet implemented the parallel sort!)

4 Testing

The code you turn in should be able to run the following commands:

- `make clean ; make`
- `./mergesort 100000 3000000 48000000 1`
- `./radixsort`

and run the sorting algorithms so that the serial and parallel implementations produce the same output. If they do not produce the same output, they should fail an assert() call already in the starter code.

5 Analysis

Type your responses and submit them as a PDF file. These questions are to help you understand the amount of performance improvement your parallel versions has over a serial implementation.

1. Are these implementations compute or memory bound?

2. Run mergesort with the following parameters (changing the number of threads)

   `./mergesort 100000 3000000 48000000 0`

   for 1, 2, 4, 8, 16, 32, and 64 threads. Plot the running time against the number of threads.

3. Run radix sort with 1,000,000 integers and 16,000,000 integers for both the serial and parallel implementations. Calculate the bandwidth as `number_of_elements / running_time`.

   - Describe in what way the relative performance is different for the two different sizes.
   - Provide one reason this could have happened.
   - Hint: it may be useful to look at the output from running the command `cat /proc/cpuinfo`

4. Suppose your machine has p processors. You create $c\,p$ threads. What is the result when:

   (a) $c < 1$

   (b) $1 < c < 4$

   (c) $c >> 10$

   Does the number of elements we are sorting change your answer? Why or why not.

6 Submission instructions

You should submit the complete source code for your working solution and a brief PDF file containing your responses to the short answer section.

- Create a directory, for example `pset4`, on your Leland account.
- Copy the files you would like to submit in the directory `pset4`. Do not include auxiliary files that are not needed.
- From within the directory `pset4`, run the shell command
  `corn:> /afs/ir/clas/cme213/bin/submit pa4`
  This will send the entire content of the directory to us and will label it `pa4`.

A timestamp is also included. We will use your last submission before the deadline for grading. Please send a message to the course staff through piazza if you encounter a problem.

7 Hardware

We will continue to use icme-gpu1, even though we are now only taking advantage of its CPU capabilities. You may directly run your code on any of the nodes or you may use the torque queuing system.

If you choose to use `msub`, it is helpful to request all of the cores on a node to maximize your performance.

7.1 Using the Torque Queue system

ICME-GPU1 uses the Torque queuing system. The manual is available on-line at
`http://icme.stanford.edu/Computer%20Resources/docs/TORQUE_Administrator's_Guide.pdf`

But for our purposes, it will be sufficient to only use two commands: qsub to submit jobs, and qdel to delete jobs.

To run a job with qsub, you must provide a script containing the commands you wish to run. We have created an example script in `pa4.pbs`. To run this job, simply execute:
`qsub pa4.pbs`

You may need to update the `$USER` or `$pa4_home` values within the script if you have configured your system with something other than your username and put your code in something other than `/home/sunet/pa4`.

The output will be stored in a file named `<script>.o.<jobID>` and `<script>.e.<jobID>`. For example, if this was job 15, the standard output would be placed into `pa4.pbs.o15` and standard error in `pa4.pbs.e15`. You should not be executing more than five jobs concurrently, or else you may get kicked off the cluster.

You can delete a job by running

```
qdel jobID
```

You can check the status of the queue with

```
qstat
```

Torque is designed around throughput, not latency, so an individual job may take longer to run than you would expect. We encourage you to develop locally and only test with qsub once you have a final version of your program.