CS133 Lab3

Xiaohui, Zhou

104-014-248

a) Please briefly explain the sorting algorithm you have used. How is it parallelized? What is the complexity of this algorithm? What is the expected communication overhead?

ALGORITHM:

I used a modified version of the sorting algorithm called burst sort.

The basic idea of the original algorithm is using a trie to store prefix of strings, and each node (bucket) in that trie has a growable array for storing contents. As the buckets grow, then the buckets are "burst". As the bucket is much smaller than the total size of the data, the memory usage will be much smaller and with a better cache efficiency as the bucket is small and can fit into cache.

In my implementation, for simplicity reason, instead of using a real trie, I used 95^2 mini-buffers (95 is the number of elements can be found in the key) to simulate a 2 level trie. Then, I would sort all these mini-buffers and then produce the result.

Related information:

http://en.wikipedia.org/wiki/Burstsort

http://goanna.cs.rmit.edu.au/~jz/fulltext/alenex03.pdf

HOW PARALLELIZED:

Because openCL cannot pass a large number of small buffers into the kernels, so in the parallel version, I merged all these mini-buffers into a single buffer and provide an integer array storing the starting and ending positions for all the mini-buffers. Then the kernel would sort a portion of mini-buffers. For each mini-buffer, the kernel will use quick sort for in-place sorting. Then the single buffer will have the correct result afterwards.

COMPLEXITY:

It will take O(n) time to load all the data, and then O(l * (m log m)) time for sorting, where l is the number of buckets used, and m is the size of the buckets.

In my implementation, l is approximately 10,000 and m is 1,000 if the number of entry is 10,000,000.

In this case, ` l * m log m` is approximately equal to 100,000,000, which is close to 10 * n. Thus the performance for this algorithm is much better than only use quicksort or merge sort.

OVERHEAD:

Because openCL cannot pass a number of buffers and so compare to the serial version, I need to allocate an additional single big buffer for passing to the kernels. This also includes the time needed for copying data from mini-buffers to the big buffer. Thus the I/O overhead is quite big. By comparing the serial version and the parallel version, I found the their total running time is close, so I believe that the time shorted by parallel computation is close to the extra time taken by the I/O overhead. I believe that if the number of entries increased significantly, for example, 1000 times larger, then this algorithm will have a much better performance than simply using quick-sort or merge sort.

<u>b) Measure the execution time of your serial version. Using this number as the baseline, report and discuss the scalability of your parallel algorithm using 1, 2, 4, 8, 16, 32 processors.</u>

Serial version: 32 seconds

Parallel version:

1 processors: 39 seconds	**2 processors:** 36 seconds
4 processors: 35 seconds	**8 processors:** 33 seconds
16 processors: 32 seconds	**32 processors:** 28 seconds

I/O will take the most of the used for running the algorithm and so it is not clear for the use of parallel version with openCL.

<u>c) Please list and quantify all optimization you have used.</u>

1. Using simplified burst sort for cache efficiency, as discussed above.

<u>d) Challenges you encountered and how you solved them</u>

1. How to simply the burst sort algorithm and keep its advantage of cache efficiency. Used 10000 buckets to simulate a 2 level trie.

2. How to pass all these mini-buffers to the kernel. Solved by using a big buffer and an integer array for recording the positions for each mini-buffer.

3. How to run the quick sort without using recursion. Solved by using a simple stack.

4. How many level of trie I should simulate. Solved by trying with 1 level, 2 levels, and 3 levels, and doing calculations on time complexity.

<u>OTHERS:</u>

I believe that this algorithm should have a better performance if I use GPU instead of CPU because there are a large number of mini-buffers. Also, it is more suitable for openMP which do not require sending only a very small number of buffers.