A parallel implementation of bitonic sort in combination with quick-sort , using pthreads

Christophoros Bekos (mpekchri@auth.gr)

October 25, 2017

1 Introduction - Bitonic Sequence

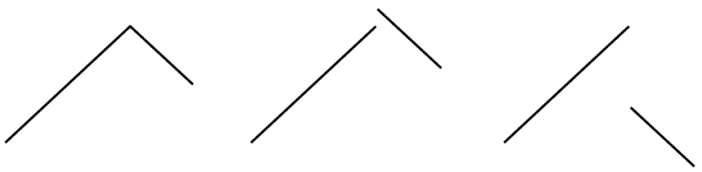
Sequence a_n is bitonic if it monotonically increases and then monotonically decreases, or if it can be circularly shifted to monotonically increase and then monotonically decrease.
In other words, an array $x[n]$ is Bitonic if there exists an index i where 0 $\leqslant$ i $\leqslant$ n-1 such that :

$$x_0 \leqslant x_1 \leqslant ... \leqslant x_i \ \ and \ \ x_{i+1} \geqslant x_{i+2} \geqslant ... \geqslant x_{n-1}$$

Note that :

1. If the direction changes more than two times we cannot have a bitonic sequence.
2. If there are two changes in direction, we MAY have a bitonic sequence.
3. The reverse, as well as, any cyclic rotation of a bitonic sequence are also bitonic sequences.

The following figures will help you get a better understanding :



Those 3 figures consist bitonic sequence (note that sequence does not need to be continuous)

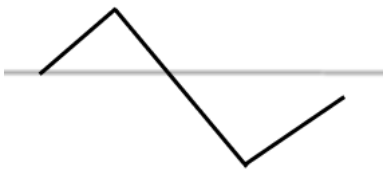


Figure is a bitonic sequence

Figure is NOT a bitonic sequence

2 Bitonic Sort

Bitonic mergesort is a parallel algorithm for sorting an array, whose elements consist a bitonic sequence . Pseudo-code for merge-sort is given bellow :

Algorithm 1 Bitonic Merge

```
1: function BITONICMERGE(int* array ,int lo, int cnt, int dir)
2:  if(cnt>1) {
3:     int k = cnt/2
4:     for (int i=lo; i<lo+k; i++){
5:        compare(array,i, i+k, dir);
6:     }
7:     bitonicMerge(array, lo, k, dir);
8:     bitonicMerge(array, lo+k, k, dir);
9:  return;
10: }
```

bitonicMerge will always sort the array since it's elements consist of a bitonic sequence. If dir is equal to 'asc' then array is sorted in ascending order, otherwise (dir == 'desc') in descending . The compare function which is used , is described bellow :

Algorithm 2 compare

```
1: function COMPARE(int array,int lo, int j, int dir)
2:  if (dir == (array[i]>array[j])) {
3:     exchange(array,i,j);  }  return;
```

Where exchange(int* array,int i,int j) function swaps the elements in positions i and j .
The sequential complexity of bitonicMerge is $O(nlog(n))$. Function bitonicMerge can be executed in parallel as shown in algorithm 3. Thus the parallel complexity of bitonicMerge is $O(log^2(n))$ since we are using n processors (actually the same bound can be achieved with n/2 processors) .

Algorithm 3 Bitonic Merge

```
1: function BITONICMERGE(int* array ,int lo, int cnt, int dir)
2:  if(cnt>1) {
3:     int k = cnt/2
4:     for (int i=lo; i<lo+k; i++){
5:        compare(array,i, i+k, dir);
6:     }
7:     # parallel (done by current thread) bitonicMerge(array, lo, k, dir);
8:     # parallel (done by a new thread) bitonicMerge(array, lo+k, k, dir);
9:  return;
10: }
```

3 Quick sort

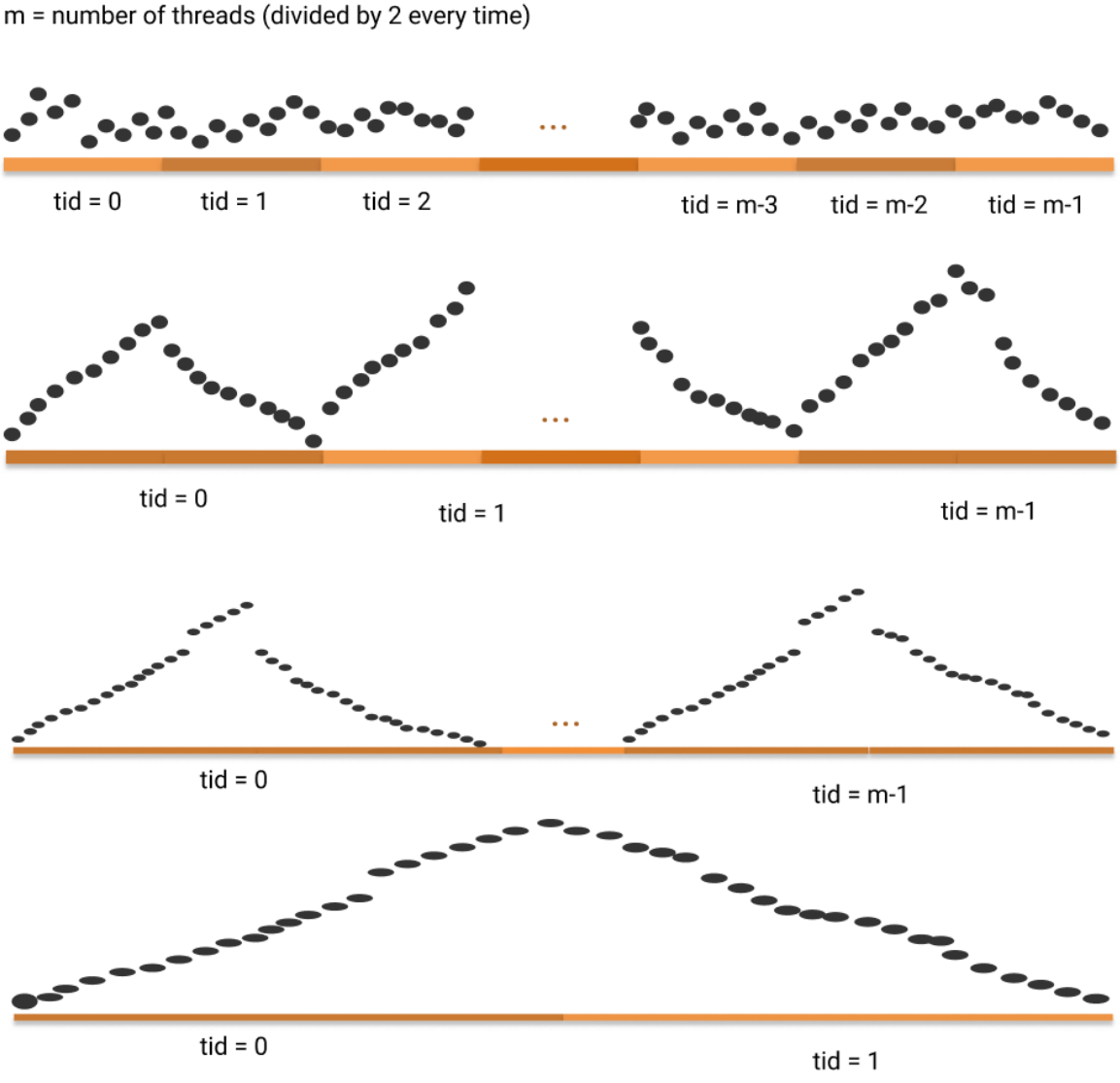
QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of QuickSort that pick pivot in different ways. QuickSort's worst-case running time is as bad as selection sort's and insertion sort's: $\Theta(n^2)$. But its average-case running time is as good as merge sort's: $\Theta(nlog(n))$ .So why think about QuickSort when merge sort is at least as good? That's because the constant factor hidden in the big-O notation for QuickSort is quite good. In practice, QuickSort outperforms merge sort, and it significantly outperforms selection sort and insertion sort. In our implementation we will use function qsort() (a C library function) to implement QuickSort .
You may read more about quick sort here , or find qsort's documentation here .

4 Our Implementation using Pthreads

Algorithm 3 , as showed above , requires the existence of n (or $\frac{n}{2}$) processors , or in other words, it requires a lot of processors . Thus , it would make sense to try such an implementation if we wanted to create a CUDA application , but since in this project we will use pthreads , we must consider that demanding a huge amount of threads would cause poorer performance .

4.1 Combining quick and bitonic sort

Having all these in mind , we will start a prudential number of threads (t-num new threads in total) and each one of them will, **[step 1] :** use qsort() to sort a continues part of our array ; threads with even id sort in ascending order ,while threads with odd id sort in descending order . Thus , we create a set of sub arrays whose elements consist of a bitonic sequence. Therefore, **[step 2] :** we may now start $\frac{t-num}{2}$ threads , which will use bitonic sort to sort , using a pair of the sub-array's set (pair consists of a sub array whose elements are sorted in ascending order, and one sub array with elements in descending order) . Obviously each thread modifies two continues sub-arrays in order code to be cache friendly . **[step 3] :** This procedure can be executed **recursively** , and the number of threads will keep be divided by two until we reach one thread .
The aforementioned procedure is showed in figure below :



4.2 The c code

The file named 'parallel.c' contains the code that is described bellow (makefile is also provided) . You may run the file by opening an terminal and typing **./bitonic 23 4** , where array a will contain N = 2^{23} elements and our program will use 4 threads (plus the one running) . Our task is to define something faster than quick-sort , thus total execution time of our function is compared to qsort()'s execution time . As said before , main() takes as argument the p = desired number of active threads . Then pre-sort() is called and creates $2 * p$ threads , which one of them, uses qsort() to sort a sub-array of a , in ascending/descending order and then all my-sort() is called. My-sort() calls bitonicMerge() and compare() functions in different threads , as explained before. In case you want to create some plots or keep some statistics you may modify or use 'run-tests.sh', **in folder 'count-time'** , which is a bash file that runs our script for different number of N (array's a length) and different number of threads . You might also find useful 'visualize-results.m' , a matlab's file to plot our simulations results .

4.3 Code's weak spots - further improvements

Of course this code is not optimal : main thread is not used in implementations ,contrariwise it just waits for other threads to finish their job (wasting cpu power and recourses). Also using join function as a barrier it's not as optimal as using direct signals , pthread-cond-wait and pthread-cond-signal could be used instead .

5 Results