EE 382V: Parallel Algorithms **Summer 2017**

Lecture 9: June 17

Lecturer: Vijay Garg *Scribe: Tiffany Tillett*

9.1 Introduction

During this lecture, we discussed the following popular parallel sorting algorithms:

- Radix Sort
- Merge Sort
- Quick Sort

This set of lecture notes will briefly re-examine the topics covered in this lecture, in the order in which they appeared during class. Note: This order is different than that of the lecture notes (lect-sort.pdf)

9.2 Radix Sort

Radix Sort is a non-comparison based sorting algorithm. It utilizes the radix r to place elements into buckets based on their value. The radix is also known as the base. For example:

- base 2
- base 8
- base 10

9.2.1 The Deck of Cards Example

In the case of a deck of cards, we know that there are exactly 52 cards divided into 4 suits with 13 unique cards belonging to each suit. We can create a bucket for each suit:

- Spades
- Hearts
- Diamonds
- Clubs

Within each of these 4 buckets, we can create buckets for each of the 13 card values. Once the cards are all divided into the correct buckets, it is straightforward to put them in the correct order.

9.2.2 A Numerical Illustration

Now lets look at an example using numbers. We have an array $A = [35, 42, 71, 21, 74]$. In this case, the radix is the set of numbers 0...9 and we have two digits.

Lets look at the case where we divide based on the most significant digit (MSD) first:

radix	0	1	2	3	4	5	6	7	8	9
values			21	35	42			71, 74		

We would next divide each box into ten sub-boxes and repeat for the least significant digit (LSD). However, we will instead look at the case where we divide based on the LSD first.

radix	0	1	2	3	4	5	6	7	8	9
values		71, 21	42		74	35				

If we then take the elements in order, we have an array $B = [71, 21, 42, 74, 35]$. Again, we would next divide each box into ten sub-boxes and repeat for the MSD. After the second pass, we have our final sorted array $C = [21, 35, 42, 71, 74]$. `We will always be able to compute a sorted array in` d `passes where` d `is the number of digits.`

9.2.3 The Concept of Stable Sort

$StableSort \equiv$ A sorting algorithm is stable if it preserves the order of entries that are equal.

For example, we can see that in the second pass, both 71 and 74 will be in buckets for $MSD = 7$. However, we must maintain the results from the first pass in order to ensure the correct final order.

9.2.4 The Serial Radix Sort Algorithm

```
1  for i:= 1 to d do
2    StableSort on digit i
```

This algorithm has time complexity $O(d * n) = O(n)$ meaning that we iterate through all of the elements d times where d is the number of digits and n is the size of the array.

9.2.5 Parallel Radix Sort

Lets look at the case where all numbers are binary. On the first pass, we will look at the LSD (right most digit). This process will result in all elements ending in 0 rising to the top and all elements ending in 1 falling to the bottom. Note that we are using stable sort, so the order of the elements ending in 0 will be maintained. Similarly, the order for elements ending in one will also be maintained. We will move one place to the left and repeat the process until we have looked at all 4 digits.

initial	after 1st pass	after 2nd pass	after 3rd pass	final
1011	0110	1000	1000	0000
0110	1000	0000	0000	0110
0111	0000	0110	1011	0111
1000	1011	1011	0110	1000
0000	0111	0111	0111	1011
1111	1111	1111	1111	1111

We will use a function called `compacting an array`. The steps are as follows:

```
1 filter out interesting elements
2 make a new array with a value of 1 for the interesting elements and a value of 0 for
3 use scan (exclusive prefix sum) to calculate the new indices for the interesting ele
4 calculate the indices for the non-interesting elements
5 drop all elements into the new array at the correct position
```

Here is an example where we want to move all elements with the last number of 0 to the front of the array:

A	010	111	110	001	000	101
match	1	0	1	0	1	0
scan(match)	0	1	1	2	2	3
B	010	110	000	111	001	101

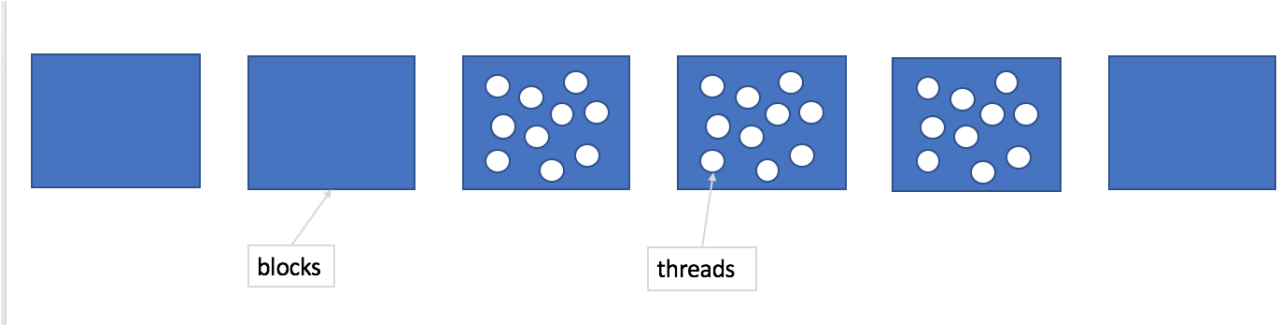
9.3 Merge Sort

Merge Sort is a common sorting algorithm. The simple version of the algorithm is:

```
1 mergesort(Left Half)
2 mergesort(Right Half)
3 merge(Left Half, Right Half)
```

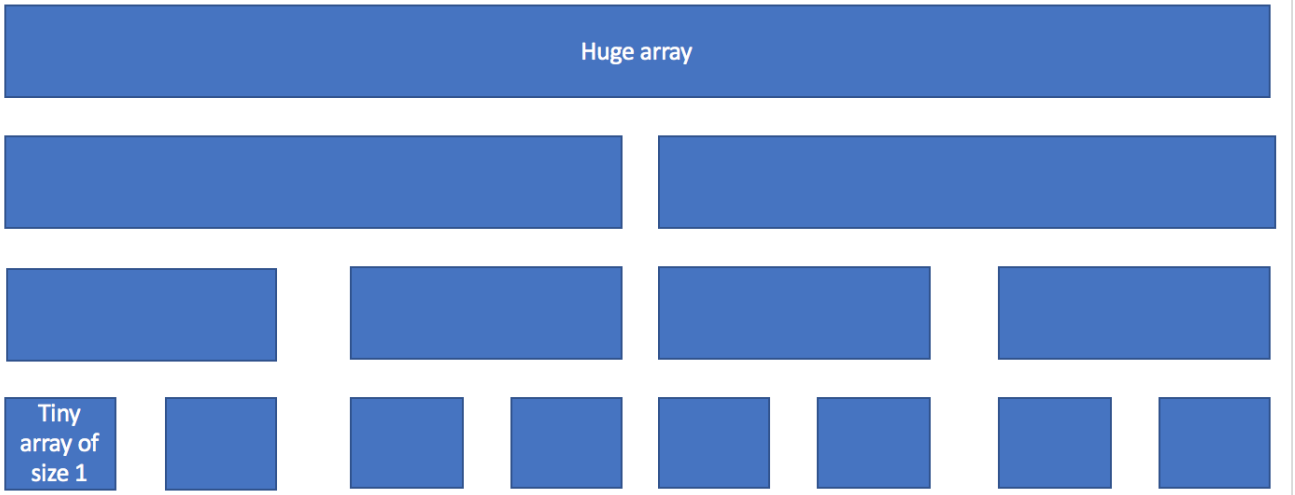
The recursive calls to mergesort can be easily executed in parallel. The interesting part here is doing the merge step in parallel.

9.3.1 GPU Architecture Review



9.3.2 Maximizing Parellelism with MergeSort

The merge sort algorithm will divide the array in a manner similar to what is shown in the image below:



To have maximum parallelism, we want to use as many threads as possible at any given time. We have lots of parallel merge operations happening near the bottom of the tree. We can have each thread do its own merge using the sequential algorithm since each merge is so small. As we move to the middle of the tree, we no longer have enough merge operations to make the sequential algorithm effective. Here, we will switch to the parallel merge algorithm that was discussed in a previous lecture (lect-merge.pdf). Near the top of the tree, we have only a few large merges. Here, we will divide each merge into a lot of merge tasks to create parallelism. The idea here is that we choose our merge algorithm based on the size of the arrays that we are merging.

An example of this on the GPU would be:

Stage 1: Within a block, we can use many threads to compute the small merge threads.

Stage 2: At a certain point, each block will only have two sections to merge (one merge task). This is when we switch to the parallel merge algorithm.

Stage 3: Once each block has completed its merging, we now need to merge the results from each block to get the final result. Here is where we use the idea of splitters to break each merge task into multiple merge tasks so that each merge task can be assigned to a different block.

Using parallel merge, we have:

$T(n) = T(n/2) + O(logn)$

$T(1) = 1$

$T(n) = O(log^2 n)$

9.3.3 Merge Sort in $logn$ time?

There is a complicated algorithm called Cole's Algorithm which can do sort in $O(logn)$ time. However, this algorithm comes with a lot of overhead and is not used in practice.

9.4 QuickSort

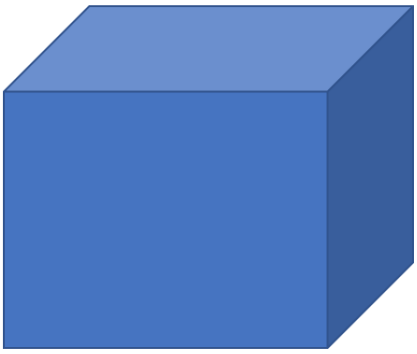
The concept of QuickSort is as follows:

```
1 choose a pivot element
2 partition the array
3   all elements smaller than the pivot will be to the left of the pivot
4   all elements larger than the pivot will be to the right of the pivot
5   the pivot will be in its final sorted position
6 quicksort(left)
7 quicksort(right)
```

The entire algorithm will take $O(n^2)$ time in the worst case and $O(nlogn)$ time in the average case.

Let's look at a hypercube example.

Assume that we have p processors and each node has a chunk of the array where each chunk is of size n/p.



Each node has an address:

- 000
- 001
- 010
- 011
- 100
- 101
- 110
- 111

9.4.1 QuickSort Algorithm 1: Parallel QuickSort

```
1 Some processor chooses a pivot value x
2 That processor broadcasts x to all processors
3 Each processor will partition their portion into two parts ( <= x, > x)
4 Assign each processor a partner
5 Each partner will exchange their unneeded portion with their partner.
6   If I am 000, I will send values > x to 100
7   If I am 100, I will send values <= x to 000
8 proceed recursively until the array is sorted
```

9.4.2 QuickSort Algorithm 2: HyperQuickSort

```
1 Each processor will sort its own portion
2 Processor0 chooses the median as the pivot
3 Processor0 broadcasts to all processors
4 Proceed as above except we exploit the fact that all arrays are sorted.
```