MichaelDuanGPU

Xuheng Duan

March 2019

1 Michael Duan

1.1 Introduction: List Ranking Problems

Generally, in List Ranking problems, we need to find the distances of each node to the end of the linked list. In the following sections, I am going to introduce one naive way, and a corresponding way to solve it in parallel. Furthermore, I will introduce ways to improve the original algorithms.

1.2 Pointer Jumping Algorithm

In general, we rely on the pointer jumping algorithm to figure out the distance between certain node to the end of the list.

1.2.1 The assistant lists: Successor and Result

In order to apply pointer jumping algorithm, firstly we have to initialize two extra lists to help us maintaining the structure of the single linked list and recording the final result. Namely, we use one list, Successor, and another list, Result, to do the job.

Successor, as its name suggests, is a list which contains current node's successor. For instance, imagine there is a successor list: $2- > 5- > 4- > 0- > 6- > 3$. Then, the first 2 in the list suggests that 2 is node 1's successor; 5 is node 2's success and so on. In another word, if we translate the successor list to a normal linked list, then we will have: $1- > 2- > 5- > 6- > 3- > 4$. Since node 4's successor is zero, then we can tell that node 4 is the end of the linked list.

Meanwhile, we need another helper list to record the results, which are the distances from one certain node to the end of the linked list. For instance, in our previous example, the result list will be: $5- > 4- > 1- > 0- > 3- > 2$, because node 1 is 5 nodes away from node 4, and node 2 is 4 nodes from node 4 and so on. Initially, the result list is slightly different. We initiate our result list by assigning 0 to the end node, and 1 to the rest. So, the initial result for the example above should be like: $1- > 1- > 1- > 0- > 1- > 1$

1.2.2 Basics of Pointer Jumping

In short, the pseudo code for pointer jumping algorithm is straightforward. Here is the original pseudo code from the book

Algorithm 2.2:

```
Input: 1. initial result list (Using R to represent it);
       2. Successor list (Using S to represent it);
Output: final updated result, R;
Begin:
    for 1 <= i <= n, pardo
        set Q(i) = S(i);
        while (Q(i) != 0 and Q(Q(i)) != 0):
            Set R(i) = R(i) + R(Q(I));
            Set Q(i) = Q(Q(i));
End
```

Implementation: In Linear the implementation in linear is straight forward.

```
int *updateRes(int *r, int *s, int len){
  //              result, successor, lenth of array
  int *res = r;
  int *q = s;
  //can use q[i] to find the element
  for(int i =0; i< len; i++){
    while(q[i] != 0 && q[q[i]] != 0){
      res[i] = res[i] + res[q[i]];
      q[i] = q[q[i]];
    }
  }
  return res;
}
```

the input r is the initial result list, which is a bunch of zero, input s is the initial list of successor, and input len is the length of the list. I implemented initial functions to set the initial values for r and s. To use updateRes, we just put in the initial r and s list. Implementation: In Parallel Similar to the functions in subsection 2.2.1, we just transform our algorithm in to cuda language.

```
__global__ void updateResAll(int *r, int *s, int *q, int n){
  int i = threadIdx.x;
  if (i < n){
    q[i] = s[i];
    __syncthreads();

    //try to update all
    while(q[i] != 0 && q[q[i]] != 0){
      r[i] = r[i] + r[q[i]];
      q[i] = q[q[i]];
      __syncthreads();
    }
  }
}
```

This kernel can deal with the data up to the volume of 1024, because it only calls one block to complete the work. Due to cuda language, we can easily synchronize our threads in one block, but cuda doesn't support sychronization between different blocks.Thus, we have to figure a way to solve this problem.

Implementation In Parallel, Optimization For Blocks Firstly, here is the kernel:

```
__global__ void updateOnceBetweenBlocks(int *r1, int *r2, int *s1, int *s2, int n){
  //update only one round. Use if to control it;
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if( i <= n){
    int next = s1[i];
    int nextNext = s1[next];
    int check = (next != nextNext);
    r2[i] = r1[i] + check*r1[next];
    s2[i] = nextNext;

    // if(s1[i] != 0 && s1[s1[i]] != 0){
    //   res2[i] = res1[i] + res1[s1[i]];
    //   s2[i] = s1[s1[i]];
    // }
  }
}
```

this kernel can make sure that blocks and blocks are synchronized by accessing global memory each time it finished doing work. The down side is that it only achieves one updates in a cycle. Thus, the main function will decide how many cycles are going to happen. Generally, the point jumping algorithm can finish in O(Log n) time, which means ideally, the main function only need to call updateOnceBetweenBlocks logN times. Therefore, I come up with the following main function:

```
int counter = 1;
int div = 2;
while(counter <= arrayLen/2){
  //1->2, 2->1
   //<<<block,thread>>>
  updateOnceBetweenBlocks<<<div,arrayLen/div>>>(devRes1, devRes2,devArr1,devArr2,arrayLen);
  updateOnceBetweenBlocks<<<div,arrayLen/div>>>(devRes2, devRes1,devArr2,devArr1,arrayLen);
  counter = counter * 2;
}
```

In my main function, I set the boundary as half of the input length, just to make sure that the algorithm won't miss any nodes. However, theoretically, it is not necessary.

1.3 Optimization: Independent Sets

This section will introduce one way to optimize our previous algorithm, which is "Independent Sets". The general idea for this optimization has following steps:

1. Prepare several arrays to help us record the necessary data: First array, r, standing for result, stores the data of distances.

Second array, s, standing for successor, stores the data for successors.
Third array, p, standing for predecessor, stores the data for predecessor.
Last array, u, is an empty 2-D array. We are going to store corresponding information about independent sets in u.
2. u list is an Nx4 2-D array, which N is the length of our original input list. For each row, we need to store the node's value, predecessor, successor and distance.
Here is the original pseudo code from the book:

```
input: A linked list of nodes, and corresponding list S(i) for each node's successor
output: distance R(i) of node i from the end of the list


for 1<=i<=n pardo:
    set Q(i) = S(i)
    while (Q(i) != 0 and Q(Q(i)) != 0):
        R(i) = R(i) + R(Q(i))
        Q(i) = Q(Q(i))
```

However, the original pseudo code does not have a clear way to preserve the data we need in later step. Thus, I made some adjustments on the original pesudo code. Here is the kernel I design to extract the independent set from the original set:

```
__global__ void indepSet(int *r1, int *r2, int *s1, int *s2,int *p1,int *p2, int n, int *u){
//use 1 as prototype, and store data to 2
  int id = threadIdx.x + blockDim.x * blockIdx.x;
  if( id <= n){
    if(id % 2 == 0){
      r2[id-1] = r1[id-1]+ r1[id];
      p2[s2[id]] = p1[id];
      s2[p2[id]] = s1[id];
//// --------store data-------
      u[4*id+0] = id;
      u[4*id+1] = p1[id];
      u[4*id+2] = s1[id];
      u[4*id+3] = r1[id];
      r2[id] = 0;
      s2[id] = 0;
      p2[id] = 0;
    }
  }
}
```

For each single list, I create two alike ones (for instance, r1 and r2) to prevent threads interfering each other. Also, in my demo, the input list is going to be a pre-design ascending list, from 1 to N. Therefore, I just sift nodes with even values out and keep the odd node in place.
In detail: firstly, I assign thread id for each thread, and mark those threads with even id. Then, I start updating the nodes by putting the data in secondary lists(r2,s2,p2). Imagine the original list is 1->2->3->4->5->6. When we pick even nodes out, the new list would look like this: 1->3->5. So we need to change the successors, predecessor, and distance of the odd nodes. Then, we update

the u list. For each deleted even nodes, we load its node's value, predecessor, successor and distance in to the u list. One important thing here is that I did not find a feasible way to transfer 2-D list between CPU and GPU, so I transfer the 2-D list into a 1-D list. After we settle it down, we mark even nodes' successors and predecessors to 0, meaning that we don't need them anymore, for now. Let's make a simple example here: we delete 2 from the list, so in u list, <2,1,3,1>, standing for node 2, predecessor is 1, successor is 3 and distance is 1 (by default).
What happens to the odd nodes? Now they have changed successor, predecessor and distances. Successor and predecessor are easy parts. For distance, I add the odd node's distance value to the one before it. For instance: 1->2->3->4->5->6. When we delete 2, I add 2's distance to 1; 4's distance to 3 and so on.
We use the kernel in this way:

```
indepSet<<<div,arrayLen/div>>>
(devRes1,devRes2,devSuc1,devSuc2,devPre1,devPre2,arrayLen,devStorage);
```

After GPU done its work, devRes2, devSuc2, devPre2 and devStorage have the data we need.

1.3.1 Optimization: Optimal List Ranking

Finally, we can put everything together, and get the optimal list rnaking algorithm:
1. Generate successor and predecessor lists. I simply used an ascending and an descending list to represent the list: 1->2->3->4->...->N.
2. Initialize the result list.
3. Use kernel "indepSet" to contract the list.
4. Use kernel "updateOnceBetweenBlocks" to update the distances
5. Using the data stored in U list to restore the predecessor list and true distance.
The sample code is on my github.

1.4 Analyze: Running Time

I ran the program on CPU and GPU, respectively. 1.

List size(Node number)	1024	1024*256	1024*512	1024*1024
Linear algorithm	3.595	169406.8	Too long	Too long
Parallel Without Independent sets	0.20	1.134	1.75	3.03
Parallel With Independent sets	0.19	0.003	0.0039	0.003

Table 1: Run time in Ms, both parallel algorithm has 2 blocks

Block size	2	16	64	256
Parallel Without Independent sets	3.03	2.999648	3.004608	3.009216
Parallel With Independent sets	0.003	0.003936	0.003904	0.003904

Table 2: Run time in Ms, the input list size is fixed at 1024*1024

I tested the performance of my algorithm by assigning them different number of nodes and different number of blocks. In the first table, I compared the performance between linear algorithm,

naive parallel algorithm and the parallel algorithm with independent sets. I only tested for four different numbers of input. According to the result, it is clear to see that parallel algorithms take shorter time, and our independent sets actually work as we expected, further decreasing the performance time.
In table two, I focused on testing the block number's influence on running time performance. However, even though I adjusted the block number numerous times, the running time does not change a lot at all.
In conclusion, I suggest that my algorithm works in general, and parallel algorithm can significantly boost the performance as we expected in the beginning.

1.5 Future Possible Improvement

1. I only implemented and tested the algorithm based on pre-made input lists, namely a successor list containing ascending numbers (1->2->...->n) and another predecessor list containing descending numbers (n->n-1->...->2->1). As needed, I can write some helper functions to generate randomized successor and predecessor lists. After that, I may adjust my algorithm to fit the randomized input lists.
2. MY "Independent Set" function will only contract the input list once, which means it only decrease the list size by half. If I want to further decrease the list size, I need to manually adjust the variable to achieve so. I may improve this function by making it a recursive call, which can automatically decrease the input list size to O(1).