石邢越 16337208

张永东教授

高性能计算程序设计基础

2019 年 1 月 24 日星期四

期末实验: List Ranking

1 实验目的

- 查阅文献了解 List Ranking 问题，学习已提出的各种算法，如 Wyllie 算法和 Random Mate 算法；
- 对串行代码进行可串行化分析。分析串行版本代码是否可以以及如何修改成并行版本代码；
- 进一步巩固 MPI 和 CUDA 程序的调优。

2 实验要求

根据 List Ranking 算法完成串行版本代码，分析串行版本代码的可并行性，将串行代码改写成 MPI 并行版本和 CUDA 并行版本，并比较三种版本代码的运行速度。

3 算法原理

3.1 问题描述

List Ranking 是一个比较经典的研究并行算法设计的问题（经典到我在 Google 上甚至很少找到 2000 年后的相关资料），这主要是因为这个问题的并行算法相对于串行版本很难有十分明显的效果改进。

List Ranking 问题描述如下：给出一个结点的值为非负整数、长度为 N 的链表，链表中的 N 个结点的值就是从 0 到$(N-1)$这 N 个数，这 N 个数的顺序是打乱的。现在我要做的是计算其中各个值 i 对应的 rank，rank 定义为

$$\text{rank[i]} = \text{链表中值为 i 的那个结点到链表尾部的距离.}$$

我看到不少 slides 上给 rank 的定义是到链表头部的距离，到头部和到尾部两者区别不大，可以用同样的算法计算，算出来一种后用链表长度减一下就可以得到另一种。

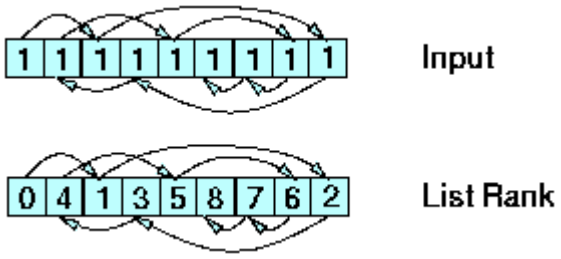


Figure 1 Linked List Ranking 问题的示意图

3.2 串行扫描算法

最简单的串行算法即扫描整个列表。对于每一个结点，记它上面的数值为 value，最后得到的所有 rank 放在一个记为 Rank 的数组中，代码如下：

```
1  node* nodeptr=head;
2  int dis=N-1;
3  int[N] rank;
4  for (int i=0; i<N; i++){
5        int value=nodeptr->value;
6        dis=dis-i;
7        rank[value]=dis;
8  }
```

Figure 2 最初的算法尝试——串行扫描算法。

我分析认为串行扫描算法修改出来的并行算法不会有理想的加速效果，这是因为列表这种数据结构不可以像数组，映射等数据结构一样通过 key 来找到对应的 value，也就是说它不是键值对类型的数据结构。那么如果我把扫描的工作放到 p 个处理器上，有的处理器就得先把自己负责的那些结点前面的结点也扫描了，这会导致大量重复的工作（尤其是当 p 增大时）。

最后我的实验没有用这种算法，用的是使用了 pointer jumping 的 Wyllie 算法。

3.3 Wyllie 算法

Wyllie 算法即使用 point jumping 来计算各个 rank，是最简单的并行算法。算法流程和伪代码如下：

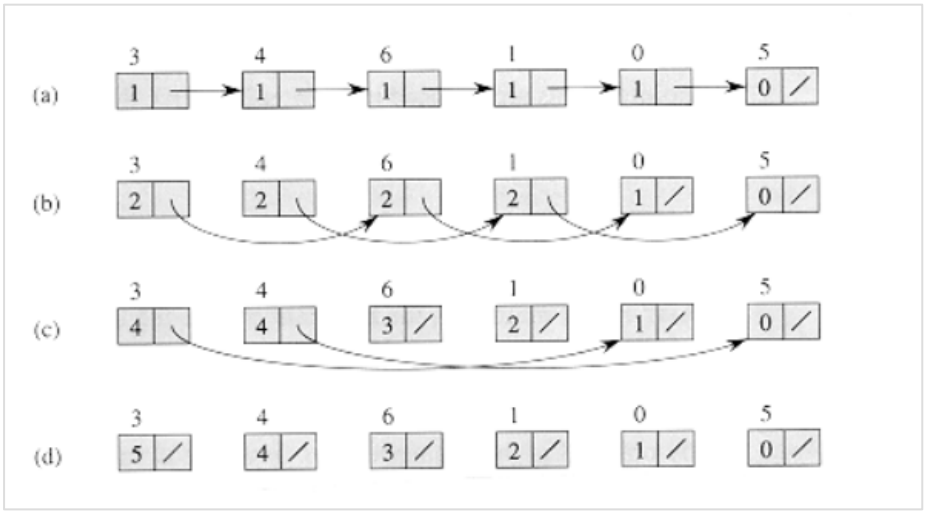
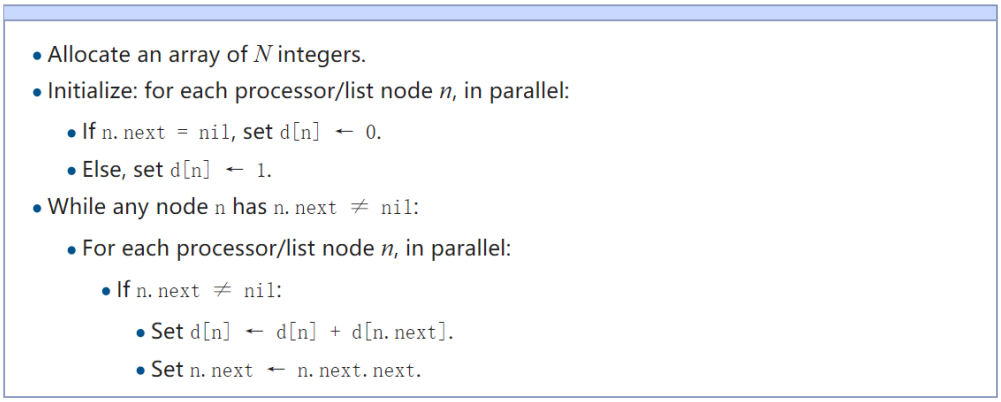


Figure 3 Point Jumping 算法的 List ranking. 经过 4 次循环，每一个结点都没有后继结点了，算法完成，此时已经得到了各个 rank。



- Allocate an array of N integers.
- Initialize: for each processor/list node n, in parallel:
  - If `n.next = nil`, set `d[n] ← 0`.
  - Else, set `d[n] ← 1`.
- While any node `n` has `n.next ≠ nil`:
  - For each processor/list node n, in parallel:
    - If `n.next ≠ nil`:
      - Set `d[n] ← d[n] + d[n.next]`.
      - Set `n.next ← n.next.next`.

Figure 4 Wyllie 算法的伪代码，while 循环部分使用了 pointer jumping。实际编写代码的时候，我把 while 循环单独编写成一个 point jumping 的函数。

Algorithm 1 Wyllie's Algorithm

Input: An array S, containing successors of n nodes and array R with ranks initialized to 1
Output: Array R with ranks of each element of the list with respect to the head of the list

1: **for each** element in S **do in parallel**
2: **while** $S[i]$ and $S[S[i]]$ are not the end of list **do**
3: $R[i] = R[i] + R[S[i]]$
4: $S[i] = S[S[i]]$
5: **end while**
6: **end for**

Figure 5 并行的 Wyllie 算法伪代码。每个 item 对应的 rank 的计算过程可以并行开来执行。

最后写出来的串行代码如下：

```
1  void wyllie_list_rank(int* S, int N) {
2        //R -array to store ranks
3        //length = N-1
4        //set value to 1 everywhere except at the tails
5        int* R;
6        R = (int*)malloc((N-1) * sizeof(int));
7        for (int i = 0; i < N; i++) {
8              if (i != 0) {
9                    R[i] = 1;
10             }
11             else {
12                   R[i] = 0;
13             }
14       }
15
16
17       for (int i = 0; i < N-1; i++) {
18             while (!(S[i]==-1||S[S[i]]==-1)){
19                   R[i] += R[S[i]];
20                   S[i] = S[S[i]];
21             }
22       }
23
24       //print result
25       cout << "Here is the array of ranks:" << endl;
26       for (int i = 0; i < N-1; i++) {
27             cout << (N-2)-R[i] << " ";
28       }
29       cout << endl;
30
31       return;
   }
```

MPI 的 0 号进程的代码分别如下。其他进程则去掉 send 和最后输出结果的部分。

```
for(int i=1;i<num_procs;i++){
      int col=0;
      for(int j=i*chunk_size;j<((i+1)*chunk_size);j++){
          sub_arr[col]=input[0][j];
          sub_arr[col+1]=input[1][j];
          col+=2;
      }
      MPI_Send(sub_arr,chunk_size*2,MPI_INT,i,WORKTAG,MPI_COMM_WORLD);
}
int col=0;
for(int j=0;j<(chunk_size);j++){
   sub_arr[col]=input[0][j];
   sub_arr[col+1]=input[1][j];
   col+=2;
}

int present=0,start=1;
for(int i=start;i<2*chunk_size;i+=2){
   if(sub_arr[i]==0){
     ranked[++index]=sub_arr[i-1];
     sub_arr[i-1]=sub_arr[start-1];
     sub_arr[i]=sub_arr[start];
     start+=2;
     index_prev=index;
     for(int j=1;j<num_procs;j++){
          MPI_Ssend(&index,1,MPI_INT,j,WORKTAG,MPI_COMM_WORLD);
          MPI_Ssend(&ranked[index],1,MPI_INT,j,WORKTAG,MPI_COMM_WORLD);
     }
     present=1;
   }
}
if(present==0){
    MPI_Recv(&index,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
    MPI_Recv(&ranked[index],1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,
&status);
}
do{
  present=0;
  for(int i=start;i<2*chunk_size;i+=2){
   if(sub_arr[i]==ranked[index]){
     ranked[++index]=sub_arr[i-1];
     sub_arr[i-1]=sub_arr[start-1];
     sub_arr[i]=sub_arr[start];
     start+=2;
     index_prev=index;
     for(int j=1;j<num_procs;j++){
          MPI_Ssend(&index,1,MPI_INT,j,WORKTAG,MPI_COMM_WORLD);
          MPI_Ssend(&ranked[index],1,MPI_INT,j,WORKTAG,MPI_COMM_WORLD);
     }
     present=1;
   }
  }
  if(present==0){
      MPI_Recv(&index,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,
&status);
      MPI_Recv(&ranked[index],1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,
&status);
  }
}while(index!=(size-1));
```

CUDA 的核函数代码如下：

```
__global__ void ListRank
(int* List, int size)
{
      int block=(blockIdx.y*gridDim.x)+blockIdx.x;
      int index=block*blockDim.x+threadIdx.x;

      if (index<size){
            while (1){
                  int node=LIST[index];
                  if (node>>32==-1) return;
                  __syncthreads();

                  int mask=0xFFFFFFFF;
                  ing temp=0;
                  int next=LIST[node>>32];

                  if (node>>32==-1) return;

                  temp=node&mask;
                  temp+=next&mask;
                  temp+=(next>>32)<<32;

                  __syncthreads();
                  LIST[index]=temp;
            }
      }
}
```

3.4 Helman and JáJá 算法

Helman and JáJá 算法是一种为 SMP 硬件设计的算法，运行的例子见下。为了更好的利用 GPU，可以将这种算法改成递归 Helman and JáJá 算法（Recursive Helman JáJá Algorithm，RHJ），两种算法的伪代码如下：

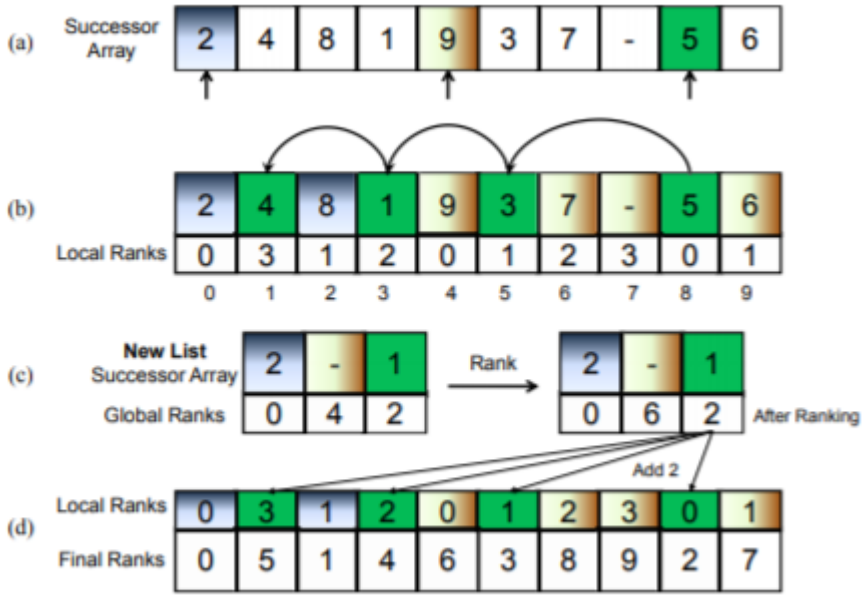


Figure 6 Helman and JáJá 算法的一个例子。这种算法比较适合在大型计算机上运行（Bader 的文章中是放在 Cray MTA-2，d Sun enterprise servers 和 IBM Cell processor 上运行的[6][7][8]）。这类机器的架构和 GPU 不太一样，所以如果要把 Helman and JáJá算法改成 CUDA 版本的运行在 GPU 上，应该要对它做一些改进。

Algorithm 2 Helman and JáJá List Ranking Algorithm

Input: An array L, containing input list. Each element of L has two fields *rank* and *successor*, and $n = |L|$

Output: Array R with ranks of each element of the list with respect to the head of the list

1: Partition L into $\frac{n}{p}$ sublists by choosing a splitter at regular indices separated by p.
2: Processor p_i traverses each sublist, computing the local (with respect to the start of the sublist) ranks of each element and store in $R[i]$.
3: Rank the array of sublists S sequentially on processor p_0
4: Use each processor to add $\frac{n}{p}$ elements with their corresponding splitter prefix in S and store the final rank in $R[i]$

Figure 7 Helman and JáJá 算法

Algorithm 3 Recursive Helman-JáJá **RHJ**(L,R,n,$limit$)

Input: Array L, containing the successors of each list element and $n = |L|$
Output: Array R containing the rank of each element in L

1: **if** $n <= limit$ **then**
2: Rank the final list L sequentially
3: **else**
4: Choose p splitters at intervals of $\frac{n}{p}$
5: Define the sublists on L,
6: Copy these splitter elements to L_1
7: Perform local ranking on each sublist of L
8: Save the local ranks of each element of L in R.
9: Set successor pointers for L_1
10: Write sublist lengths in R_1
11: Call Procedure **RHJ**(L_1,R_1,p,$limit$)
12: **end if**
13: Each element in R adds the rank of its sublist from R_1 to its local rank.

Figure 8 Recursive Helman JáJá Algorithm

4 实验过程（screenshot）

4.1 如何生成链表和怎么使用它们

实验输入数据由老师给出的两个函数生成。第一个函数生成的链表中的 N 个元素是从小到大依次的，第二个函数生成的链表中的 N 的元素是打乱顺序的。

容易看出，第一个链表中各个元素的 rank 很容易计算，即为

$$\mathrm{rank[i]} = \mathrm{N} - 1 - \mathrm{i}$$

我可以用这个链表来初步判断我写的函数有没有问题。

第二个链表通过一系列 swap 打乱了元素的顺序，所以 rank 无法像第一个那样用一个统一的公式就可以表示出来了，所以我之后用它来测试不同版本代码的速度。

Figure 9 老师给的两个链表数据生成函数生成的链表数据

4.2 串行程序

运行串行程序，用一个简单的表示链表的后继数组 S 来测试正确性，发现串行的 Wyllie 算法得到的正确的结果。

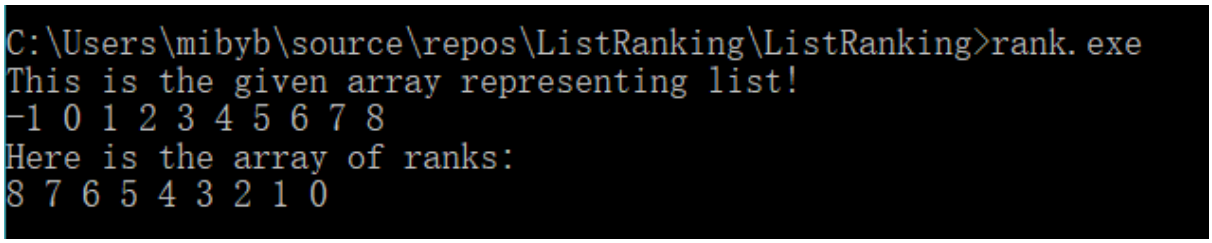
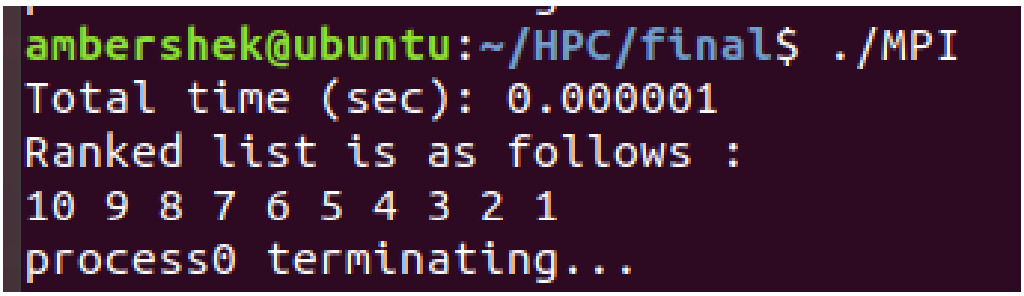


Figure 10 测试串行 Wyllie 算法的正确性。用一个长度为 10 的小链表来进行测试，可以看出结果是正确的。

4.3 MPI 并行程序

（之前 MPI 部分实验在 Ubuntu 虚拟机上完成，相关的东西装在那台机器上，所以这部分实验是在 Ubuntu 系统下完成的）



4.4 CUDA 并行程序

```
C:\Users\mibyb\source\repos\cudatest\x64\Release>cudatest.exe ha=1920 wa=1920 hb=1920 wb=1920
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "GeForce GTX 1050" with compute capability 6.1

Computing result using CUDA Kernel...
done
Performance= 286.22 GFlop/s, Time= 49.457 msec, Size= 14155776000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performancemeasurements. Results may vary when GPU Boost is enabled.
```

5 实验数据分析与问题讨论

5.1 实验数据分析

因为 MPI 的运行时间与线程数 p 有关，对长度为 1M 的链表使用 Wyllie 算法，得到的各个运行用时如下。我得到的加速效果不好，MPI 程序的用时始终比串行要大，这可能和 Wyllie 算法本身原理有关，它不能很好地利用并行资源。

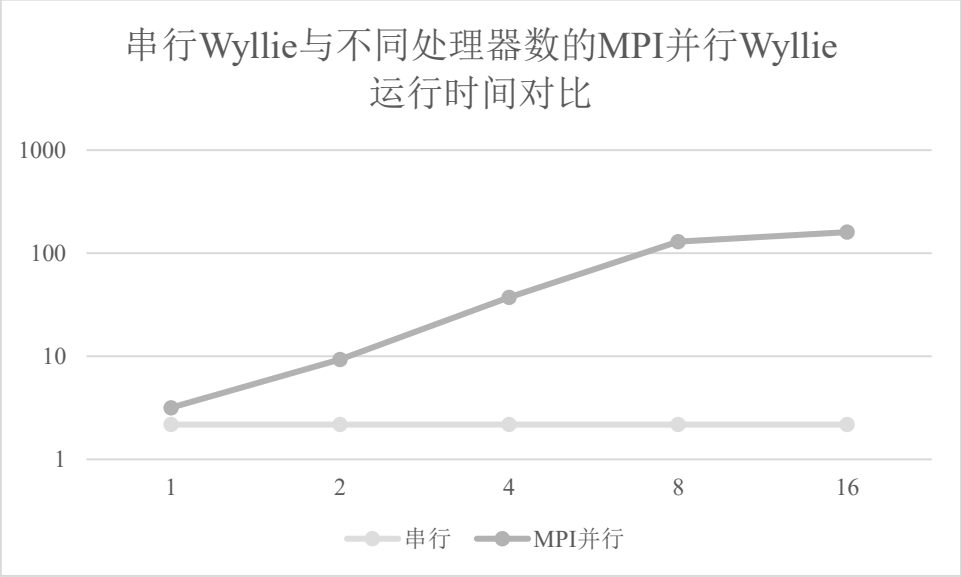


Figure 11 串行 VS MPI

对于不同规模的链表，可以看到对于小数据，串行的代码甚至比 CUDA 更快，但当数据逐渐增大到 1M 的过程中，CUDA 版本的速度超过的串行版本。

节点数目	1
核数	56
dimGrid	10
dimBlock	10000

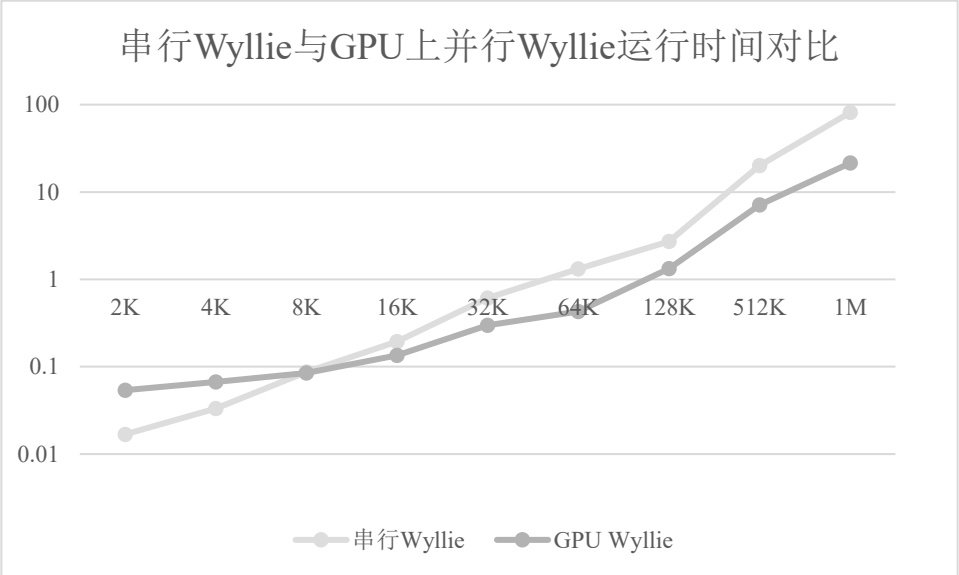


Figure 12 串行 VS CUDA

5.2 结果分析与总结

这次实验，我分别用串行、MPI 和 CUDA 三种方式实现了 Wyllie 算法。在此之前，我还尝试了 sequential scan 算法，但因为算法复杂度不够低最后没有采取这种算法。之后我还学习了解了 Helman and JáJá算法和递归 Helman and JáJá算法，递归 Helman and JáJá算法似乎会比 Wyllie 算法更适合在 GPU 上运行，不过时间原因我还没有实现这种算法，无法做出比较。

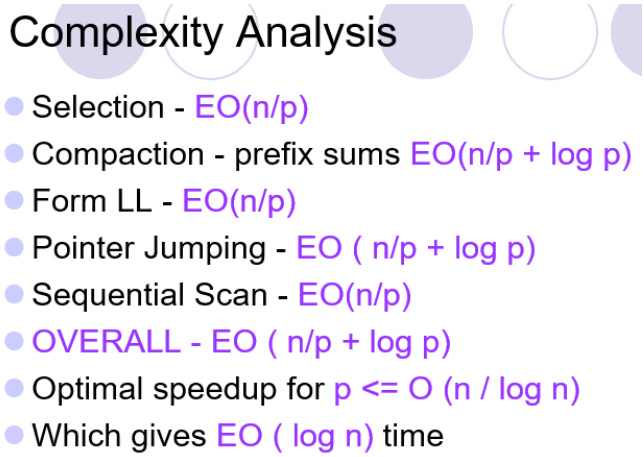


Figure 13 几种并行 List Ranking 算法的复杂度对比，我做了其中的 pointer jumping 和 sequential scan.

参考文献

[1] http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap30.htm

[2] https://en.wikipedia.org/wiki/Pointer_jumping

[3] http://www.cs.cmu.edu/~scandal/alg/listrank.html

[4] https://www.cs.cmu.edu/~glmiller/Publications/Papers/ReMiMo93.pdf

[5] http://cdn.iiit.ac.in/cdn/cstar.iiit.ac.in/~kkishore/ics152-rehman.pdf

[6] D. Bader, G. Cong, and J. Feo. On the Architectural Requirements for Efficient Execution of Graph Algorithms. International Conference on Parallel Processing (ICPP), 2005, pages 547–556, June 2005.

[7] D. A. Bader, V. Agarwal, and K. Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 1–10. IEEE, 2007.

[8] D. A. Bader and G. Cong. A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors (SMPs). Journal of Parallel and Distributed Computing, 65(9):994 – 1006, 2005.

[9] https://link.springer.com/content/pdf/10.1007%2FBFb0056600.pdf