Name: Jiaqiang Ruan; ID: jruan

1 What approaches did you take to parallelize the different parts of the program?

The final result achieve 64 points.

1. **start up**: I allocate an integer and a boolean array with size of node to record the index of the hub node and indicate whether the current node is a hub node or normal node.

2. **compute weights**: I first parallel on the hub nodes using the indexes of hub in the integer array I calculate in advanced and then I parallel on the normal nodes. When paralleling on the normal nodes, in order to utilize the block structure of the image, I split the image into different block and make sure each thread will iterate on one block. By experiment the block with square shape yields good performance. no wait"command is utilized.

3. **compute sums**: similar to compute weights. I first parallel on hub nodes then normal nodes. The block of the image is a vertical long line. This is because in compute sums, the program needs to write back values to to two double arrays. The value is store in row major, so a long-shape block is good for lowering the cache miss rate. Also, dynamic schedule"is utilized. This help balance the work load for each threads. Because some threads will start earlier than the others. Keep the size of work slightly small can help the threads evenly do the work.

4. **find moves**: I first parallel on rats in the batch, using ätomic"command to make sure the update of the delta countto be correct. The dynamic"command is utilize again to chop the work to small pieces and also into a cache friendly size, like 64. Then I parallel on the nodes to update the rat countfor the graph.

2 How did you avoid wasting time using synchronization constructs?

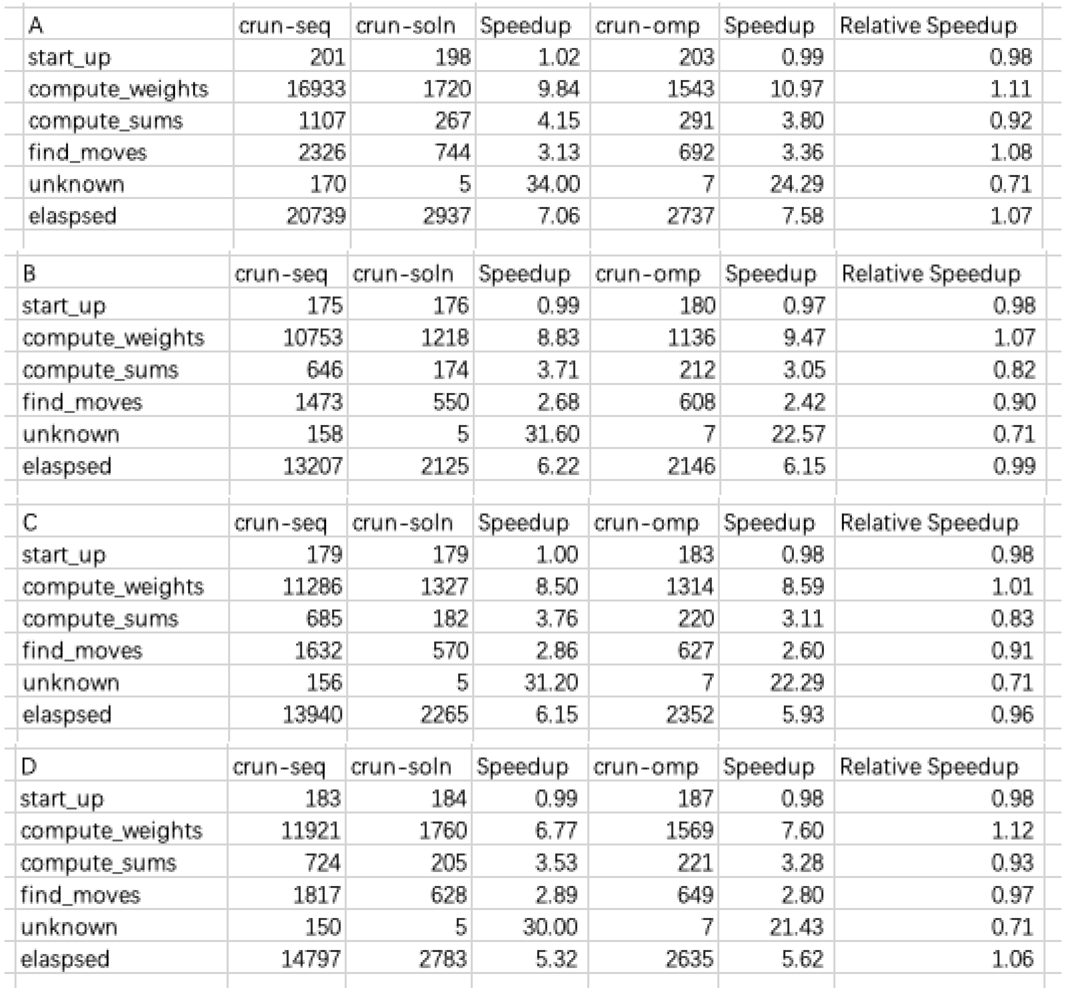
1. In the parallelism for hub nodes and normal nodes, I use nowait"command to break the implicit synchronization of ömp for", so that the thread can start calculating for normal nodes without waiting for other threads to finish calculating for hub nodes.

2. In the find movespart, the original code need to iterate on rat one by one to make sure the update for the delta count"is correct. Here I use parallel with atomic construct to avoid this iteration. In order to alleviate the imbalance of workload because of this contention, I use dynamic schedule to chop the work into smaller size.

2.1 How successful were you in getting speedups from different parts of the program? (These should be backed by experimental measurements.)

Total performance.

```
Name     steps    threads  secs    NPM     BNPM    Ratio   Pts
uniA     50       12       2.75    54.53   55.58   1.019   16
uniB     50       12       2.15    42.72   42.32   0.991   16
fracC    50       12       2.36    46.83   44.87   0.958   16
fracD    50       12       2.60    51.67   48.54   0.939   16
AVG:                               48.94   47.83
TOTAL:                                                     64
Test time for 12 threads = 61.93 secs.
```



A	crun-seq	crun-soln	Speedup	crun-omp	Speedup	Relative Speedup
start_up	201	198	1.02	203	0.99	0.98
compute_weights	16933	1720	9.84	1543	10.97	1.11
compute_sums	1107	267	4.15	291	3.80	0.92
find_moves	2326	744	3.13	692	3.36	1.08
unknown	170	5	34.00	7	24.29	0.71
elaspsed	20739	2937	7.06	2737	7.58	1.07

B	crun-seq	crun-soln	Speedup	crun-omp	Speedup	Relative Speedup
start_up	175	176	0.99	180	0.97	0.98
compute_weights	10753	1218	8.83	1136	9.47	1.07
compute_sums	646	174	3.71	212	3.05	0.82
find_moves	1473	550	2.68	608	2.42	0.90
unknown	158	5	31.60	7	22.57	0.71
elaspsed	13207	2125	6.22	2146	6.15	0.99

C	crun-seq	crun-soln	Speedup	crun-omp	Speedup	Relative Speedup
start_up	179	179	1.00	183	0.98	0.98
compute_weights	11286	1327	8.50	1314	8.59	1.01
compute_sums	685	182	3.76	220	3.11	0.83
find_moves	1632	570	2.86	627	2.60	0.91
unknown	156	5	31.20	7	22.29	0.71
elaspsed	13940	2265	6.15	2352	5.93	0.96

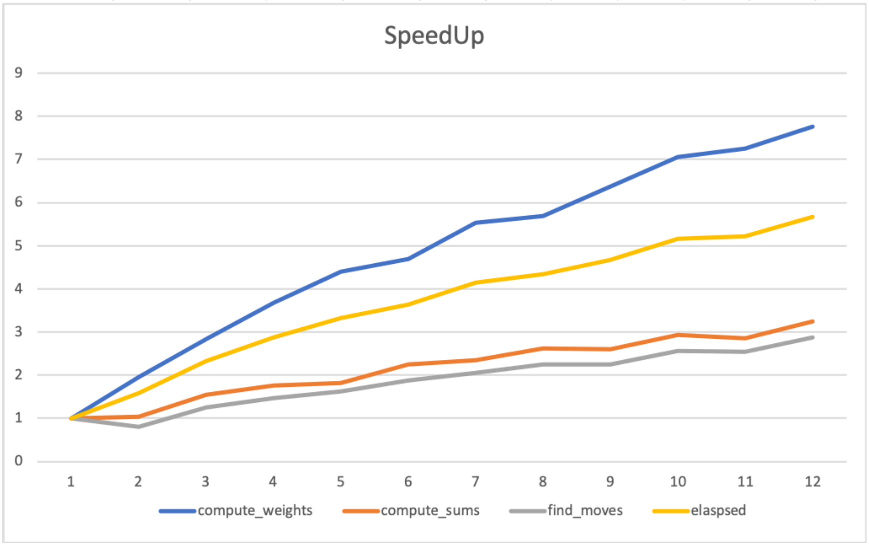
D	crun-seq	crun-soln	Speedup	crun-omp	Speedup	Relative Speedup
start_up	183	184	0.99	187	0.98	0.98
compute_weights	11921	1760	6.77	1569	7.60	1.12
compute_sums	724	205	3.53	221	3.28	0.93
find_moves	1817	628	2.89	649	2.80	0.97
unknown	150	5	30.00	7	21.43	0.71
elaspsed	14797	2783	5.32	2635	5.62	1.06

Figur 1: Result

3 How does the performance scale as you go from 1 (running crun-seq) to 12 threads? What were the relative speedups of different parts of your program?

Take graph D as example. Make other setting the same. As we can see the relation betweeen speedup and thread is nearly linear.

D	1	2	3	4	5	6	7	8	9	10	11	12
start_up	204	191	191	189	189	189	190	189	189	190	189	189
compute_weights	11967	6088	4221	3255	2719	2552	2163	2105	1877	1694	1649	1544
compute_sums	728	701	473	415	401	323	309	278	279	249	255	224
find_moves	1826	2280	1449	1251	1119	970	889	812	809	715	718	635
unknown	6	6	6	6	6	6	6	6	6	7	7	8
elaspsed	14734	9269	6341	5119	4436	4043	3560	3392	3160	2857	2819	2600



Figur 2: SpeedUp vs. Thread

4 How did the graph structure and the initial rat positions affect your ability to exploit parallelism? What steps did you take to work around any performance limitations?

1. **graph structure**: graph A and B are relatively regular. So when I am changing the block shape of the parallelism on the node. The result does not change much. But when I change block shape for graph C and D. The result varies a lot. My parallelism on the nodes are separated into hub part and normal part. For the normal nodes, the calculation needs the information from its four direct neighbor and also its hub. So if one thread happened to process a region in the graph, it can keep the hub information in the cache all the time. However, if a processor keeps calculating for nodes from different region, the information of the hub (that cache line) need to be load each time and it slows the process.

my solution: carefully adjust the block size, dynamic schedule size and use nowait command to avoid the potential cache miss and balance the workload.

2. **Rat position**: When paralleling on the nodes, the rat position does not make a difference. But when program iterates on rats and try to update the position of rat and update the number of the rat on a node, the rat position has influence on the speed. In my implementation, the rat position is updated by parallel on the rat and use the ätomic"command to make sure the delta countärray is updated correctly. So, when the rat are very concentrated, it is more to

likely to cause contention on the delta countärray. When the rat are more sparse on the graph, the contention is less and the parallelism is faster.

my solution: use dynamic schedule to chop the rat batch count into small pieces, so that when one thread finish earlier without much contention, it can start the next small piece. Also, I make the number of size to be 64, a number might be better for the cache performance.

5 Were there any techniques that you tried but found ineffective?

1. Use a $bcount * nnode$ size of buffer to store the update of the rat position, so that I do not need to use atomic command. The aggregation of bount for each node is proven to be not efficient for the program.

2. Same idea to use a buffer to store the result of "imbalance"value of the compute weight and maintain a boolean array to indicate when the value needs to be updated and skip those that does not need to update. This is not efficient because after the move, almost all the node need to be update. There is no meaning to maintain the boolean array to indicate whether a node needs to update or not.

3. Create an index array, where it store all the index of the node. The first part is hub node's indexes, the second half is the normal nodes. Then I can change the order of those normal node in the array. When parallel on the node, I will parallel on the hub nodes first, then one the other part. The idea of iterating on normal nodes in a specific way is good, but in this way, it's hard to control which thread calculates exactly for what kind of nodes. It is better to make sure each thread run on a specific block.