Saturday, 29 October 2016

MPCS 51100 HW2 Report

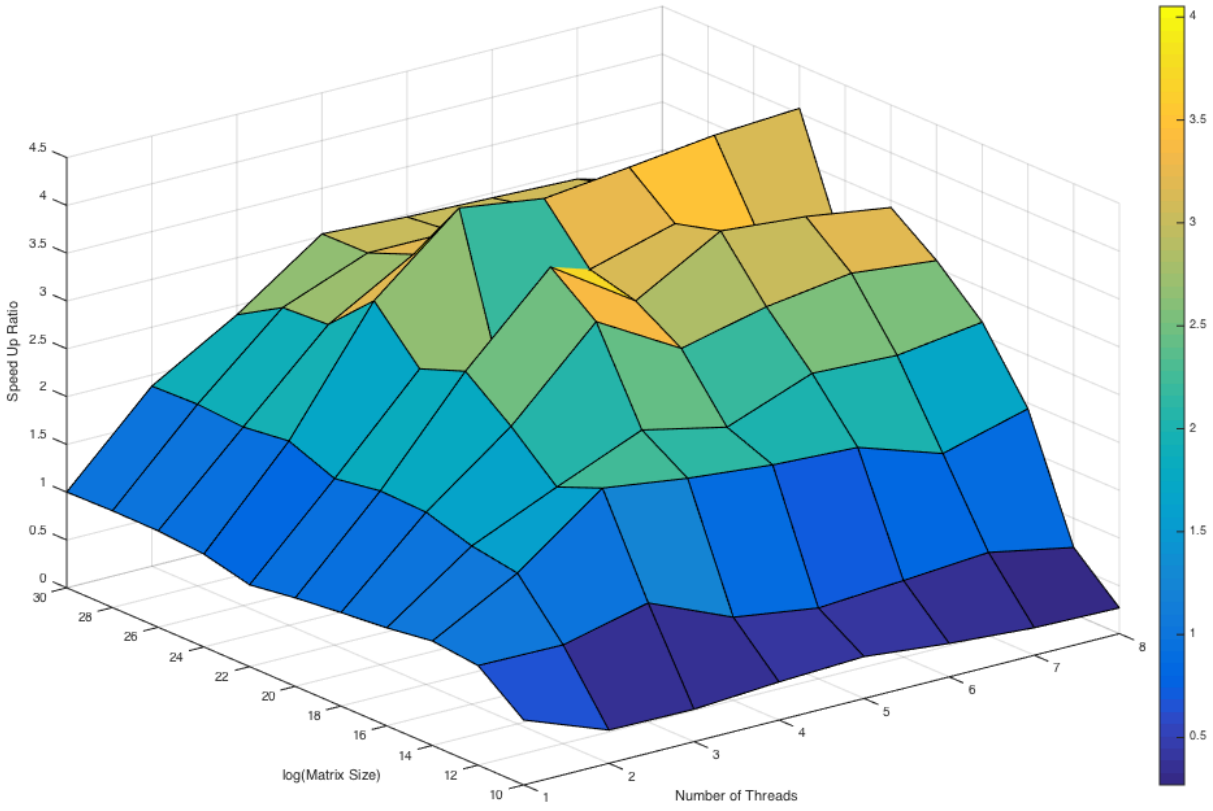
Yan Cui

- The runtime is calculated on my PC, here's the information about CPU and Memory
  - Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz
  - Physical CPU: 4, Logical CPU: 8
  - Memory 16GB 1600MHz DDR3

P1

I compared the performance of serial version and OpenMP version in different size of the problem and threads.

Speed Up Ratios of OpenMP in Different Size of Problem and Threads



Run Time of Serial Version and OpenMP Version

Size		Serial	1	2	3	4	5	6	7	8
32	RunTime	0.00001	**0.000015**	0.000028	0.000029	0.000025	0.000023	0.000028	0.000034	0.000037
	Speed up	1	**0.678082**	0.347368	0.339041	0.399194	0.438053	0.349823	0.287791	0.269755
64	RunTime	0.000032	0.000031	0.000031	**0.000026**	0.000037	0.000043	0.00004	0.000037	0.000046
	Speed up	1	1.042345	1.032258	**1.245136**	0.869565	0.740741	0.802005	0.869565	0.695652
128	RunTime	0.00013	0.000119	0.000083	**0.000058**	0.000062	0.000064	0.000066	0.000077	0.000067
	Speed up	1	1.093043	1.576784	**2.24055**	2.116883	2.031153	1.987805	1.697917	1.940476
256	RunTime	0.000444	0.000434	0.000268	0.000217	0.000184	0.0002	0.000174	0.000176	**0.000168**
	Speed up	1	1.024205	1.655365	2.04558	2.415987	2.219281	2.556387	2.518707	**2.639929**
512	RunTime	0.001732	0.001772	0.000962	0.0007	**0.000518**	0.000611	0.000569	0.000546	0.000563
	Speed up	1	0.977874	1.801497	2.47394	**3.343304**	2.837373	3.046957	3.172496	3.077811
1024	RunTime	0.007201	0.007781	0.00396	0.002532	**0.001941**	0.002302	0.001982	0.002024	0.002102
	Speed up	1	0.925438	1.818301	2.844076	**3.709282**	3.1281	3.634033	3.558257	3.426695
2048	RunTime	0.028531	0.033344	0.016388	0.010713	0.013085	0.008773	**0.008142**	0.009093	0.010478
	Speed up	1	0.855655	1.740919	2.663235	2.180479	3.251918	**3.504268**	3.137621	2.722993
4096	RunTime	0.131199	0.13509	0.06805	0.041391	**0.033519**	0.034642	0.033749	0.032826	0.032377
	Speed up	1	0.971198	1.927961	3.169717	**3.914136**	3.78724	3.887496	3.996768	4.052182
8192	RunTime	0.450475	0.446747	0.241171	0.165976	**0.139809**	0.145973	0.150929	0.151602	0.154463
	Speed up	1	1.008343	1.867867	2.714101	**3.222069**	3.086017	2.984683	2.971433	2.916388
16384	RunTime	1.903057	1.881285	1.001451	0.708608	0.627772	0.627988	0.622149	0.620905	**0.62164**
	Speed up	1	1.011573	1.9003	2.685628	3.031446	3.030406	3.058845	3.064971	**3.061351**
32768	RunTime	7.301317	7.289987	3.875762	3.037428	**2.414342**	2.454091	2.476095	2.500583	2.726401
	Speed up	1	1.001554	1.88384	2.403782	**3.024143**	2.975161	2.948723	2.919846	2.678005

The table shows that when the size of the matrix is small, the performance of OpenMP version is the same as the serial version or even worse, since scheduling and synchronizing among different thread may takes up a lot of runtime.

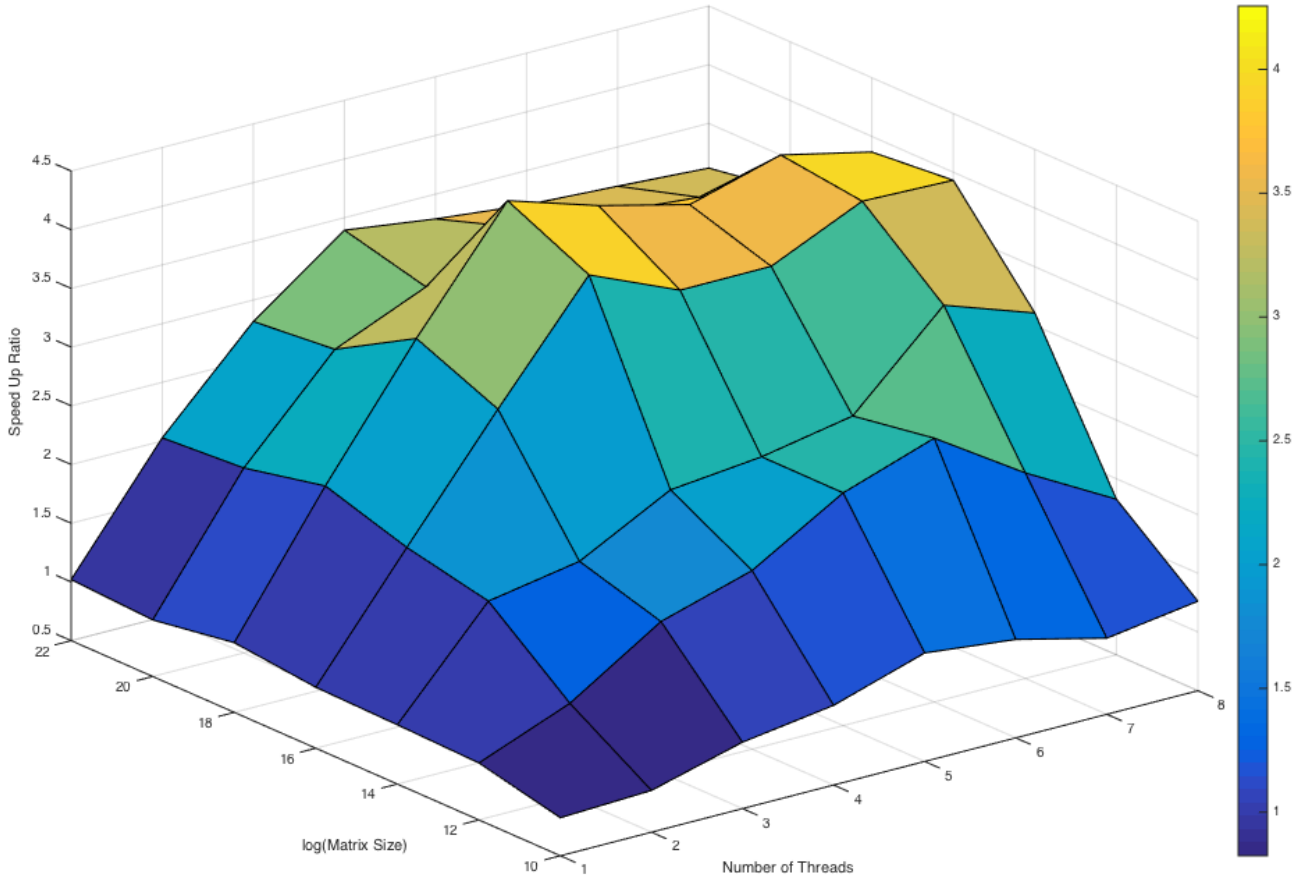
When the size of the matrix is large enough so that the time spending on scheduling and synchronizing could be ignored, the speed up ration is close to the

number of threads we use. But if the number of threads exceed the number of the processors, it will not speed up the program any more since some tasks are not parallel any more.

- **P2**

First I modified the order of the loop so that the product of two matrices can be computed in a more cache friendly way. Then I compared the performance of serial version and OpenMP in the same way as P1.

Speed Up Ratios of OpenMP in Different Size of Problem and Threads



Run Time of Serial Version and OpenMP

Size		Serial	Optimal	1	2	3	4	5	6	7	8
32	RunTime:	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	**0.0001**	0.0001	0.0002	0.0001
	Speed up:	1	1	0.8241	0.8558	1.0659	1.1788	**1.424**	1.3383	1.1484	1.2624
64	RunTime:	0.0012	0.0011	0.0011	0.0009	0.0006	0.0006	0.0005	**0.0004**	0.0005	0.0006
	Speed up:	0.9167	1	0.9852	1.2922	1.7883	2.0214	2.4825	**2.7476**	2.2505	1.8258
128	RunTime:	0.0098	0.008	0.008	0.0043	0.004	0.0033	0.0032	0.003	**0.0024**	0.0026
	Speed up:	0.8163	1	1.0049	1.8574	1.9953	2.4037	2.4842	2.6311	**3.3736**	3.1067
256	RunTime:	0.0845	0.0657	0.0635	0.0322	0.0217	0.0165	0.018	0.018	**0.0164**	0.0165
	Speed up:	0.7775	1	1.019	2.0091	2.9863	3.9288	3.5983	3.6005	**3.9545**	3.9319
512	RunTime:	0.9197	0.5693	0.5205	0.2555	0.1734	**0.1338**	0.1419	0.149	0.1408	0.1472
	Speed up:	0.6190	1	1.0936	2.2278	3.2833	**4.2561**	4.013	3.821	4.0427	3.8672
1024	RunTime:	14.253	4.2349	4.231311	1.996941	1.374464	**1.135677**	1.178976	1.18812	1.236751	1.249904
	Speed up:	0.297	1	0.9603	2.0348	2.9563	**3.5779**	3.4465	3.42	3.2855	3.2509
2048	RunTime:	117.67	34.25	33.563	16.8997	12.1587	**10.092**	10.4141	10.6777	10.8181	10.9798
	Speed up:	0.291	1	1.0205	2.0268	2.817	**3.3939**	3.2889	3.2077	3.1661	3.1195

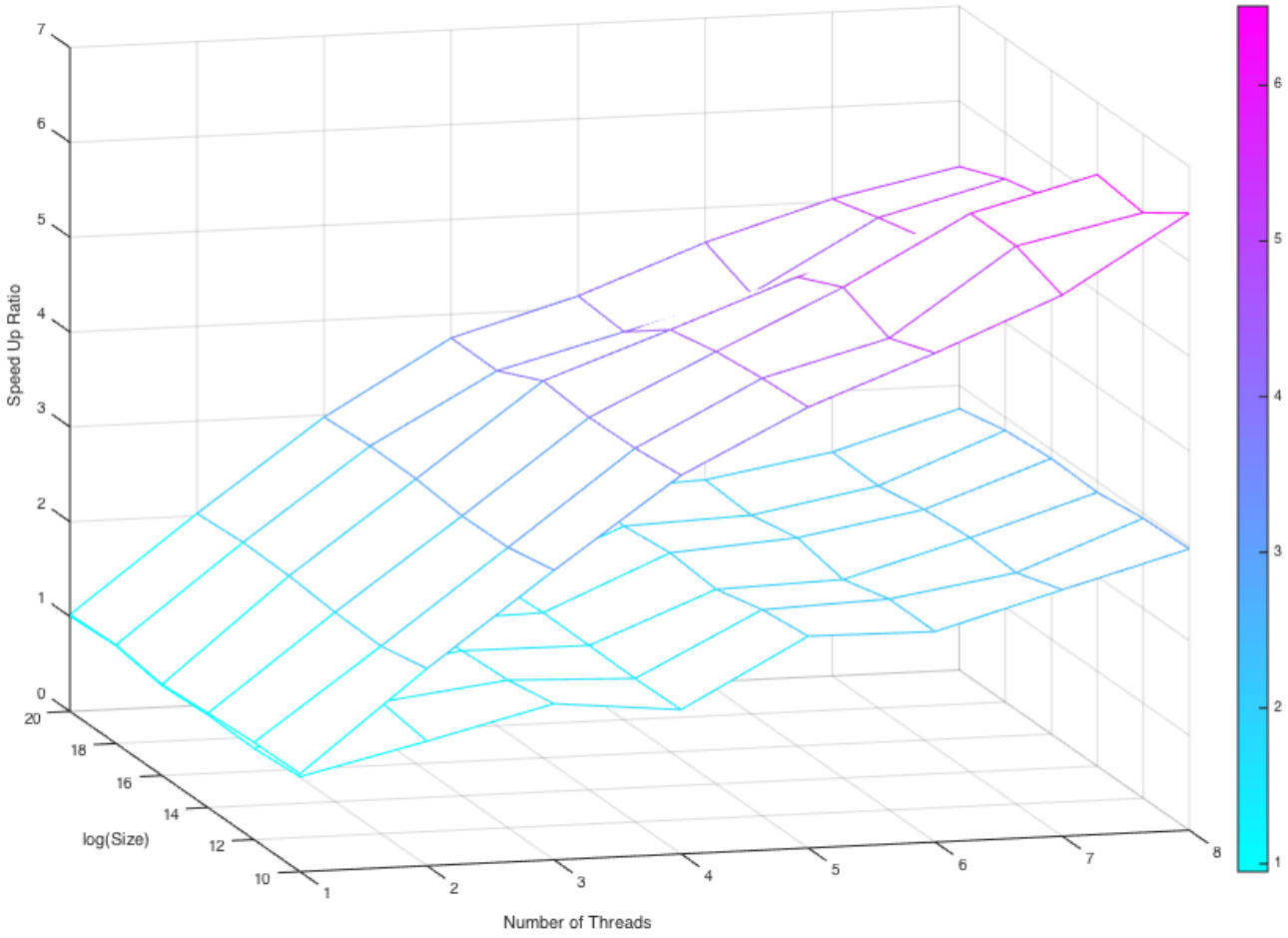
The result is similar to P1. When the size of the matrix is large, the speed up ration is close to the number of threads we use but can not exceed the number of processors.

I also notice that although my computer has 8 logical processors the speed up ratio is no more than 4 even though I use 8 threads. I think the reason is that the computation in each literation is quite simple in P1 and P2, so writing the answer back which should be synchronized among different cores becomes the bottleneck here. I only have 4 physical CPU core, 2 logical core can't write data to a shared variable at the same time. So if writing becomes a bottleneck, the speed-up ratio may be limited.

- **P3**

I compared the performance of serial version and OpenMP version using dynamic scheduling and OpenMP version using static scheduling in the same way as P1.

Speed Up ratio for Dynamic Version(Upper Surface) and Static Version(Lower Surface)



Run Time of Serial Version and OpenMP(Static, Dynamic)

Size	Schedule		Serial	1	2	3	4	5	6	7	8
64	Static	RunTime	0.046213	0.04611	0.035167	0.028077	0.030385	0.020687	0.020824	0.017785	**0.015578**
		Speed up	1	1.0022	1.3141	1.6459	1.5209	2.2339	2.2192	2.5984	**2.9666**
	Dynamic	RunTime	/	0.044783	0.022226	0.015134	0.011575	0.00994	0.008962	0.008099	**0.007101**
		Speed up	/	1.0319	2.0792	3.0536	3.9925	4.6492	5.1565	5.706	**6.508**
	Static	RunTime	0.175745	0.182823	0.126398	0.112835	0.116253	0.080797	0.078976	0.072025	**0.05954**
		Speed up	1	0.9613	1.3904	1.5575	1.5117	2.1751	2.2253	2.4401	**2.9517**

Size	Schedule		Serial	1	2	3	4	5	6	7	8
128	Dynamic	RunTime	/	0.171358	0.088867	0.059452	0.044598	0.03807	0.03532	0.029864	**0.028466**
		Speed up	/	1.0256	1.9776	2.9561	3.9406	4.6164	4.9758	5.8848	**6.1739**
258	Static	RunTime	0.693935	0.710149	0.512162	0.454059	0.455859	0.338251	0.331586	0.280878	**0.24039**
		Speed up	1	0.9772	1.3549	1.5283	1.5223	2.0515	2.0928	2.4706	**2.8867**
	Dynamic	RunTime	/	0.696876	0.347377	0.235165	0.176704	0.152232	0.134083	0.117796	**0.111249**
		Speed up	/	0.9958	1.9976	2.9508	3.9271	4.5584	5.1754	5.891	**6.2377**
512	Static	RunTime	2.84621	3.001664	2.065414	1.901707	1.861159	1.357053	1.297611	1.171664	**0.980012**
		Speed up	1	0.949	1.3792	1.4979	1.5306	2.0991	2.1953	2.4313	**2.9067**
	Dynamic	RunTime	/	2.964869	1.394101	0.948388	0.716643	0.640032	0.576116	0.537903	**0.505389**
		Speed up	/	0.9608	2.0433	3.0036	3.975	4.4507	4.9445	5.2958	**5.6365**
1024	Static	RunTime	45.300615	11.023356	8.042566	7.245931	7.584086	5.568964	5.428759	4.856097	**3.970731**
		Speed up	1	1.0318	1.4142	1.5697	1.4997	2.0424	2.0952	2.3422	**2.8645**
	Dynamic	RunTime	/	10.93237	5.526528	3.776287	3.038577	2.779811	2.566387	2.199757	**2.062141**
		Speed up	/	1.0404	2.0581	3.012	3.7432	4.0917	4.432	5.1706	**5.5157**
2048	Static	RunTime	45.300615	45.080411	32.507103	29.345927	29.22342	21.827186	21.266991	19.191851	**16.437739**
		Speed up	1	1.0049	1.3936	1.5437	1.5501	2.0754	2.1301	2.3604	**2.7559**
	Dynamic	RunTime	/	44.361899	22.369472	15.209202	12.079092	10.964884	9.777429	9.005666	**8.534474**
		Speed up	/	1.0212	2.0251	2.9785	3.7503	4.1314	4.6332	5.0302	**5.308**

P3 is different from P2 and P1 because it contains a while loop in each literation. The termination of the while loop is related to the pixel's location in the set so that the time of each literation is different. If we schedule the loop in a static way, which means that each thread gets an equal number of literation, it may be less efficient since some threads may get "harder" tasks and have to spend longer time while other threads may already finish their work. However, in dynamic way, each thread will get a task when it gets current work done, which means that all the threads will be working as long as there are tasks available.

From the table, we can find that the OpenMP version speed up the process of generating the Mandelbrot Set and the dynamic schedule version outperformance the static version significantly.

Compared with P1 and P2, the computation for each literation in P3 is much more complex, the time for writing the result back to the shared variable no longer

influence the runtime significantly. So I can have a speed up ratio bigger than 5 when using 8 threads.

The png can be find in folder p3, view.cpp is the code to generate the png file, opencv is need to compile it.

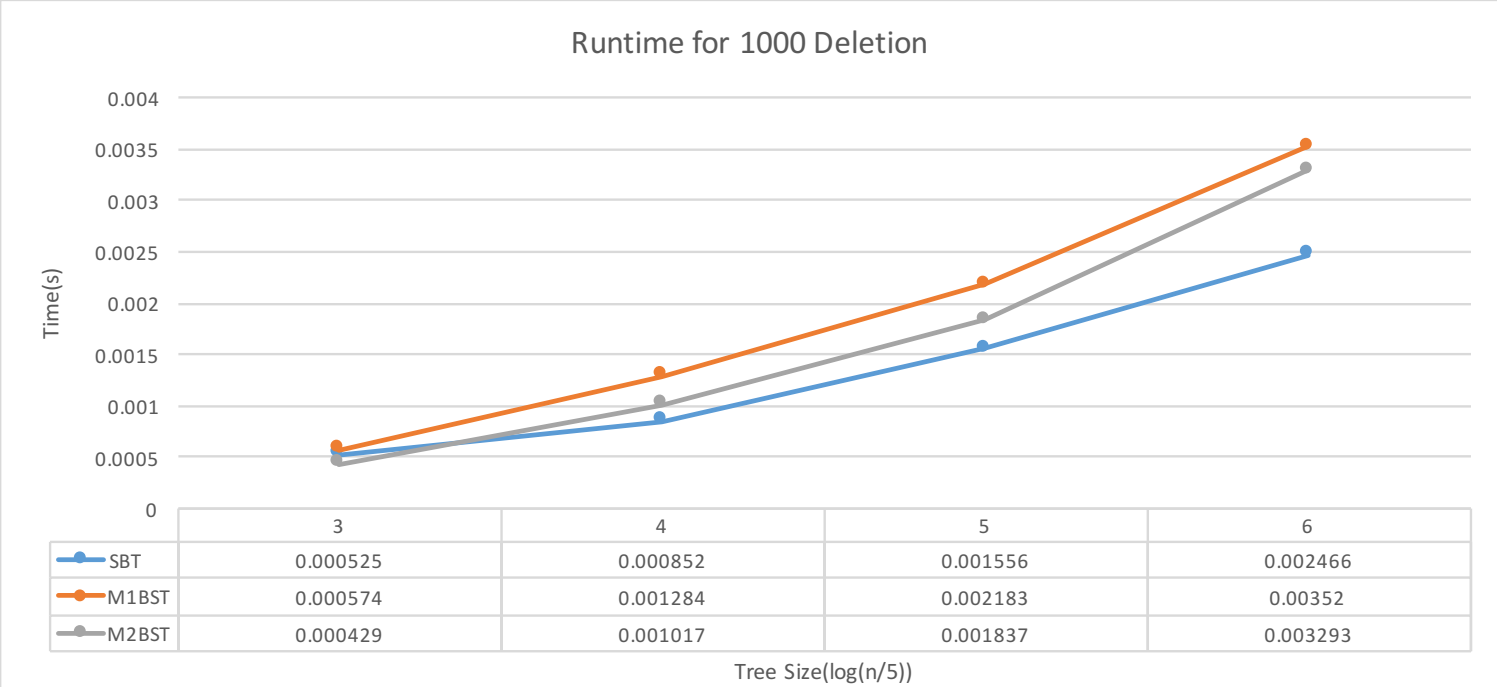
- **P4**

Using makefile to compile. m1_dic is the dictionary based on Model 1 version binary search tree, m2_dic is the dictionary based on Model 2 version binary search tree, sbt_dic is the dictionary based on a self balancing which called Size Balanced Tree(https://link.zhihu.com/?target=http%3A//www.stanford.edu/~cqf/SBT.zip). Operations like insert, look up, delete of this data struct is log(n) just like other self balancing binary search tree. It is easy to implement and has a quite good performance in practical since when it maintains the size, the depth of all the nodes is decreasing. This dictionary support these commands in terminal:

- add word “definition” : Add a word and associated definition(surrounded by quotation marks) to the dictionary. If the word is already in the dictionary, then it can’t be added again.
- delete word : Remove the word from dictionary, if the word is not found, this operation will fail.
- find word: Print the word and its definition if it is found.
- print: Show all the word and its definition in alphabetical order.
- random NUMBER: randomly generate strings and add them into the dictionary.
- height: print the max depth of the tree which store the keys and values
- quit: exit and free all the pointers

Then I tested the performance of these three types of tree using test.c. In including deleting nodes, searching nodes, insert nodes. Since the data is random, I run each test test for several times and use the average runtime to compare their performance.

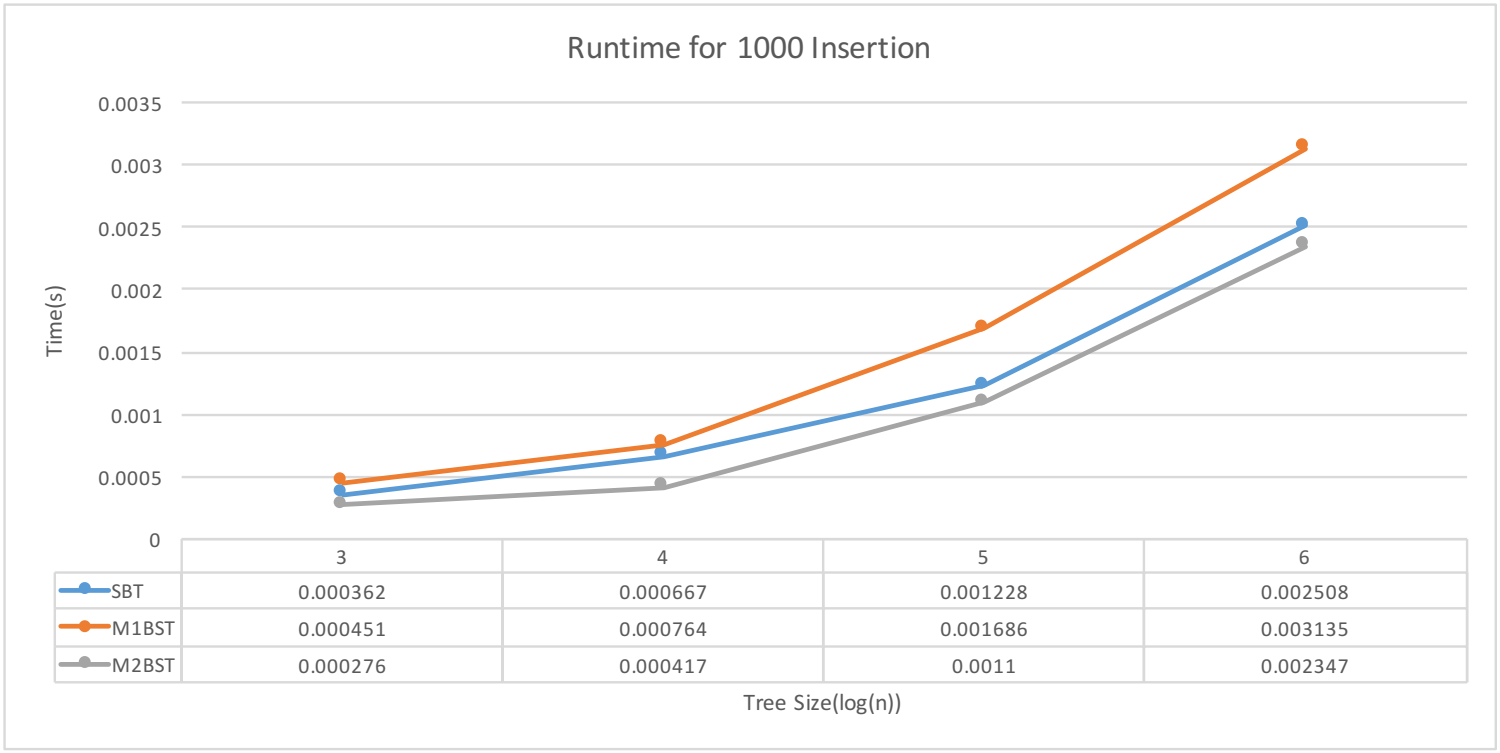
When testing the runtime for deletion, I delete 20% of the nodes in a tree with 5,000 to 5,000,000 nodes. The following table shows the result.



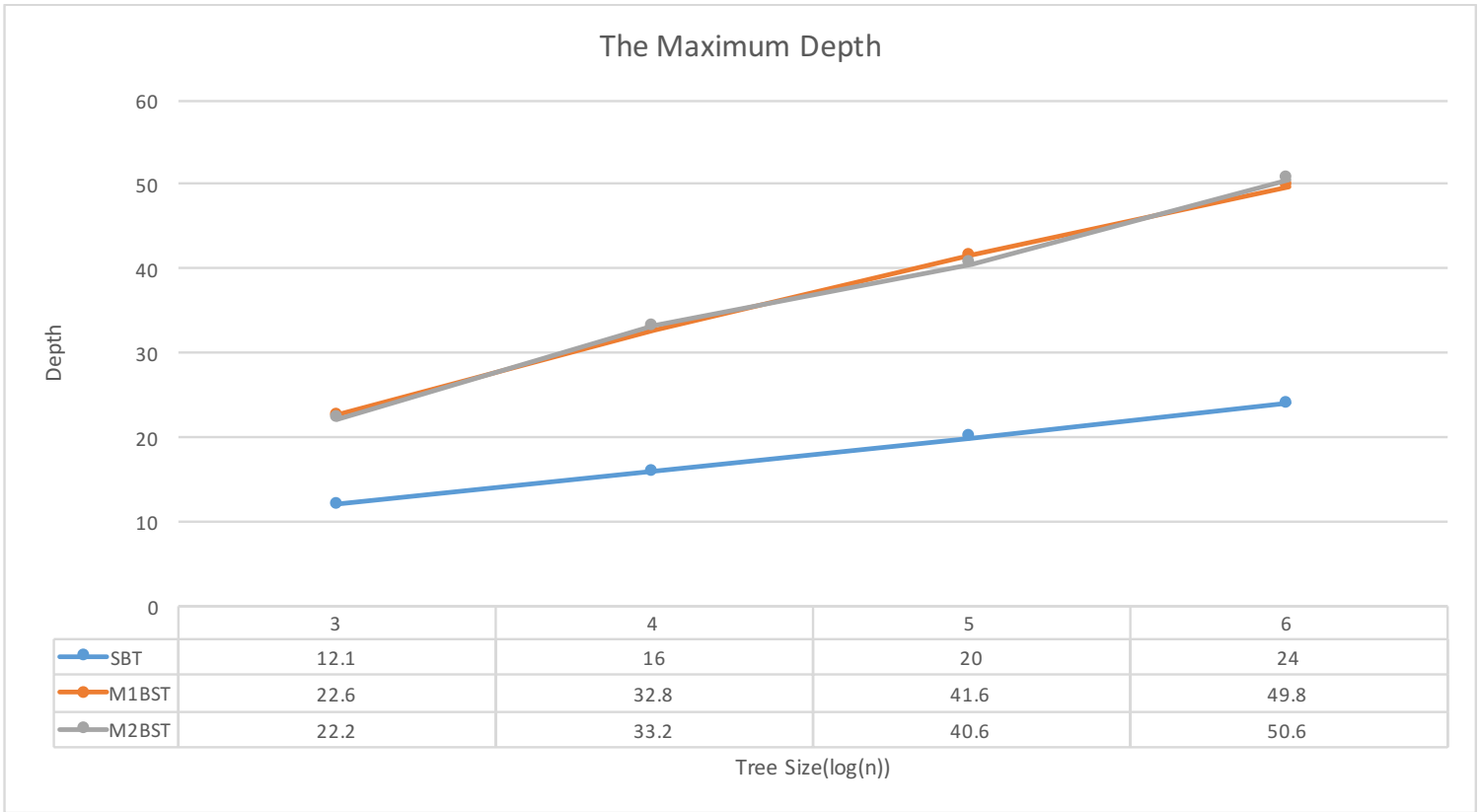
	3	4	5	6
SBT	0.000525	0.000852	0.001556	0.002466
M1BST	0.000574	0.001284	0.002183	0.00352
M2BST	0.000429	0.001017	0.001837	0.003293

Since I set log(treesize/5) as the x-axis and only delete 20% of nodes in a tree, if the deletion has a time complexity of O(log(n)), the runtime should be a similar to line in this graph. We can find that SBT has the best performance when the tree is big and the time complexity of it is O(log(n)) according to its line.

All of three models make a copy of keys and values in my code instead of simply assign the pointers, so I don't need to worry about the time spending on creating a node in these model will be different. When testing the runtime for insertion, I randomly generate some words and insert them into an empty tree. The size of words is from 1,000 to 1,000,000. The following table shows the result. This test can not represent the time complexity of inserting a node since the size of the tree is increasing from 0 to n during the process. So I use this test to compare the efficiency of insertion between these tree models. I also test the tree's maximum depth of model after all the insertion is done. The depth should be in a straight is the tree is a balanced tree with a depth of O(log(n)). Here's the result:



	3	4	5	6
SBT	0.000362	0.000667	0.001228	0.002508
M1BST	0.000451	0.000764	0.001686	0.003135
M2BST	0.000276	0.000417	0.0011	0.002347

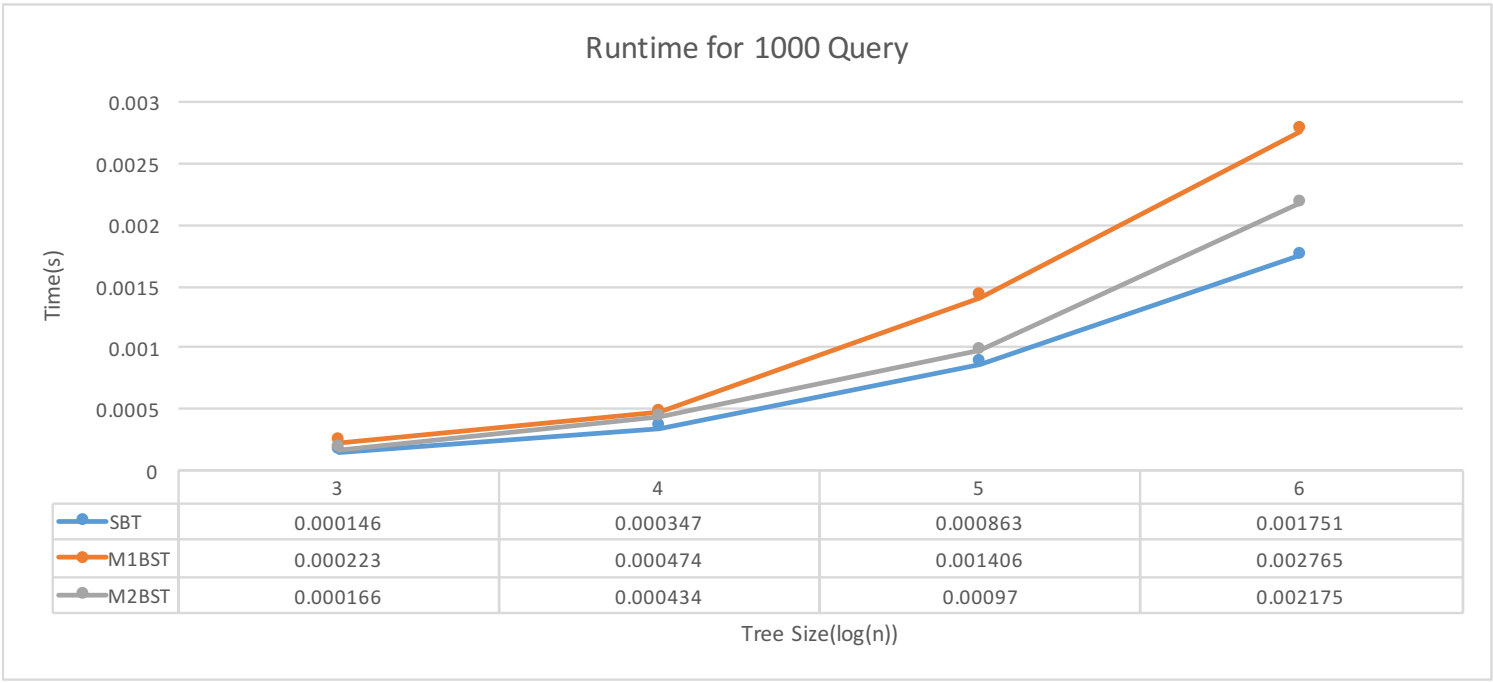


	3	4	5	6
SBT	12.1	16	20	24
M1BST	22.6	32.8	41.6	49.8
M2BST	22.2	33.2	40.6	50.6

We can find that Model 2 binary search tree has the fastest insertion since we just need to create an node and attach it to a node, but in Model 2 binary search tree and SBT we have to do something else to maintain the structure.

Both of these models have depth of O(log(n)) since we insert the key in a random instead of increasing or decreasing order. SBT have a significantly smaller depth than the unbalanced model.

When testing the runtime for searching, half of the query are designed to find the existing keys and half of them are designed to find the keys that not belongs to the tree. Here's the result:



	3	4	5	6
SBT	0.000146	0.000347	0.000863	0.001751
M1BST	0.000223	0.000474	0.001406	0.002765
M2BST	0.000166	0.000434	0.00097	0.002175

We can find that SBT has the best performance since it has a lower depth.

- **P5**

In the none recursive code, I use a stack to store the precursor of a node and check if its key is bigger than the precursor's key to determine if the BST is valid or not.

In the recursive code, I just check if the key of a node is within its upper and lower bound.

- **P6**

I test the runtime for building the table, actual size, and runtime for a large amount of random queries of each look up table in Figure 2 and Figure 3. Here's the result:

Runtime for building look up table in Figure 2: 1.188301

Memory for look up table in Figure 2: 364615KB.

Runtime for building look up table in Figure 3: 0.035524

Memory for look up table in Figure 3: 64945KB.

Performance for 10000000 random query:

Binary search in each array: 54.844914

Speed up with table in Figure 2: 4.230136

Speed up with table in Figure 3: 10.322949

The result shows that both struct can significantly speed up the searching process. The struct in Figure 3 is more efficient in space while the struct in Figure 2 is more efficient in time. The structure in Figure 2 is faster since we can get the index in every array of the target as soon as we located it in the whole list. But if we use the structure in Figure 3, we have to make 68 comparisons to get the exact index.

- **All the original result can be found in folder result as px_res.txt**