Introduction:
This project is about parallelizing the construction of optimal binary search trees. The optimal binary search tree is constructed using dynamic programming. I have attempted to parallelize the construction of the matrix using threads and shared memory.

Algorithm:
Let us look at the algorithm to construct OBST.
Suppose we 4 keys and we want to generate an optimal binary search tree using these keys. Let the probabilities of the keys be {0.2, 0.3, 0.1, 0.4}

Now to generate the optimal binary search tree first we generate binary trees considering keys (1,2),(2,3),(3,4) i.e. create binary treas considering two nodes.

In the next step, we create binary trees using three keys i.e one tree will be constructed using trees (1,2,3) and another using (2,3,4). For constructing these trees we use the cost of the trees built in the previous step. In the next step, we generate the entire binary search tree.

We create a matrix to help us construct the optimal binary search tree.

0.2	?	?	?
0	0.3	?	?
0	0	0.1	?
0	0	0	0.4
0	0	0	0

Parallelization:
If we look closely at the algorithm we can see that the cost computation of binary trees of the same size can be done in parallel. That is the cost of binary trees with size 2 can be generated in parallel, then size 3 trees can be generated in parallel. This means that to generate an OBST we can calculate C(1,2),C(2,3), and C(3,4) in parallel. But we would need to wait for the threads to complete the calculation of optimal cost for size=2 before moving to size = 3.

The reason for this is we need the cost of the trees with size 2 to generate any tree with size 3. To ensure that all threads wait before calculating the cost of the next tree size a barrier is used which will make the threads wait until all the threads reach the barrier. We can also see that as we are constructing binary trees with bigger sizes, we need to perform a greater number of computations. For example, when computing the cost of binary trees of size=2, for each binary tree we perform two computations and find the minimum. When computing the cost of binary trees of size=3, for each binary tree we need to perform three computations.

Parallel Approach:

Case 1:
N==P
In this case, we have processors equal to the number of nodes. For this case, my approach will generate P threads and each thread will work on one row of the matrix.

Thread1	0.2	?	?	?
Thread2	0	0.3	?	?
Thread3	0	0	0.1	?
Thread4	0	0	0	0.4
	0	0	0	0

The thread which works on the first row will take the maximum time since it needs to do the maximum number of computations. For n=4 it needs to compute the trees of size 2,3,4. For n=5 it needs to compute trees of size 2,3,4,5. Hence the complexity of the first thread will be O(n^2). When performed serially the complexity is O(n^3). In both cases, I am storing the sum of probabilities in an array and taking constant time to calculate the sum.

Speedup = Ts/Tp = $n^3 / n^2 = n$

Case 2:
N>>P
In this case, each thread will compute the values for N/P rows. Hence the time complexity for this program will be O(n^3 / P). Since we will have P threads running at a particular time.

Speedup = Ts/Tp = $n^3 * P / n^3$ = P

Thread1				
Thread2				
Thread3				
Thread4				
Thread1				
Thread2				
Thread3...				

This scheme does not cause load imbalance. One thing to notice here is that for generating the cost of a tree of a particular size the same amount of work is required. Hence when N>>P the most imbalance will be caused when the first thread is dealing with one row more than the other threads. As we proceed the number of rows to be calculated decreases hence there will be a point when the number of rows<P. In this case, some threads will be idle, but this case will also arrive for N=P.

Conclusion:
This project helped understand and implement a general method to parallelize programs which are based on dynamic programming. Shared memory seems to be a better option to implement this program since in message-passing there will be a lot of messages that need to be exchanged which will deteriorate the performance.