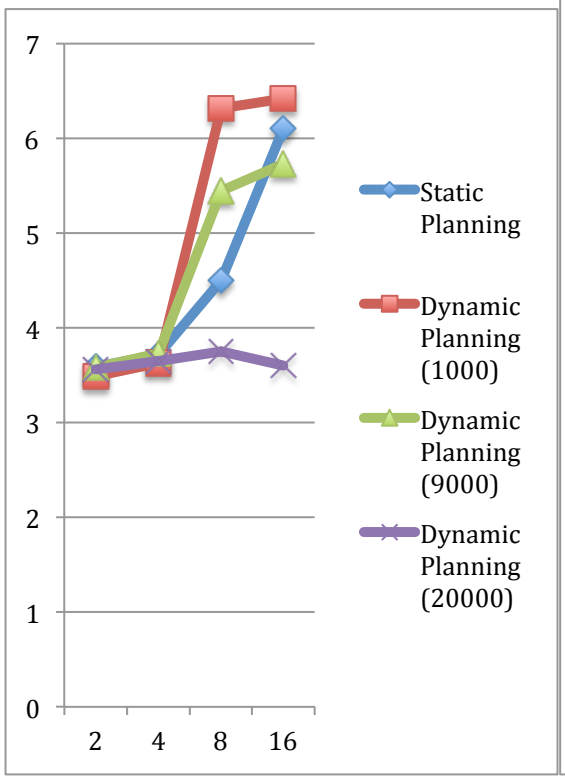
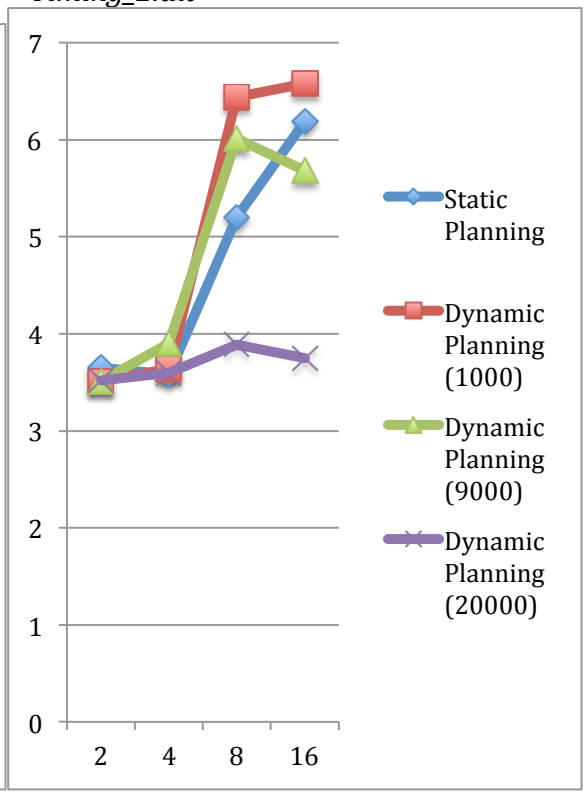
Ian Schweer **Project 1** **22514022**

Graphs for runtimes vs. thread count.

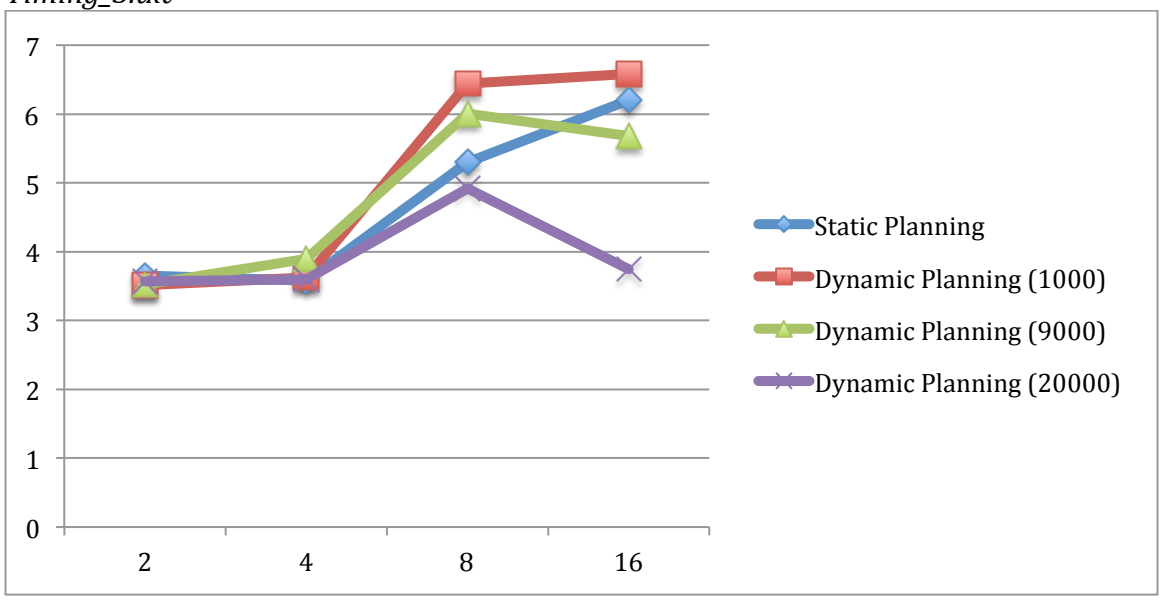
Timing_1.txt



Timing_2.txt



Timing_3.txt



Begin Analysis

Technique A:

Technique A is a upfront scheduling approach. We divide the work evenly ahead of time between threads. This approach works very well for files where each thread gets a large amount of work. As the number of threads increase, we can see a slowdown. This is explainable when considering: the time the task T takes to complete, and the operating system having to do context switches. Having to search each string for a palindrome takes at least O(N) time using Manchers algorithm. When each thread has to hold 1 / N amount of data, with 5MB total, context switching becomes an overhead. These two together explain why we see an increase of time occurs with more threads. **This is best used when a small amount of threads has a large amount of data.**

Technique B:

Technique B has us dividing work dynamically. A thread will "report" to get a new set of data to work on, and then return to get another set. This technique requires the use of locks. This is a huge overhead when working with small datasets. When data becomes larger and the thread numbers increase, we see a huge speed up, since each task has a small amount of work. **This is best used when a large amount of threads has a small amount of work.**

Chunk size:

The chunk size changes the "report amount". This caused a lot of threads to have no work because there was no work left to do. The problem is as the number of threads goes up, less efficient scheduling can happen, causing some threads to wait longer. **A medium sized chunk is best to ensure threads are working.**

Thread work:

There were some threads that did less work due to: The locks, the remaining amount of work being small. In application, you would want to optimize by reducing the amount of threads to ensure you always have a majority of your threads running.