Casey Brooks

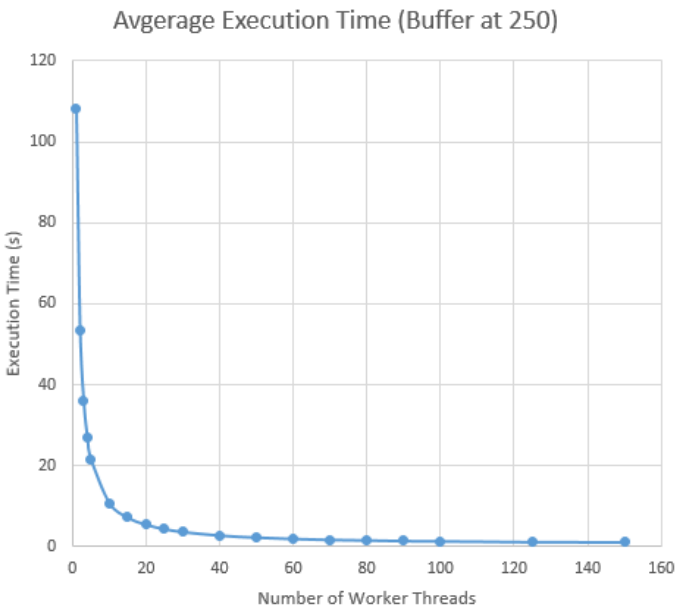
CSCE 313 - 504

MP3 Report

The purpose of this assignment was to give us experience in working with various aspects of a multithreaded environment, including: creation and management of POSIX pthreads, synchronization between threads using semaphores, implementation of semaphores using a POSIX mutex and condition variable, and analysis of performance using the these tools.

This program sets up a data server and makes calls to it, storing the results into a histogram specific to each person that is calling the server. To achieve this, each person has an associated request thread and statistics thread. The request thread dumps requests of the form "data <person name>" into a bounded buffer, and once it has deposited the required number of items it quits. Next, a number of worker threads continually take items from this buffer until the correct number has been retrieved. After taking each request out of the buffer, it sends the request to the server, which is able to return multiple requests concurrently, and because the request has the name of the caller, it then deposits the return value into another bounded buffer for that specific person. Each worker thread can take any data request and store it with any given person, and therefore, the worker threads are completely independent of the people. Any number of worker threads can work without issue, so long as the machine can allocate resources for them. A statistics thread for each person then consumes the replies sent by the server and deposited into their buffer, and these values are then stored into a histogram, which is then printed out.

A large part of this assignment was to determine performance increases gained by using multiple worker threads. If there is only one thread taking requests and serving them up, then it takes a long time to execute because the request channel to the server is slow. So as more threads are working together, the program speeds up significantly, to a point. Before I began testing, I had a hunch that there would be a point when the cost of scheduling the threads would overcome their individual performance increase, and while this at first seemed correct, it turns out to not be the case. The first machine I was testing on was a web-based IDE which serves many users and gave quite inconsistent execution times, varying by up to several seconds for the same test. Upon testing with compute.cse.tamu.edu, I found the results to be very consistent and, contrary to my prediction, only ever increased in speed with very large numbers of worker threads, all the way until the program began to get errors from having too many threads/request channels open concurrently. The program is compiled and run using make: 'make' to compile and 'make run' to run it using the arguments defined in the makefile, or with ‘./client –n <number of requests> -b <size of buffer> -w <number of workers>’. Running this program on the compute.cse.tamu.edu gives the following results:



Number of Threads	Average Time
1	108.05
2	53.50
3	35.88
4	26.89
5	21.56
10	10.77
15	7.22
20	5.43
25	4.37
30	3.66
40	2.79
50	2.28
60	1.95
70	1.71
80	1.59
90	1.42
100	1.35
125	1.17
150	1.10
200	Error: Too many

As the data shows, there is a clear relationship between the execution time and the number of active threads, while the size of the buffer has a much weaker correlation. The execution time decreased exponentially with the number of workers, such that the difference between one and two threads was an improvement of more than 50%, and after 40 threads or so, the effect of adding any more wasn't quite so worth it compared to the resources that it would require.