Kerry Zeng, Cesar Ortega
CSCE 313 Sections 503, 501

MP5 Report

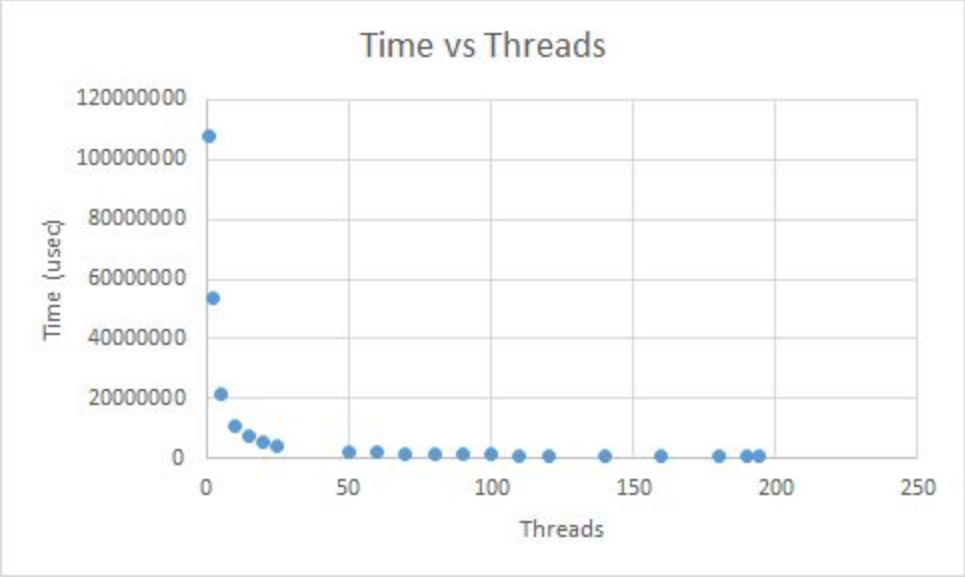
The purpose of this machine problem was to use multiple threads to handle and process requests. Through this, we learned how to thread a program for better performance as well as the effect that it can have as the number of threads increases. The code that we wrote included a thread safe buffer class ReqBuffer, a parameter struct PARAMS, a populateRequests helper method, and parts of the original code to incorporate the classes we wrote and handle the threading.

The buffer class serves as a thread safe container that the 3 different params (John, Jane, and Joe) can push and pop requests to. This is needed to keep track of the requests in a way that is not affected by having multiple threads running. It also allows for a centralized request buffer rather than a buffer for each "user."

The params struct contains a bunch of parameters including addresses to vectors, std mutex locks, and other information. This allows each thread to pass information between each other without the use of global variables. This allows for better memory management and no memory leaks.

The majority of the project was to modify the client.cpp in order to have it create 3 threads to populate the request buffer, and then *w* threads to process the requests. *w* is a variable that the user can set with a command line argument -w ## where ## is a number. We tested the program for up to 195 on compute.cse.tamu.edu with base *n* = 10,000. When running client, you can input *./client -n 10000 -w 195* to test that case.

The following is a graph of the running time vs w, the number of threads, for a set n=10000:



From the graph we observe that, performance improves in a linear fashion for values of w until around w=10, but in general it resembles a 1/x curve. This curve demonstrates Amdahl's law,

which states that performance will increase as the number of processors increase, but only up to a certain point after which the diminishing returns kicks in. After w=180, the performance actually decreased due to increased overhead; however, the run time for these all very similar and may be within the range of error.

The maximum number of threads in compute.cse.tamu.edu is *-w 195*. When you try to create past that, the OS throws a resource not available exception the number of times you exceed the max. For example, when I tested *-w 200* it gave me 6 such exceptions. In response to these exceptions, the client program seems to hang and no longer completes.

In conclusion, threads are helpful in dealing with large numbers of repeated instructions. By executing them in parallel, you reduce the total number of time that the cpu has to deal with a small set of instructions at high volume. However, if there are multiple variables at play, keeping the execution thread-safe may cause more overhead than the time it alleviates, and thus may not be worth it for certain applications.