Travis Takai

Lynne Okada

Aidan Gadberry

Assignment 4 Report

Overview and Build Instructions:

For the second phase of the MIS implementation we used TCP sockets in conjunction with the C++11 Thread library. To build our project type in “make” and the makefile will produce all the necessary executables. This will create a “server” and “client” executable where the server requires no arguments and defaults to port number 9999. For arguments the “client” executable takes in the .mis file being run, the address the server is on, and the name of the output file for .err and .out. The port number the client connects to defaults to port 9999.

Protocol Choice:

Since the ordering of the instructions in the MIS program is crucial, our decision was to use TCP to ensure there is a reliable connection between the sockets and the information obtained is always in-order. Especially in the case of concurrency, the ordering of the messages obtained were critical. With UDP we would not have the guarantee of receiving datagrams in order which would defeat the purpose of having an ordered MIS input. Additionally with UDP there is less of a guarantee of receiving messages intact or as reliably.

Threading Choice:

The reason for using C++ threads rather than the pthread library is synchronization between development environments for all of the project members as well as integration with C++ features such as lambda functions. Since not all of our members were using a Linux-based operating system the pthread library was not as easily usable for all members to efficiently program. Additionally the documentation and examples for the C++ thread library are very comprehensive and can integrate well into our pre-existing code. We also wanted to use some features such as lambda functions, which cannot be used in conjunction with pthreads as well.

Design Implementation:

Sockets:

Our server uses the TCPServerSocket code given by Dr. Sobh to enable a hand off of incoming data via new sockets. We treat each incoming connection as a new ClientThread as to eliminate time spent on building threads serially rather than concurrently. We transmit the contents of each file in a vectorized format from the Client to the Server object using TCPSockets. The Server object then accepts the connection and hands the newly created socket to a spawned ClientThread which handles the parsing and execution. When the ClientThread finishes execution it sends back the error and output messages to the Client via its stored socket.

Threading:

Since our pre-existing program from Phase 1 of the project is suited to handle file input and processing we used this functionality for each of our Client threads running on the server. The ClientThread object uses its own predetermined buffers for storing the input and output as well as the socket it is tied to. When running the MIS program, the ClientThread executes the code as if it were a single MIS object without regard to the ClienThreads since it is not sharing any resources with them. Each ClientThread object is spawned by the server when it receives a new connection so as to eliminate delay between serving incoming clients (eg not delaying to parse the incoming file and then spawning a thread). Within each ClientThread, the instructions can spawn a nested thread known as a WorkerThread. WorkerThreads are spawned once a ClientThread has verified that a given THREAD_BEGIN instruction also has a corresponding THREAD_END instruction. The WorkerThread also contains an MIS object that parses and runs the instructions given by the ClientThread. Each WorkerThread is synchronized by receiving access to the ClientThread's output buffers with a mutex to prevent unsynchronized writing to output. WorkerThreads are only joined if there is a given BARRIER instruction and run detached otherwise. This implementation allows for the execution of nested THREAD_BEGIN statements if they are used.