ECE 420

LAB 3: Gauss Jordan Elimination

SEC: H2

Kelvin Liang 1395077

John Symborski 1387814

Date: Mar 9th , 2017

***Description of Implementation***

For lab 2 we were given the task of implementing a server client program using pthreads. To start we were given a base implementation of the problem with a single client that interacted with the server. The client would send strings to the server which would echo back the entered string.

For the client side, we developed two methods to communicate with the server side. They were called readFromPos and writeToPos. The readFromPos calls would request to read from a specific location from the server. It would first send a ‘R’ char over to the server followed by an integer that indicates the position that the client wants to read from. Lastly the server would return the read information back to the client. The writeToPos call works in a similar fashion, but it instead sends a ‘W’ to the server to request that the client wants to a specific position. After this the client, would send the position in which to write to. When determining whether a client thread will read or write we use the mod function. If mod 20 = 0 for that specific thread number, we make that thread perform a write otherwise it performs a read. The above methods would then be used to perform the transaction.

On the server side of the we made two methods again readFromPos and writeToPos to send back information to the client. For the server side the key to solving the problem was having proper data protection when reading and writing from the array. We had two implementations that solved this problem. One implementation used a single mutex lock. A server thread would take the lock and lock out all other server threads. This lock would be used inside the writeToPos and readFromPos methods. In our second attempt, we used to read write locks instead to protect our array replacing the mutex lock. This lock allows multiple reads or a single write at one time. The server thread would wait for requests from clients. It would wait for a R or W which would tell the server if it was a read or write request. Once the server received a connection request it would create a thread to handle that request.

***Testing and Verification***

As we were developing the program we started with implementing a single request client server system and just relying on a command line entry to tell the client which spot to read or write from. This was to ensure that the reading and writing commands were being processed correctly on the server side. This testing was mostly through print statements. Once we could confirm that we were reading and writing correctly we added in thread creation and print statements on the client side. We put in the mutex lock on the server side. The client would print off all the messages it received from the server. We then started with a reasonable amount of threads 20 or so. We would run the program and then inspect the print statements to ensure that the writes were changing the values inside the array. Once we confirmed this, we scaled the program to run 1000 threads with the print statements. We would copy and paste all the print statements into a text editor. Once they were in the text editor we would inspect a couple random spots in the array. We would use search functions to find the spots we read and wrote to that position and check to ensure the values were what we were expecting. Once we ran this test a couple time we confirmed that the clients server system we set up was functioning correctly.

Top of Form

Bottom of Form

******

Figure 1: CDF of Mutex vs RW Lock for Array size of 10

******

Figure 2: CDF of Mutex vs RW Lock for Array size of 100

******

Figure 3: CDF of Mutex vs RW Lock for Array size of 1000

******

Figure 4: CDF of Mutex vs RW Lock for Array size of 10000

******

Table 1: Average Times for Synchronization Techniques

***Performance Discussion***

For this lab, we had to find a way to properly synchronize our client and server threads so that reading and writing from the array would be accurate and fast. For this we made two attempts at solving the problem as described above. We used a mutex lock implementation and a read write lock implementation. With the mutex lock, the biggest problem was that the lock mechanism would only allow one thread to access the array at one time. This created a lot of overhead as threads would have to line up and wait to receive the lock before any work could be done. The mutex lock was great in the sense that it solved the race condition problem however it made processing rather slow. Using the mutex lock did not fit well into this problem since most the transactions used in the client server program were reads. This meant that the lock was rather redundant as multiple threads should be able to read simultaneously due to the consistency of data.

To solve these underlying issues, we introduced the read write lock for our second implementation. These locks allowed for an unlimited number of readers while still providing the same protection whenever a thread needed to write to our array. This optimized our processing times for handling the server requests, as threads would not have to queue as much. Readers would be able to freely access the array at the same as other readers. The key to understanding the problem was that for this problem 90% of the transactions that occurred were reads. Knowing this we see that the read write locks should greatly outperform the mutex lock. This would be different if we had a greater number of writers as the behavior would be more like the mutex lock implementation.

Looking above to the four CDF plot figures we can see how much better the read write lock was versus the mutex lock. Looking throughout we see that all plot lines are mostly straight vertical, suggesting that our timing of the problem was consistent, as we do not have much variation in our time measurements. We can observe how much better the read write lock was as the speedup is roughly 3-4 times faster than when we used the mutex lock. This suggests that the read write locks were more efficient in processing the requests. This is likely due to again the nature of the problem, as we have significantly more read transactions than writes. More read transaction can be processed at one time with the read/write locks.

Another interesting performance trend we noticed with our program was that increasing the array size contributed to more speedup for our program. This is likely due the problem of false sharing. False sharing is when you have multiple threads accessing data that share the same cache line, whenever one thread writes it cause the cache controller to invalidate the rest of the line. This is problematic as it creates more overhead for the program as the threads take longer to access memory. For our problem increasing the array size allowed for memory to be separated into different cache lines, decreasing the amount of invalidates that would cause false sharing. Thus, this made our program faster as we went up in array size.

***Conclusion and Experience***

In this lab, we were required to implement a multi-threaded server client program that can handle a large amount of client’s requests. These requests are a read or write operation to a random location in an array of a specific size. The operation will be processed on the server side and a message with the results will be sent back to the client side. We were asked to improve the performance of this chat server as much as possible, specifically by reducing the read/write latencies caused by our synchronization implementation. The client would send many requests at once and the server would need to handle them all while properly protecting the critical section (reading and writing to/from the string array) and not causing race conditions. In this lab, we learned how to implement two different types of synchronization techniques. We implemented a server which protected the critical section with a mutex lock and one that used a read write lock. The read write lock as we discussed earlier was the superior choice for this problem causing a speedup of 3 to 4 when processing 1000 requests. We also learned about the effects that false sharing can have on a problem. As the problem size increased the amount of false sharing decreased, thus decreasing processing time. Overall this lab taught us about synchronization, false sharing and performance related to specific synchronization implementations.

Top of Form

Bottom of Form