ECE 420

LAB 3: Gauss Jordan Elimination

SEC: H2

Kelvin Liang 1395077

John Symborski 1387814

Date: Mar 9th, 2017

***Description of Implementation***

For lab 3 we were given the task of implementing a program to solve a linear system of equations. As part of the assignment spec we had to make use of Gauss-Jordan elimination in conjunction with OpenMP. To start we were given a serial implementation of Gauss Jordan elimination. We utilized the given lab3IO functions to start as we needed to generate our data input data and feed it to the serial implementation. We started our approach to the problem by trying to figure out where in the current solution we could parallelize and make more efficient.

After careful analysis, we figured the best places to parallelize would be the calculation portion of the Gaussian elimination, the Jordan elimination, the solution section of the problem and finally the overarching for loop of the serial implementation. We began working with the Gaussian elimination calculation section, at first glance we saw that the section was composed of two for loops. Because of this our first thought was to try using the parallel directive “collapse” due to the nested for loop. This approach would not work as we soon realized the nested for loop did not have a rectangular iteration space. With that in mind we used a #pragma parallel for directive to start due to the use of for loops within the section. We had to make k, Au, index and size shared variables as all threads would need these values to perform work. The iteration values i and j were made private so that threads could handle different iterations. Temp was also private. The next part of our implementation was to parallelize the Jordan elimination. For this section, we had to put the parallel for directive inside the initial for loop, and before the nested one. This was due to the nature of the outer for loop, as k started from size, decreased and was utilized in the inner loop. Parallelizing on the outside would cause incorrect values for the solution. Once these two sections were done we simply used #pragma parallel for directives for the solutions section and the overarching for loop. This was done simply because of the complexity of the outer for loop as lots of instructions are performed in the block. For the solutions section, there were no variables we could make private to help make indexing faster.

One final addition we made to make the program faster was to make a temp value for the Au[index[k]][k] operation in the Jordan elimination section. This value is twice in this section so by making a Temp value for it we save the program overhead time as it does not have to index the Au and index array for values every single iteration.

***Testing and Verification***

As we were developing the program we tested by making gradual changes to the overall program. We would first start with a section to work on and after we made a few changes we would compile and run the program using the check.sh script given to us. This was an effective test method for us as we could have confirmation that our changes kept the functionality of the program intact while allowing us to observe any speedup. Timing was added before the solution to the linear system began and finished once solved. Once we knew the program still solved the data input correctly our focus was redirected to the timing of our system. The check.sh script would run the program with problem sizes of 64, 256 and 1024 while changing the # of threads trying 1, 4 and 16. Timing for all combinations would be returned so that we could calculate the speedup of the program due to parallelization, by dividing the time for a sequential solution (1 thread) by the time for 4 or 16 threads. Using this information we would focus on single sections of the sequential program trying to get as much speedup as possible by parallelizing just that single section. Once we were happy with the speedup from the parallelization of the single section we would move on to other sections and repeat the same procedure. Overall we came up with four implementations as seen below. Gradually our speedup for 1024 samples went from 3.59 to 3.69 when comparing the use of 4 threads. 4 threads should be the ideal number as it represents a 4-core computer. Below are the results of our testing.

Table 1: Implementation #1 Parallelization of Calculation for Gaussian Elimination Added

|  |  |  |  |
| --- | --- | --- | --- |
| **Samples** | **Threads** | **Time** | **SpeedUp** |
| 64 | 1 | 0.00102 | N/A |
| 64 | 4 | 0.00073 | 1.397260274 |
| 64 | 16 | 0.0043 | 0.237209302 |
| 256 | 1 | 0.039407 | N/A |
| 256 | 4 | 0.011829 | 3.331388959 |
| 256 | 16 | 0.029939 | 1.316243027 |
| 1024 | 1 | 2.2293032 | N/A |
| 1024 | 4 | 0.620617 | 3.592075628 |
| 1024 | 16 | 0.832535 | 2.677729104 |

Table 2: Implementation #2 Parallelization of Jordan Elimination Added

|  |  |  |  |
| --- | --- | --- | --- |
| **Samples** | **Threads** | **Time** | **SpeedUp** |
| 64 | 1 | 0.000776 | N/A |
| 64 | 4 | 0.000437 | 1.775743707 |
| 64 | 16 | 0.015844 | 0.048977531 |
| 256 | 1 | 0.038388 | N/A |
| 256 | 4 | 0.021834 | 1.758175323 |
| 256 | 16 | 0.036595 | 1.048995764 |
| 1024 | 1 | 2.202961 | N/A |
| 1024 | 4 | 0.609167 | 3.616349868 |
| 1024 | 16 | 0.831312 | 2.649980994 |

Table 3: Implementation #3 Parallel for Directives Added for Solution and Outer Loop

|  |  |  |  |
| --- | --- | --- | --- |
| **Samples** | **Threads** | **Time** | **SpeedUp** |
| 64 | 1 | 0.001105 | N/A |
| 64 | 4 | 0.00043 | 2.569767442 |
| 64 | 16 | 0.022912 | 0.048228003 |
| 256 | 1 | 0.052014 | N/A |
| 256 | 4 | 0.23737 | 0.219126259 |
| 256 | 16 | 0.042305 | 1.229500059 |
| 1024 | 1 | 2.217369 | N/A |
| 1024 | 4 | 0.605016 | 3.664975802 |
| 1024 | 16 | 0.830947 | 2.668484272 |

Table 4: Implementation #4 Omp parallel added above Jordan

|  |  |  |  |
| --- | --- | --- | --- |
| **Samples** | **Threads** | **Time** | **SpeedUp** |
| 64 | 1 | 0.001115 | N/A |
| 64 | 4 | 0.00045 | 2.477777778 |
| 64 | 16 | 0.016508 | 0.067543009 |
| 256 | 1 | 0.047742 | N/A |
| 256 | 4 | 0.021865 | 2.183489595 |
| 256 | 16 | 0.043497 | 1.097592937 |
| 1024 | 1 | 2.192299 | N/A |
| 1024 | 4 | 0.592572 | 3.699633125 |
| 1024 | 16 | 0.827131 | 2.650485836 |

******

***Performance Discussion***

*For this lab we were required to use OpenMP whilst attempting to speed up solving a system of linear equations by Gaussian elimination with partial pivoting. We have broke our step by step improvements into 4 distinct implementations of improvement. By the end of the experiment we we were able to get a speedup up of approximately 3.7 when comparing to the single thread run time. I will first cover the what was done in each step and why it should cause improvement. After that I will discuss the chunk size and scheduling pattern trends in the tables above.*

***Implementation 1:*** *In this implementation we made the calculations after the proper pivoting parallel. This causes a significant increase in the speedup because the entire section that does all the calculation can be ran in parallel. This causes a significant speedup because there is no critical section overhead to cause additional overhead and the size of each iteration is constant. Since the size of these iteration are constant the static scheduling algorithm by default is the best scheduling. Basically each loop will take the same amount of time so simply split up the iteration equally among thread for best performance.*

***Implementation 2:*** *In this implementation we added a parallel component to the inner for loop in the Jordan elimination process. Even by just putting the inner for loop in parallel we were able to see small increase in speed up. We are able to compute this inner loop much faster for each iteration. This causes the overall Jordan elimination process. The scheduling we chose is the static with the default chunk size. This makes sense because each for iteration is doing 3 computations and hence each iteration is going to take approximately equal time to complete.*

***Implementation 3:***  *In this implementation we simply added a “#pragma for” to the spot where we assign the index array and to where we calculate the solution. These for loops may not take a huge amount of time but we did find an increase in the speedup when we put these for loops in parallel. Since we are dealing with such a small run time in total any increase in time causes a increase in speedup. Using the default scheduling policy and thread number is appropriate because of the simplicity of these loops.*

***Implementation 4:***  *In this implementation we simply changed the way syntax in our openMP Jordan Elimination. We moved “#pragma omp parallel” above the upper loop and then simply have a “#pragma omp for” before the start of the inner loop. According to the notes there is one fork and join with each parallel for iteration and putting this on the inner loop would cause an overhead due to excess of fork and join calls. By moving it to the outside of the loop we will reduce the number of fork and join calls hence reducing the overhead of making these calculation parallel. We saw a small increase in speedup by making this improvement from ~3.67 to ~3.699.*

***Chunk Size and Scheduling policies:*** *In order to achieve the max speedup we found that when it came to scheduling policies less is more. For both major areas we made parallel we found that the default scheduling policy was the fastest. This is a static scheduling policy that specifies chunk size of iteration/number of threads. For both sections this makes sense because for the most part the iterations would take almost identical time. By dividing this iteration up equally each thread should take the same amount of time to finish and cause the largest increase in run time without taking any overhead for complex scheduling policies. The dynamic and guided scheduling policies saw less increase because it takes additional overhead to actual implement a scheduling policy that changes at run time.**Changing the chunk size too large or too small imbalances the workload for each thread causing some thread to do more work then others. This causes an increase in run time because some thread would finish before the other did and at some point would not be doing any work. These factor contributed to us using the default scheduling policy with the default chunk size.*

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