Bryan Klee

0624280

Grid Assignment

The locking mechanism I used was a simple mutex lock. This worked very well for the grid level locking since it locked out any other thread from entering the swapping portion of the do_swaps method. I also used this kind of lock for the threads left section as well. When I got to the row and cell level I struggled a bit. I wanted to essentially give up what the thread would be doing if it tried to enter a row that was currently being used by another thread. So for this I used a condition variable which would make the thread sleep until another thread woke up it. To do this I had to create 2 queues, one which would hold all of the rows/cells currently being used by a thread and the other which help all of the rows/cells waiting for a row or cell to be unlocked. I had 2 methods, in which one that would check to see if a row/cell was being used and then sleep the thread if it was and another which would check at the end if the locked out rows/cells were able to be unlocked and could be swapped. I believe this also may be the reason why it was not 100% correct as when the thread that was woken would have to check if another thread entered and essentially "took" the row or cell and would have to be slept again. This also solved my deadlock problem as no thread would ever be stuck waiting for another thread to finish, for example if there was only one thread left and it was done then the method would see that no other thread is currently being used and would wake up the sleeping thread to continue on.

Obviously when ran with no locking system, it is much quicker than any other system, however the memory is corrupt so in real world we would rather have a system that is correct and slow than incorrect and fast. As grid size goes up the time goes down, except for the grid level lock. This is because only one thread can go in at a time and the bigger determinant is how many actual threads you have to complete their actions rather than how fast it can do its actions since the time is constant between swapping and only one swap can be performed at a time between each thread. The more interesting thing is looking at the row and cell level locking. As there are more place to go, the more threads can perform the swaps, instead of waiting for open areas. If there is a 2x2 grid and 10 threads, then 7-8 threads will be waiting for the other 2 threads to finish, while in a 10x10 grid potentially all 10 threads can access the grid at once making it much quicker the complete the whole action. This is even quicker with cell level locking as more threads can get into the grid and make changes, and the more real estate for changes the less chance of threads being locked out.

As the number of threads increase the time increases across the board. This is because as there are more threads the potential for more threads to be locked out increases and also more operations need to be done on the grid as each thread needs to do a certain number of operations before it is done. Cell level locking is still the lowest, no matter the amount of threads, because it allows the most amount of threads to access the grid at once. Looking at granularity, the time goes from grid, row then cell as the fastest. This makes sense, as grid only allows 1 thread to access the grid at a time, row allows a few more threads to enter and cell allows the most amount of threads to enter without causing memory corruption. The finer granularities have the biggest effect as the size of the grid increases and

as the number of threads increases. Again this is because they can allow more threads into the grid, so if you have more threads then they can finish faster or if you have a bigger grid they can do more operations.

When moving the sleep(1) call I noticed something very surprising. Whether commented out or moved to before the entire swap process, the whole thing worked even with no locking method chosen. I believe it is there because it is before the actual assignment and when one thread is sleeping another is grabbing the temp variable and the other thread wakes and changes the grid which now makes the temp variable the other thread grabbed wrong, corrupting the grid. The best time to use locking and synchronization would be if multiple threads are trying to reach data in different order such as in two different method or were doing different operations, usually in a different order. Basically the sleep makes the program not work and if it was not there then the assignment would be very easy!!