Charles Antoine Malenfant (0616332)

CS444 – Grid Assignment Writeup

Locking and Deadlock Prevention Strategies

For this assignment, there were four different granularities of locking possible. They are listed below. Each level of locking is briefly described and, if needed, the deadlock prevention strategy is outlined. Please note that if the randomly generated indices of the numbers to be swapped happen to be the exact same (for row and column), the loop in the do_swap method continues since it would be trivial for a number to swap itself!

NONE: This is used for comparison only. There is no locking or deadlock prevention for this granularity. Data integrity violations should be expected.

GRID: This is the third finest level of granularity possible for the grid. The level suggests locking the entire grid every single time a swap between two numbers from the grid is to be executed. This involves a simple lock that is to be acquired and release before and after every swap executed by a thread. Deadlock is not possible in this situation since the "hold and wait" necessary condition for deadlock to occur is broken hence there is no deadlock prevention needed. There is only one big lock, so once a thread acquires the lock, it is not possible for it to wait for another lock since there is only one.

ROW: This is the second finest level of granularity possible for the grid. This level proposes to have locks available for each row of the grid. This can be achieved by creating a one-dimensional array of locks that is the size of the grid. To give a better idea, the array looks like: rowLevelLock[MAX_GRID_SIZE]. Each index of that array is to contain a simple lock and is to refer to a row in the grid. During initialization, only the required locks would be initialized. This is achieved by traversing the array the amount of times the user decided the grid size to be during the program call (see gridapp.c). When a random row (index) is generated corresponding to the location of the first number to be swapped by a thread, the thread tries to acquire the lock for that specific row by accessing rowLevelLock[row]. Similarly, the second random row (index) for the second number to be swapped by the thread is generated and the thread waits to acquire its corresponding lock from the array. Once the two rows containing the numbers to be swapped are locked and the swap is executed by the thread, the row locks are released by the thread for other threads to use. It is important to note that if the randomly generated rows are the same, meaning that the numbers to be swapped reside in the same row, only one row needs to be locked. Deadlock is bound to happen using this granularity. What if a thread locks row (index) 2 and is waiting for the lock for row (index) 4 and meanwhile, another thread locked row (index) 4 and is waiting for the lock for row (index) 2? This is just one example of a situation where deadlock could occur. Deadlock prevention for the row granularity is implemented by breaking one of the four necessary condition for deadlock to live on. More specifically, just as in the dining philosopher's problem, we are breaking the circular wait condition. This condition is broken by numbering the shared resources and by having threads always try to acquire the lowest-numbered resource first. In this assignment, the resources are locks and they are numbered by the grid. In other words, whenever a thread needs to acquire two separate row locks, it will go for the lowest numbered randomly generated row (or index). For example, rows 3 and 5 are randomly generated and depict in which row the two numbers to be swapped by the thread are. The thread proceeds by trying to acquire row (index) 3 first, then row (index) 5. If the two rows (indices) generated are the same, deadlock will not occur. This suffices to eliminate deadlock for the row granularity.

CELL: This is the finest level of granularity possible for the grid. This level suggests having locks available for every single cell in the grid. The strategy to successfully implement this is very similar to the row granularity described above. A two-dimensional array such as: cellLevelLock[MAX_GRID_SIZE][MAX_GRID_SIZE] is necessary. Each index i, j of that array is to contain a simple lock such that cellLevelLock[i][j] is to refer to the cell in the grid at row i and column j. The initialization is executed the exact same way as for the row granularity meaning that the two-dimensional array is traversed the length that the user specified in the program call, initializing each lock in each cell along the way (see gridapp.c). When a random row and column are generated corresponding to the cell of the first number to be swapped by a thread, the thread tries to acquire the lock at cellLevelLock[row][column]. Similarly, the second random row and column are generated for the second number to be swapped and the thread waits to acquire its corresponding lock from the array. Once the two cells containing the numbers to be swapped are locked and the swap is executed by the thread, the cell locks are released by the thread for other threads to use. Deadlock is also a possibility for this granularity. What if a thread locks the cell at cellLevelLock[1][6] and wants to lock the cell at cellLevelLock[7][3] to do the swap but meanwhile another thread who already locked cellLevelLock[7][3] wants to lock the cell at cellLevelLock[1][6] to do its own swap? Deadlock prevention for the cell granularity is done the exact same way as the row granularity. The circular wait condition is broken by numbering the shared resources and by having threads always try to acquire the lowest-numbered resource first. Whenever a thread needs to acquire two separate cell locks, it will go for the lowest numbered randomly generated row first and then the lowest numbered randomly generated column if the cells are in the same row. For example, assume rows 3 and 3 are randomly generated and depict in which row the two numbers to be swapped by the thread are. The thread then checks the randomly generated columns for the two cells in which the numbers to be swapped reside. Seeing that the first cell is in column 9 and the second cell in column 4, cellLevelLock[3][4] will be the first lock the thread will try to acquire followed by cellLevelLock[3][9]. This suffices to eliminate deadlock for the cell granularity.

Also, note that another lock is used to protect the threads_left data structure. The lock is acquired every time threads_left is incremented or decremented. It is a simple lock that prevents the program from running forever if something is missed. This needed to be done because threads_left is a shared state among all threads.

Graphs

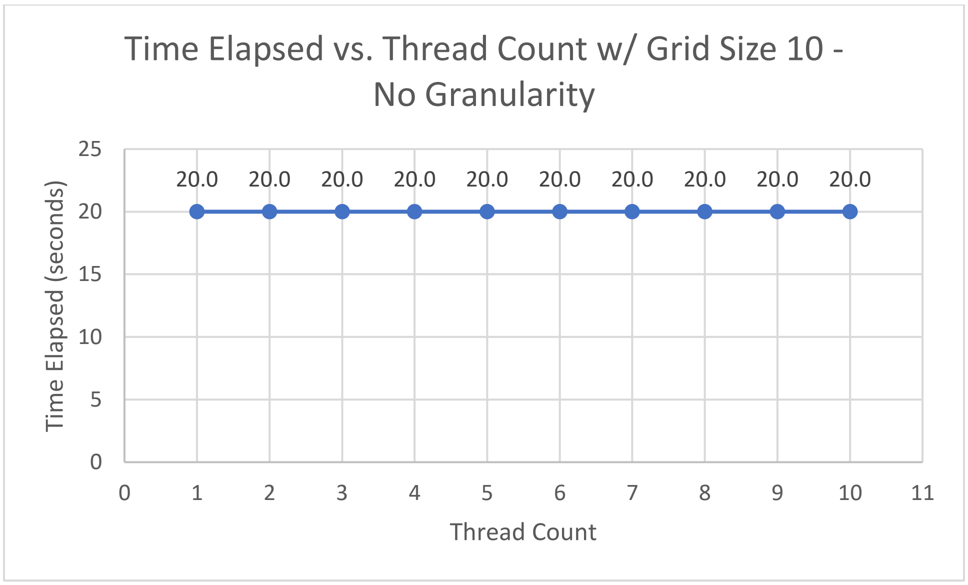

Figure 1. Time elapsed vs. thread count for no locking granularity at a constant grid size of 10.

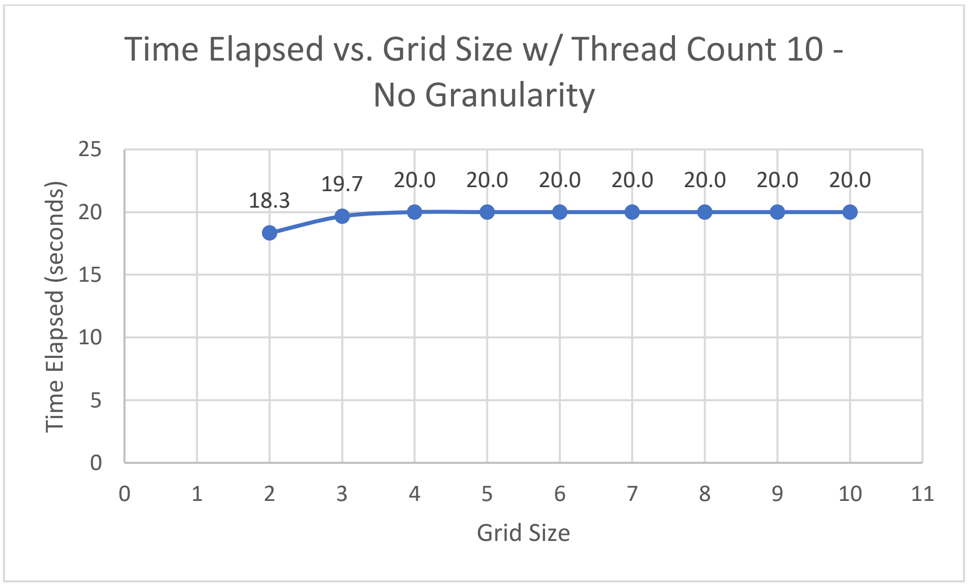

Figure 2. Time elapsed vs. grid size for no locking granularity at a constant thread count of 10.

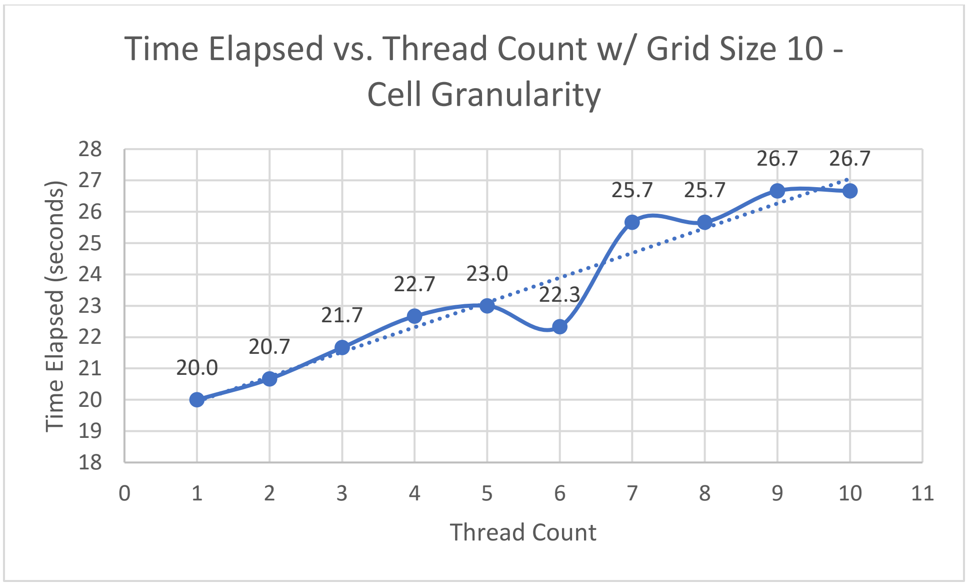

Figure 3. Time elapsed vs. thread count for the cell locking granularity at a constant grid size of 10.

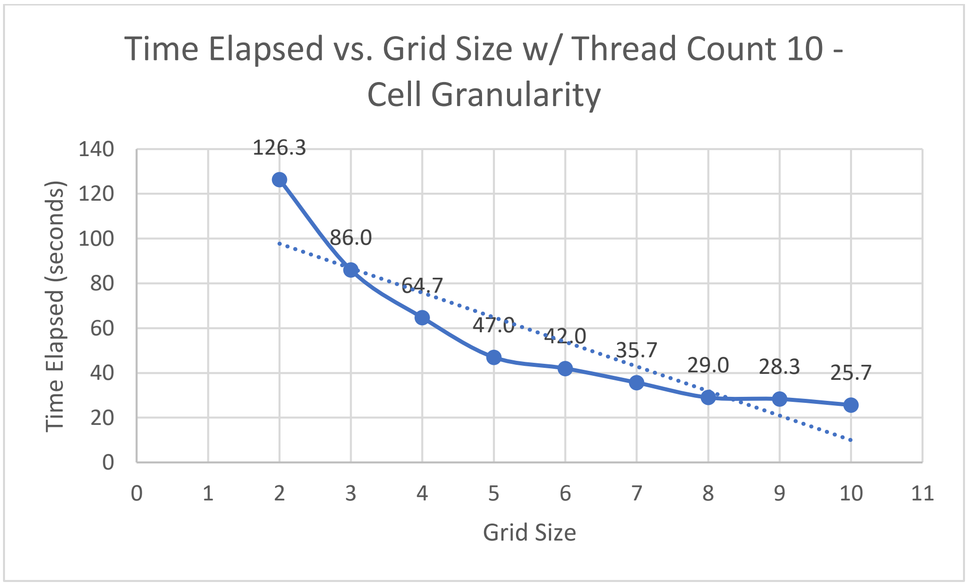

Figure 4. Time elapsed vs. grid size for the cell locking granularity at a constant thread count of 10.

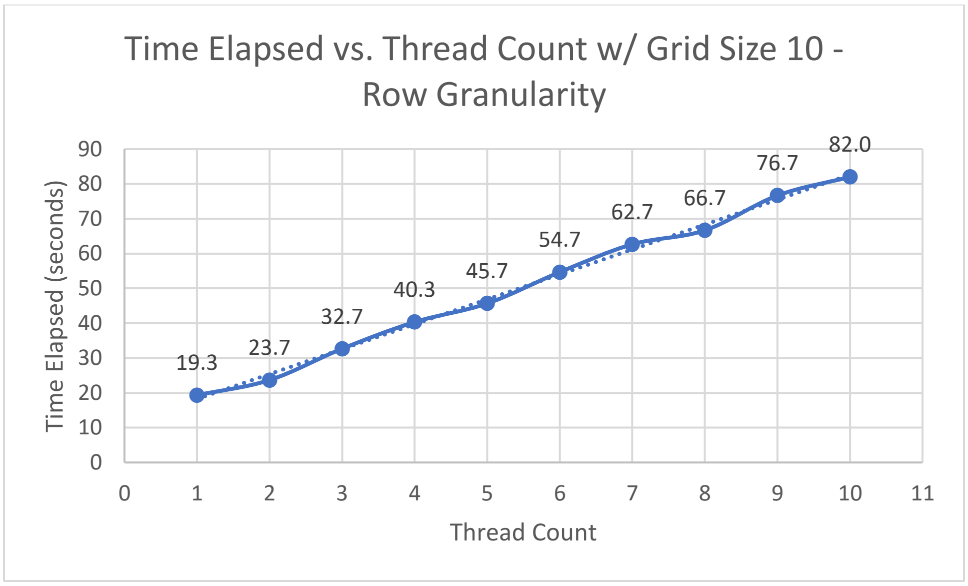

Figure 5. Time elapsed vs. thread count for the row locking granularity at a constant grid size of 10.

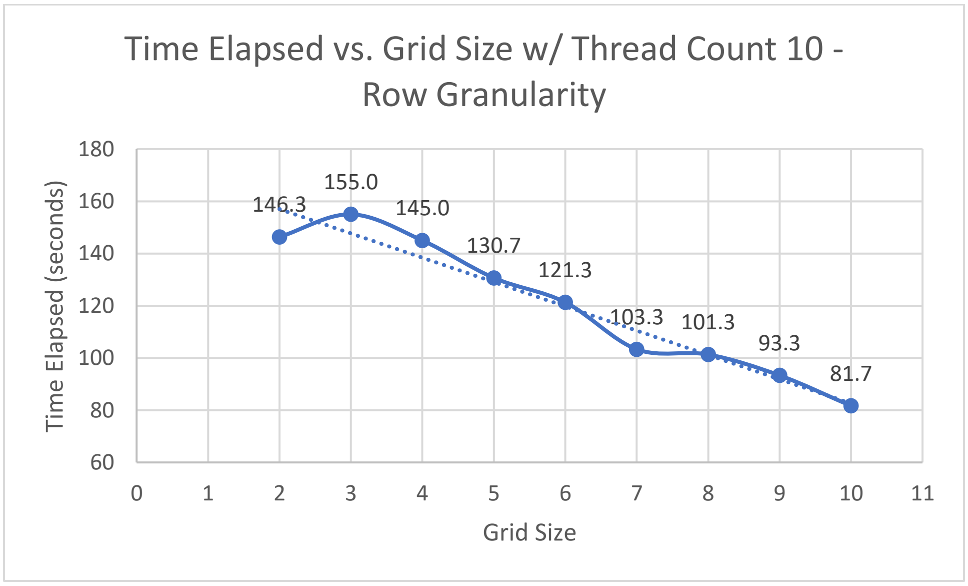

Figure 6. Time elapsed vs. grid size for the row locking granularity at a constant thread count of 10.

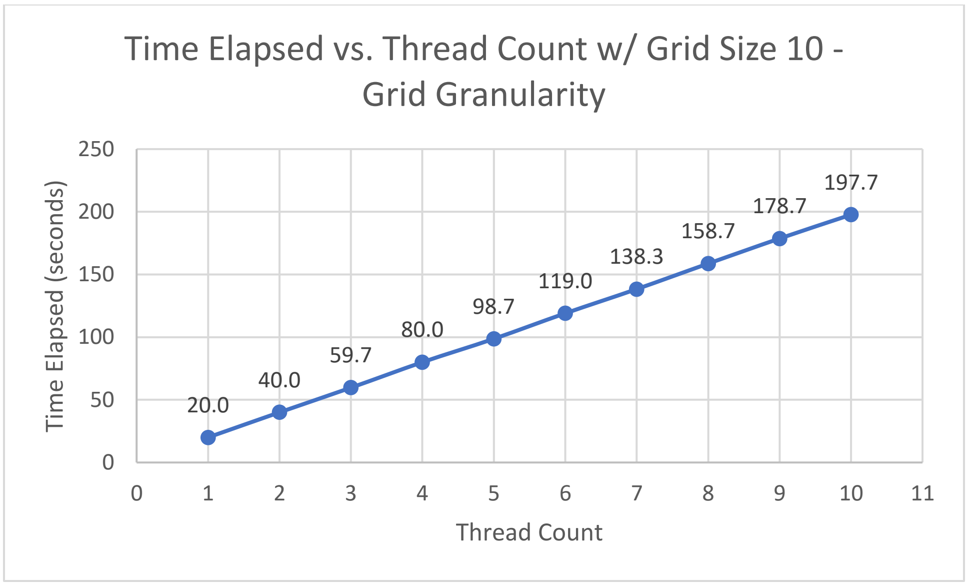

Figure 7. Time elapsed vs. thread count for the grid locking granularity at a constant grid size of 10.

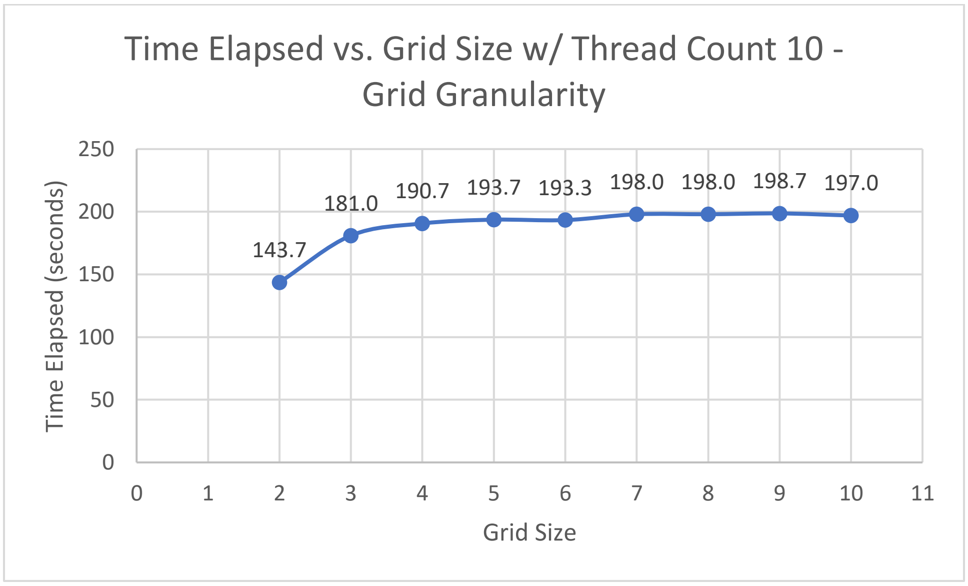

Figure 8. Time elapsed vs. grid size for the grid locking granularity at a constant thread count of 10.

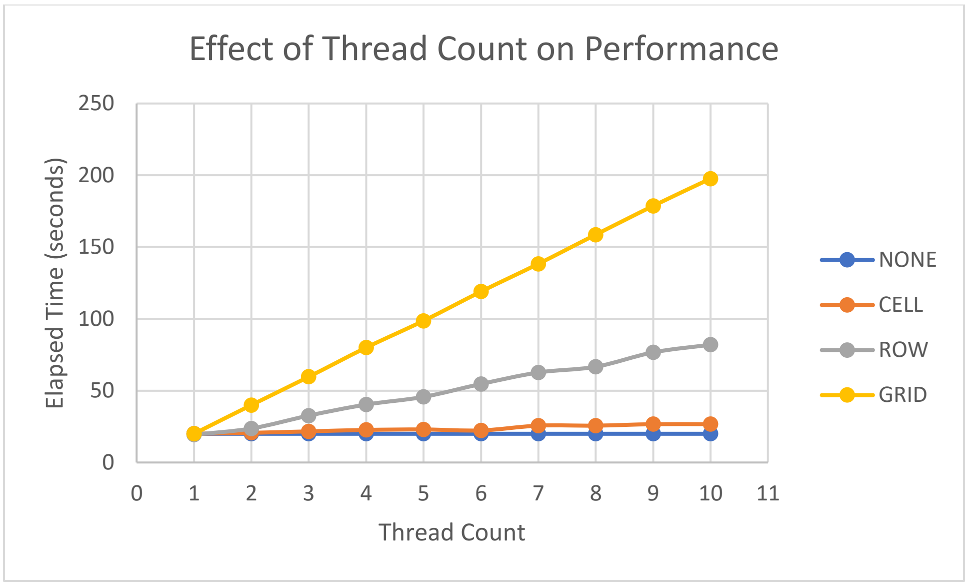

Figure 9. Comprehensive chart of the impact of thread count on performance.

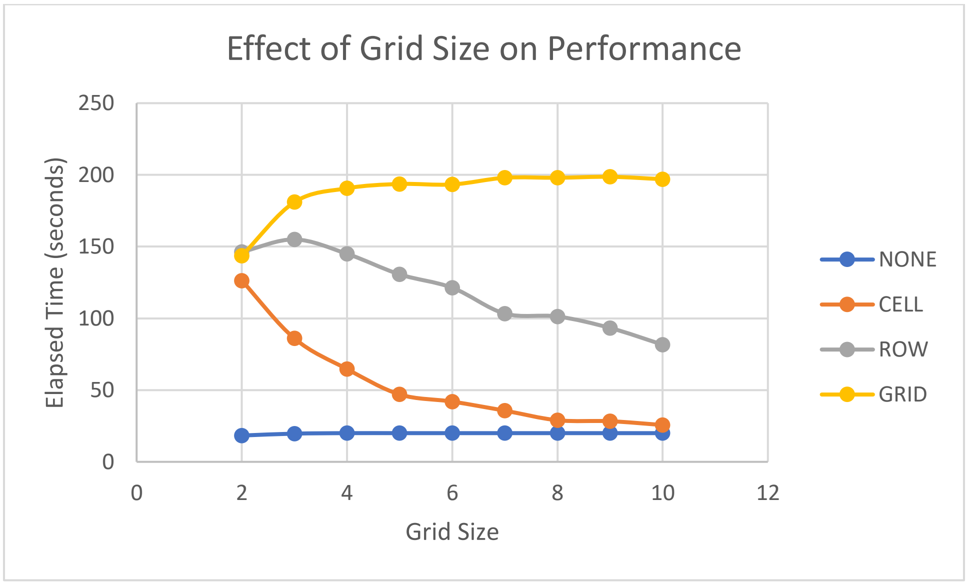

Figure 10. Comprehensive chart of the impact of grid size on performance.

Analysis

Effect on performance due to…

Grid size: The finer the granularity, the more impact the grid size has on performance. If we look at Figure 4 and Figure 6 on pages 4 and 5 (or Figure 10 on page 7), we can see that the bigger the grid, the less the chance of contention between threads for shared resources which is synonym to better performance hence the smaller the time elapsed to execute the threads. The effect is at its peak for the cell level locking scheme where there is about a 100s time difference between having a grid of size 2 compared to size 10. To conclude, one could say that grid size and performance and inversely related for finer granularities of locking. When there is no locking present, grid size seems to have little to no effect because there is no contention for locks between threads. Likewise for the grid locking scheme which makes sense since one big lock is used no matter how big the grid is.

Number of threads: The finer the granularity, the less impact the number of threads has on performance. Looking at Figure 3, Figure 5, and Figure 7 on pages 4, 5 and 6 (or Figure 9 on page 7), we can clearly infer the proportional relationship between thread count and performance. This relationship is lessened as granularity becomes finer. There is a decrease in performance by about 180s for the grid locking scheme. This is logical since the more there are locks, the less likely a thread is going to wait on a lock to execute a swap. However, the more the threads, the higher the chances of potential conflicts for a resource since they are all executing in parallel. For the grid granularity, there is only one resource (the big lock) that is shared among 1 to 10 thread(s). As for the cell granularity, if we assume a grid size of 10, there are 100 locks total that are shared among 1 to 10 thread(s). Just as the grid size, the number of threads has no effect on performance when no locking granularity is implemented. This is true because, again, there is no contention for resources (locks) since there are none. In other words, more threads mean more chances to violate data integrity but no change on performance since a thread can't be held back by waiting for a resource.

Granularity of locking: The finer the granularity of locking the more enhanced the performance. This can be deduced from Figure 3 through 10 on pages 4 through 7. The more intricate the locking scheme, the more resources are made available to the threads to use. For example, in the case of the cell granularity, if a thread wants to swap a certain cell, it only renders that specific cell unavailable to other threads for use during the time it takes to do the swap. For the row granularity, it renders an entire row of cells unavailable to other threads to manipulate. Even worse, for the grid granularity, every time a thread makes a swap, the entire grid is locked!!! This means that when two cells are locked by a thread, the chances of another thread coming in for a swap and requesting one of the same two cells are very low, hence the better performance. The chances are much higher when entire rows and locked and the chances of threads having to wait are 1005 for the grid granularity. This can easily be observed in Figure 9. and Figure 10. on page 7. In the case of no granularity, we observe steady performance.

sleep(1): The sleep(1) statement definitely slows down performance. However, I believe this lessening of performance is vital to the assignment. I think the sleep(1) piece of code is there to simulate a longer swap time or bring the “toy problem” for the class a little closer to a real-world situation. The swapping of the two numbers in the grid is comprised of about three statements that execute very quickly in constant time. Hence, the swap becomes trivial in the grand scheme of things and so do the locks without the sleep(1) statement. The swapping is so fast that all the threads swapping execute immediately and finish immediately without there ever being a chance of accessing shared resources at the same time (contention) or violating the integrity of the data. Commenting this statement or moving it away from the swapping makes the locking insignificant. All granularities under any circumstances obtain the correct final sum since there is no data violation due to the quickness at which the program executes the swap without the sleep(1) statement. Also, again for all granularities under all circumstances, the time elapsed is 0 seconds. This is surprising because I did not know that the program could be executed so fast even with 10 threads, grid size 10 at the grid granularity without any contention regarding shared resources. To sum up, this assignment would be useless without the sleep(1) call and it makes sense to write multithreaded applications that are synchronized only when the critical section of the use of a shared resource executes in a non-trivial amount of time.

When run with no locking and only 1 thread, I do not observe any data integrity violation. Data integrity violation occurs when two threads tries to swap the same number at the same time meaning that one thread will access the wrong value for the number since the other thread is not done with its swapping operation. This is not possible to occur when only one thread is active. In other words, there cannot be accessing of the same resource by two entities at the same time if there is only one entity. Data violation can indeed be avoided if few enough threads are let loose on a big enough grid. If we refer to the Excel spreadsheet where my data resides, we can see this happened when the program was run under no granularity of locking with a grid size of 10 and a thread count of 2. The numbers that the threads attempt to swap are selected at random by the program. It could happen, through luck, that the randomly selected cells to be swapped for all concurrent threads are never the same throughout the entire life span of the program. This is more likely to occur if there is less threads and more cells to choose from.

In conclusion, finer granularity is the most efficient way to implement locking performance-wise for sure when integrity of the data needs to be maintained. It is harder to implement finer synchronization in multi-threaded programs but the cost is well worth it in the long-run. The effect of finer granularity is at its peak when there are multiple threads at play. However, finer granularity loses its edge when the grid size becomes really small so it is paramount to evaluate the tradeoffs and program according to our needs and to the complexities of the situation at hand.