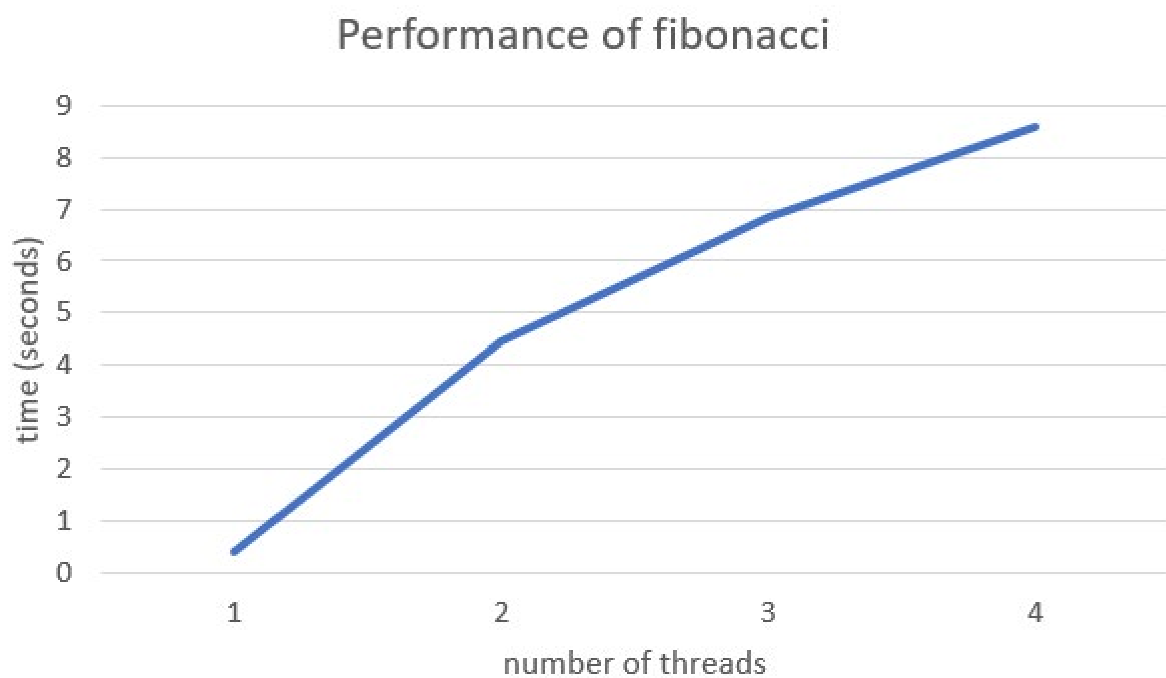
Daniel Murphy
CSCI 6454 – Fall 2020
Homework 1 – report

Question 1

Sample input/output:

```
dooms@doomsnack:/media/sf_ParallelAndSciComp-6454/hw/HW2/code$ gcc p1.c -o p1 -fopenmp
dooms@doomsnack:/media/sf_ParallelAndSciComp-6454/hw/HW2/code$ ./p1
input   result          no. threads     proc time
35      9227465         1               0.403113
35      9227465         2               4.456324
35      9227465         3               6.836462
35      9227465         4               8.566511
```



Here we see that increasing the thread count actually increases the time to compute a recursive Fibonacci. This is likely because recursive Fibonacci is a fundamentally linear equation- each iteration requires the output of the previous iteration, so instead of improving performance, parallel processing only adds the overhead of thread management.

One can envision a more efficient algorithm that stores previously calculated results in a hash table- this might eliminate some expensive calculations and may be a better candidate for parallelization.

Question 2

Sample input/output

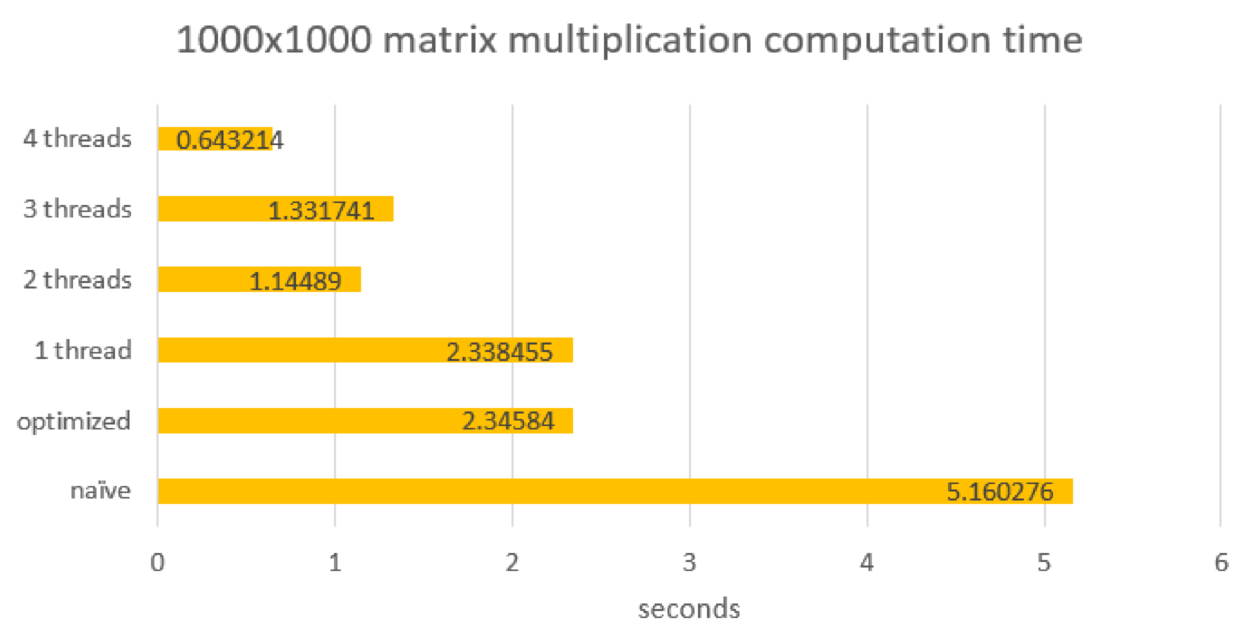
(note: the program only prints the input and result matrices for matrix sizes <= 100x100)

```
dooms@doomsnack:/media/sf_ParallelAndSciComp-6454/hw/HW2/HW2_Daniel_Murphy/HW2_Daniel_Murphy_2$ gcc p2.c -o p2 -fopenmp
dooms@doomsnack:/media/sf_ParallelAndSciComp-6454/hw/HW2/HW2_Daniel_Murphy/HW2_Daniel_Murphy_2$ ./p2

INPUT:
MatA: --------------------------        MatB --------------------------
3       3       5       4       4       2       8       3       8       0
1       6       7       5       4       2       8       3       6       5
3       0       2       7       6       1       4       4       2       0
2       3       6       5       8       4       2       3       1       7
0       6       0       3       4       5       8       4       3       2

RESULT:
MatC: ---------------------------
53      108     66      68      51
61      126     80      75      73
66      94      62      53      61
76      138     86      75      66
44      86      43      51      59

CALCULATION TIMES:
naive:          0.000001 seconds
optimized:      0.000001 seconds
4 threads:      0.000153 seconds
```



I found the naïve implementation to be exceedingly slow. Improving performance required taking into consideration the locality of the data. By transposing the MatA matrix in the optimized implementation, significant improvements could be made.

To realize a parallel implementation, it was necessary to write the results to the MatC matrix in transpose form to ensure that separate threads were not attempting to writing to contiguous memory locations. Fortunately the extra overhead of transposing MatC back requires relatively insignificant overhead.