Exercise 5

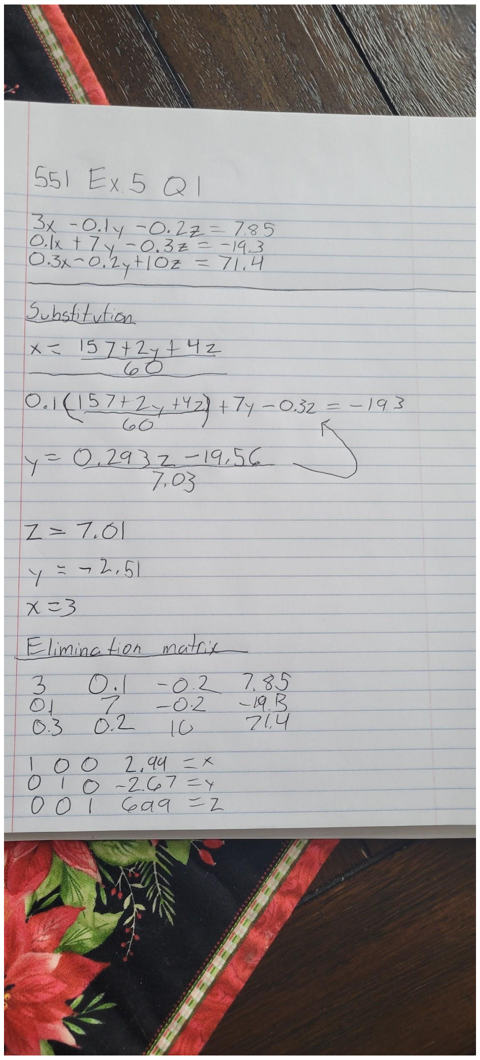
CSCI 551 Austin Dollar

Question 1

Solve the following equations:
$3x - 0.1y - 0.2z = 7.85$
$0.1x + 7y - 0.3z = -19.3$
$0.3x - 0.2y + 10z = 71.4$

[5 pts] By successive substitution, showing your work, with verification.
[5 pts] By elimination with an augmented matrix, showing your work, with verification.

Picture of solutions below:



551 Ex.5 Q1

$3x - 0.1y - 0.2z = 7.85$
$0.1x + 7y - 0.3z = -19.3$
$0.3x - 0.2y + 10z = 71.4$

Substitution

$x = \frac{157 + 2y + 4z}{60}$

$0.1\left(\frac{157 + 2y + 4z}{60}\right) + 7y - 0.3z = -19.3$

$y = \frac{0.293z - 19.56}{7.03}$

$z = 7.01$

$y = -2.51$

$x = 3$

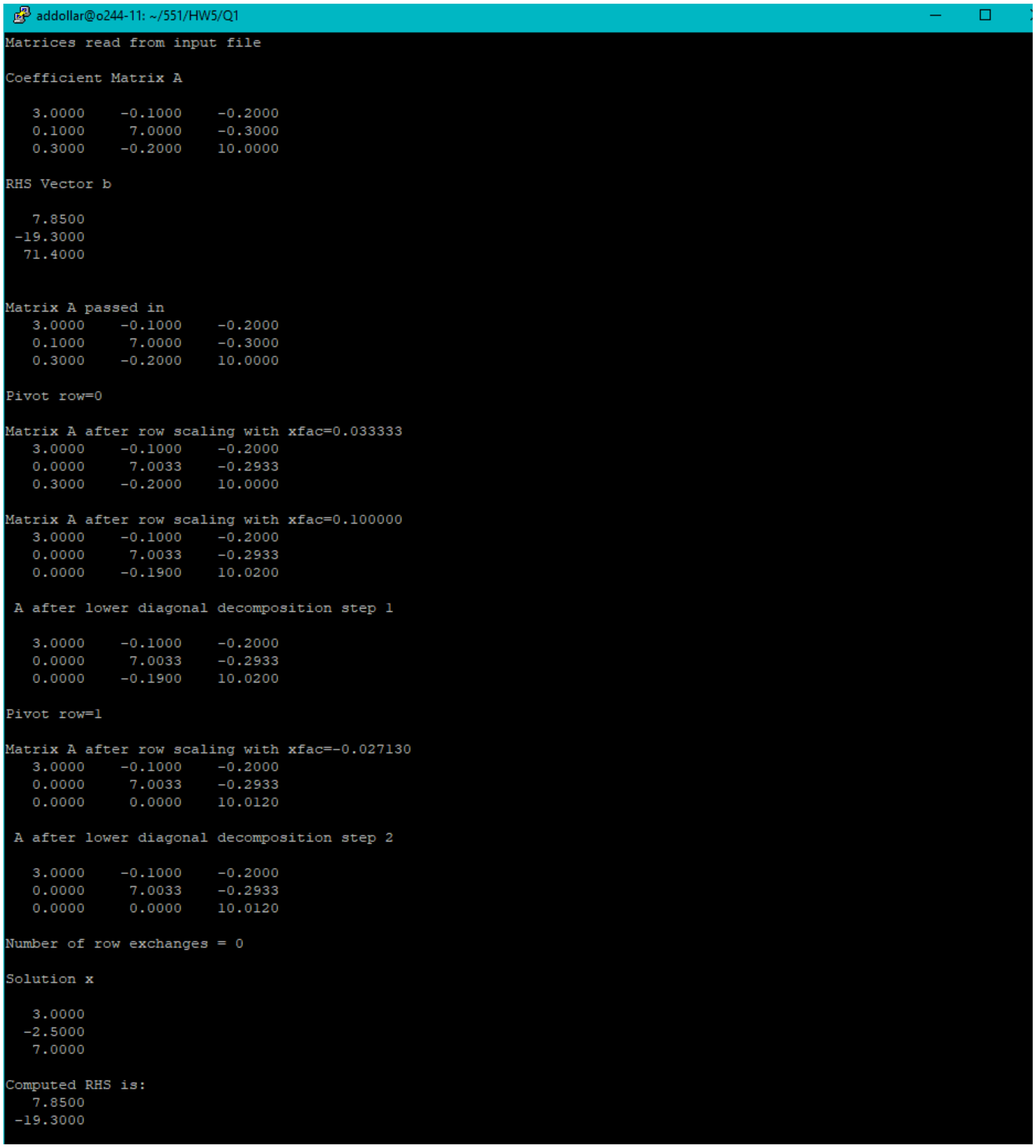
Elimination matrix

$$\begin{matrix} 3 & 0.1 & -0.2 & 7.85 \\ 0.1 & 7 & -0.2 & -19.3 \\ 0.3 & 0.2 & 10 & 71.4 \end{matrix}$$

$$\begin{matrix} 1 & 0 & 0 & 2.99 = x \\ 0 & 1 & 0 & -2.67 = y \\ 0 & 0 & 1 & 6.99 = z \end{matrix}$$

[5 pts] Use a C/C++ program to automate row operations (e.g. gewpp.c) and note any differences from your solution in terms of time, accuracy, and precision.

A screenshot of the output:



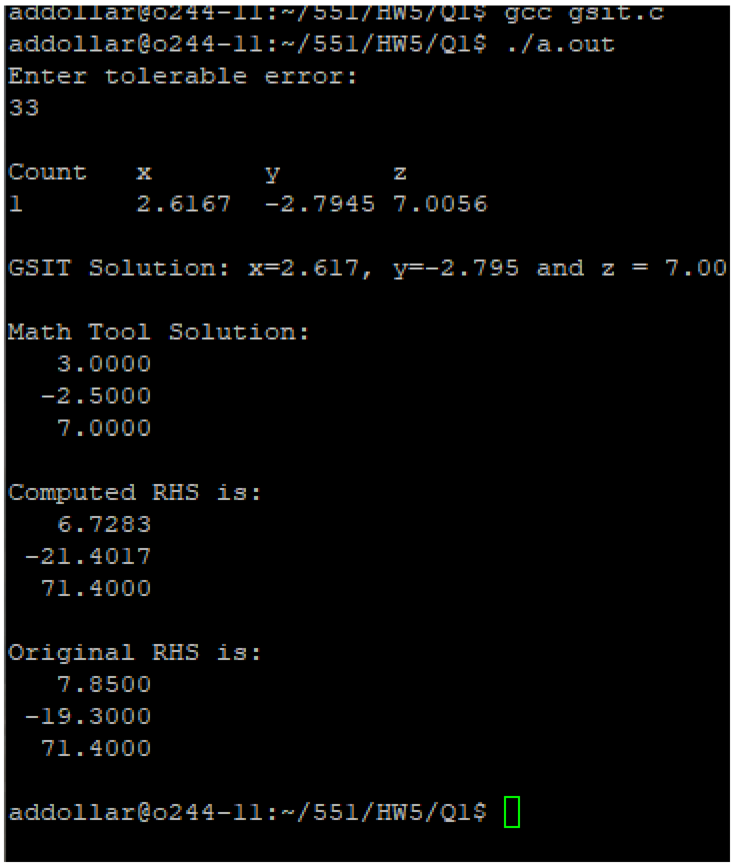
addollar@o244-11: ~/551/HW5/Q1

```
Matrices read from input file

Coefficient Matrix A

   3.0000    -0.1000    -0.2000
   0.1000     7.0000    -0.3000
   0.3000    -0.2000    10.0000

RHS Vector b

   7.8500
 -19.3000
  71.4000


Matrix A passed in
   3.0000    -0.1000    -0.2000
   0.1000     7.0000    -0.3000
   0.3000    -0.2000    10.0000

Pivot row=0

Matrix A after row scaling with xfac=0.033333
   3.0000    -0.1000    -0.2000
   0.0000     7.0033    -0.2933
   0.3000    -0.2000    10.0000

Matrix A after row scaling with xfac=0.100000
   3.0000    -0.1000    -0.2000
   0.0000     7.0033    -0.2933
   0.0000    -0.1900    10.0200

 A after lower diagonal decomposition step 1

   3.0000    -0.1000    -0.2000
   0.0000     7.0033    -0.2933
   0.0000    -0.1900    10.0200

Pivot row=1

Matrix A after row scaling with xfac=-0.027130
   3.0000    -0.1000    -0.2000
   0.0000     7.0033    -0.2933
   0.0000     0.0000    10.0120

 A after lower diagonal decomposition step 2

   3.0000    -0.1000    -0.2000
   0.0000     7.0033    -0.2933
   0.0000     0.0000    10.0120

Number of row exchanges = 0

Solution x

   3.0000
  -2.5000
   7.0000

Computed RHS is:
   7.8500
 -19.3000
```

In terms of time, accuracy, and precision, the computerized version is obviously much faster in all accounts. The code finished the system in seconds rather than minutes, and had much more precision than achievable via a handwritten solution.

[5 pts] Use a C/C++ program to automate iteration (e.g. gsit.c) and note any differences from your solution in terms of convergence, time, accuracy, and precision. Be sure to organize equations in diagonally dominant form.

A screenshot of the output:



```
addollar@o244-11:~/551/HW5/Q1$ gcc gsit.c
addollar@o244-11:~/551/HW5/Q1$ ./a.out
Enter tolerable error:
33

Count   x       y       z
1       2.6167  -2.7945 7.0056

GSIT Solution: x=2.617, y=-2.795 and z = 7.00

Math Tool Solution:
   3.0000
  -2.5000
   7.0000

Computed RHS is:
   6.7283
 -21.4017
  71.4000

Original RHS is:
   7.8500
 -19.3000
  71.4000

addollar@o244-11:~/551/HW5/Q1$
```

In terms of time, accuracy, and precision, the computerized version is obviously much faster in all accounts. The code finished the system in seconds rather than minutes, and had much more precision than achievable via a handwritten solution.

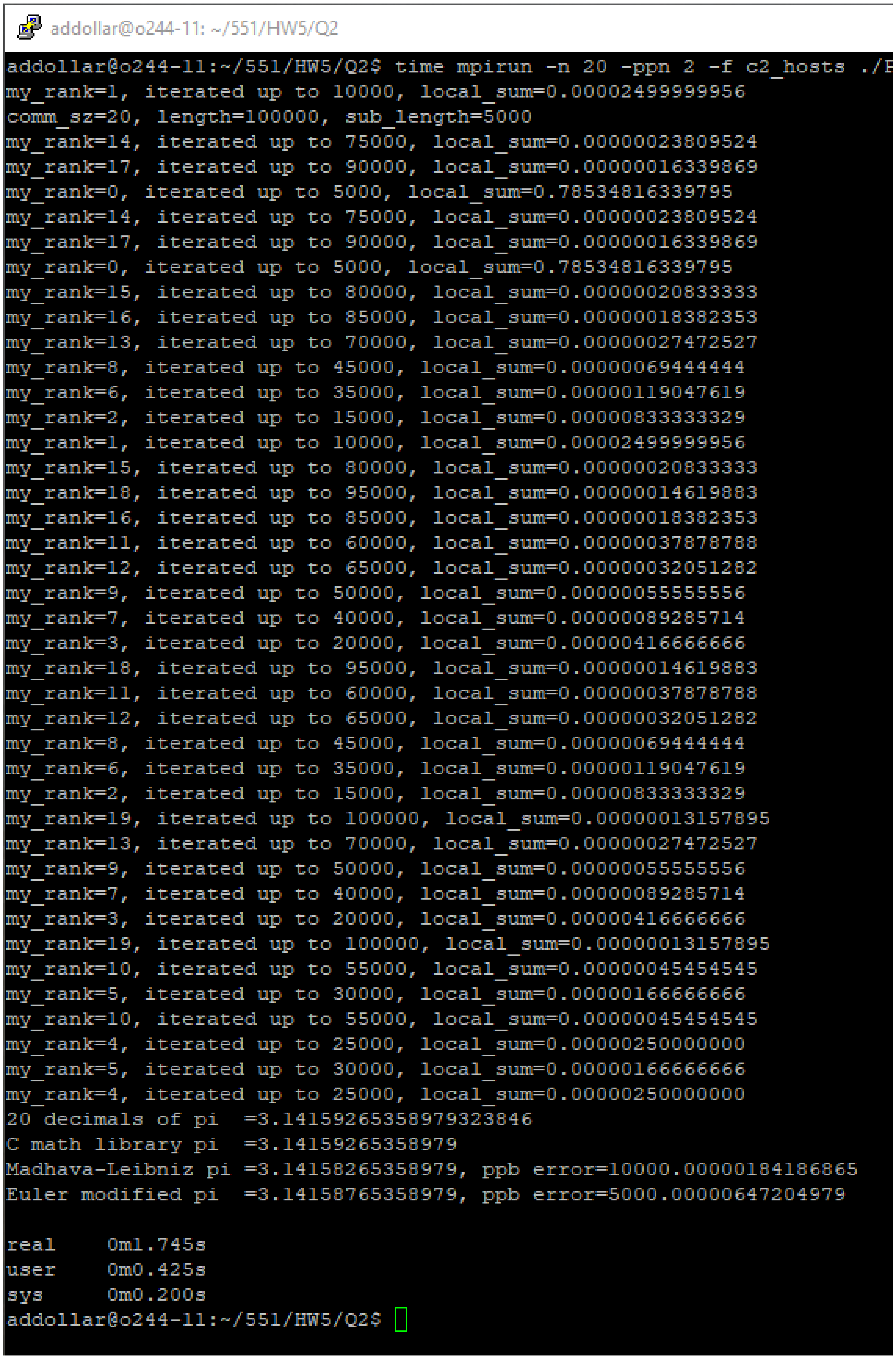
Question 2

[20 points total] Re-implement this MPI program (piseriesreduce.c) that estimates the value of Π by two well-known series noted in the MPI code, now with OpenMP (see pp. 229 in Pacheco).

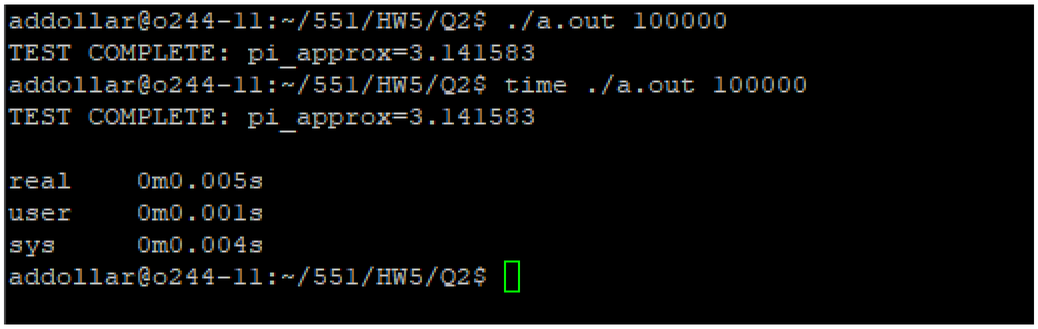
The OpenMp version of this code is included in the corresponding Question 2 folder, titled “omppi.c”

Using MPI_Wtime and the POSIX clock_gettime, compare the two implementations for rank=2 for 100,000 up to 1,000,000 iterations in terms of speed and note any differences in accuracy or precision

First we can compare for 100,000 iterations. Execution of the MPI Code is below, followed by the OpenMp version of the code:


addollar@o244-11: ~/551/HW5/Q2

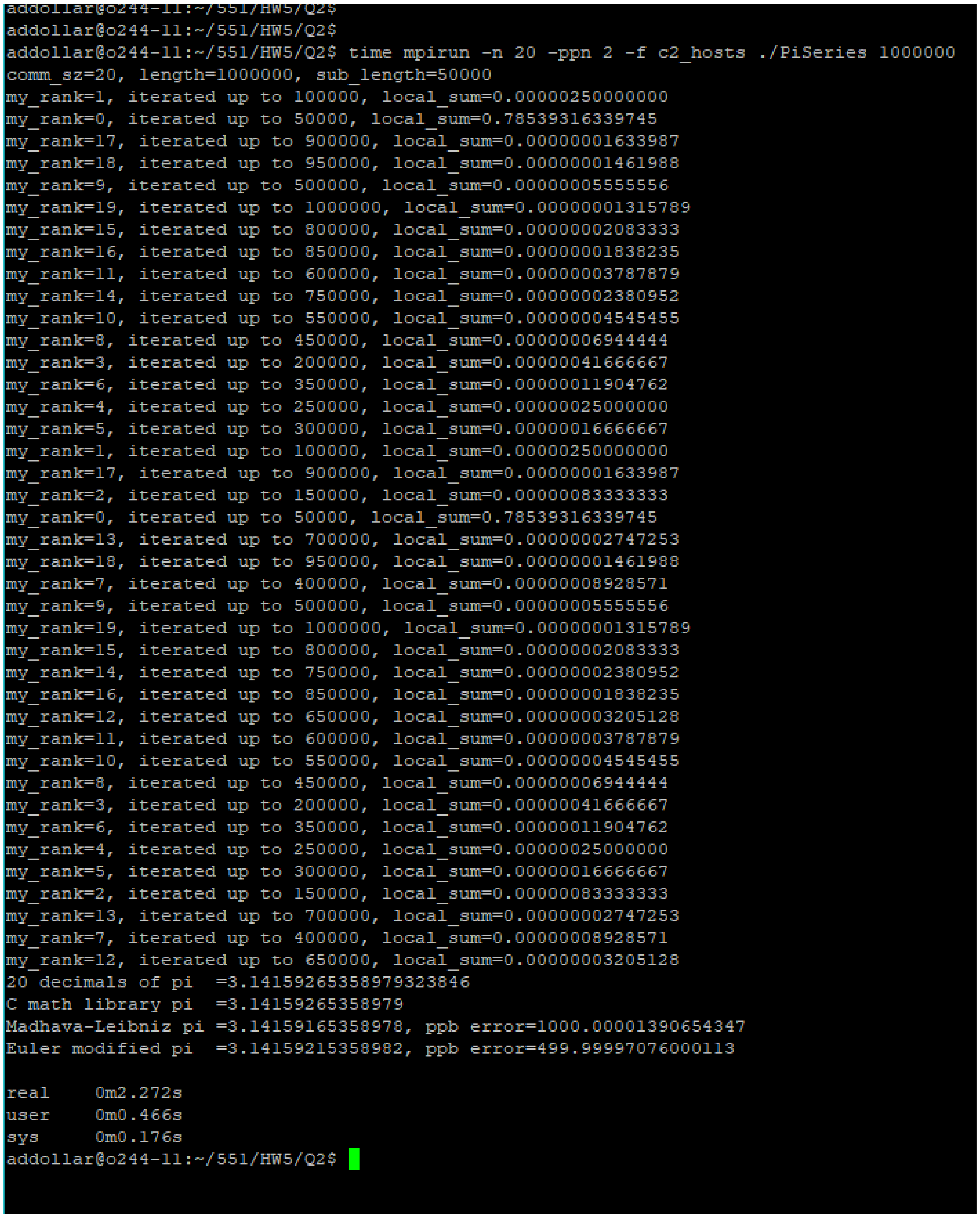
```
addollar@o244-11:~/551/HW5/Q2$ time mpirun -n 20 -ppn 2 -f c2_hosts ./P
my_rank=1, iterated up to 10000, local_sum=0.00002499999956
comm_sz=20, length=100000, sub_length=5000
my_rank=14, iterated up to 75000, local_sum=0.00000023809524
my_rank=17, iterated up to 90000, local_sum=0.00000016339869
my_rank=0, iterated up to 5000, local_sum=0.78534816339795
my_rank=14, iterated up to 75000, local_sum=0.00000023809524
my_rank=17, iterated up to 90000, local_sum=0.00000016339869
my_rank=0, iterated up to 5000, local_sum=0.78534816339795
my_rank=15, iterated up to 80000, local_sum=0.00000020833333
my_rank=16, iterated up to 85000, local_sum=0.00000018382353
my_rank=13, iterated up to 70000, local_sum=0.00000027472527
my_rank=8, iterated up to 45000, local_sum=0.00000069444444
my_rank=6, iterated up to 35000, local_sum=0.00000119047619
my_rank=2, iterated up to 15000, local_sum=0.00000833333329
my_rank=1, iterated up to 10000, local_sum=0.00002499999956
my_rank=15, iterated up to 80000, local_sum=0.00000020833333
my_rank=18, iterated up to 95000, local_sum=0.00000014619883
my_rank=16, iterated up to 85000, local_sum=0.00000018382353
my_rank=11, iterated up to 60000, local_sum=0.00000037878788
my_rank=12, iterated up to 65000, local_sum=0.00000032051282
my_rank=9, iterated up to 50000, local_sum=0.00000055555556
my_rank=7, iterated up to 40000, local_sum=0.00000089285714
my_rank=3, iterated up to 20000, local_sum=0.00000416666666
my_rank=18, iterated up to 95000, local_sum=0.00000014619883
my_rank=11, iterated up to 60000, local_sum=0.00000037878788
my_rank=12, iterated up to 65000, local_sum=0.00000032051282
my_rank=8, iterated up to 45000, local_sum=0.00000069444444
my_rank=6, iterated up to 35000, local_sum=0.00000119047619
my_rank=2, iterated up to 15000, local_sum=0.00000833333329
my_rank=19, iterated up to 100000, local_sum=0.00000013157895
my_rank=13, iterated up to 70000, local_sum=0.00000027472527
my_rank=9, iterated up to 50000, local_sum=0.00000055555556
my_rank=7, iterated up to 40000, local_sum=0.00000089285714
my_rank=3, iterated up to 20000, local_sum=0.00000416666666
my_rank=19, iterated up to 100000, local_sum=0.00000013157895
my_rank=10, iterated up to 55000, local_sum=0.00000045454545
my_rank=5, iterated up to 30000, local_sum=0.00000166666666
my_rank=10, iterated up to 55000, local_sum=0.00000045454545
my_rank=4, iterated up to 25000, local_sum=0.00000250000000
my_rank=5, iterated up to 30000, local_sum=0.00000166666666
my_rank=4, iterated up to 25000, local_sum=0.00000250000000
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.14158265358979, ppb error=10000.00000184186865
Euler modified pi  =3.14158765358979, ppb error=5000.00000647204979

real    0m1.745s
user    0m0.425s
sys     0m0.200s
addollar@o244-11:~/551/HW5/Q2$
```



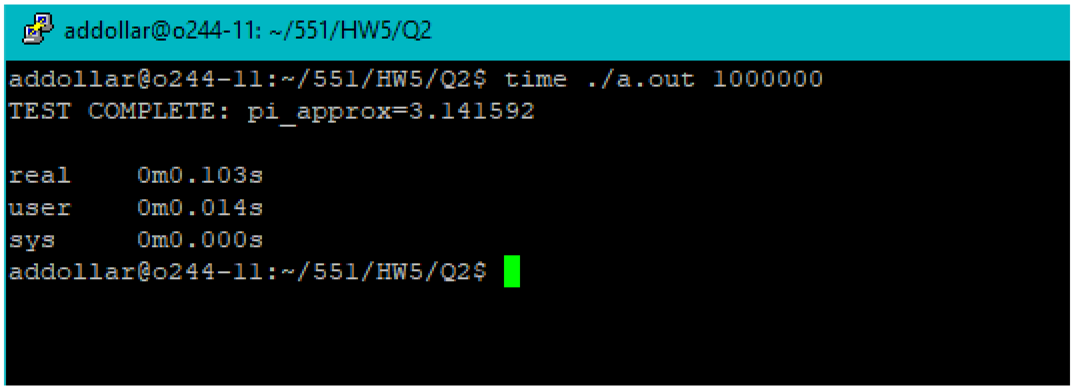
```
addollar@o244-11:~/551/HW5/Q2$ ./a.out 100000
TEST COMPLETE: pi_approx=3.141583
addollar@o244-11:~/551/HW5/Q2$ time ./a.out 100000
TEST COMPLETE: pi_approx=3.141583

real    0m0.005s
user    0m0.001s
sys     0m0.004s
addollar@o244-11:~/551/HW5/Q2$ 
```

As you can see, when we look at 100,000 iterations, our speed is much faster for the openmp value, however, we do not have nearly the accuracy or precision that we do with MPI. With MPI, we have vastly more levels of precision, as well as being more accurate. This is easily determined just by looking at the value calculated via OpenMp, as it loses accuracy at 7 digits of precision (as a double), while MPI maintains precision for its full 15 digits of counted precision.

Now, let’s look at 1,000,000 iterations. Again, below is the MPI code followed by the openmp code:



```
addollar@o244-11:~/551/HW5/Q2$
addollar@o244-11:~/551/HW5/Q2$
addollar@o244-11:~/551/HW5/Q2$ time mpirun -n 20 -ppn 2 -f c2_hosts ./PiSeries 1000000
comm_sz=20, length=1000000, sub_length=50000
my_rank=1, iterated up to 100000, local_sum=0.00000250000000
my_rank=0, iterated up to 50000, local_sum=0.78539316339745
my_rank=17, iterated up to 900000, local_sum=0.00000001633987
my_rank=18, iterated up to 950000, local_sum=0.00000001461988
my_rank=9, iterated up to 500000, local_sum=0.00000005555556
my_rank=19, iterated up to 1000000, local_sum=0.00000001315789
my_rank=15, iterated up to 800000, local_sum=0.00000002083333
my_rank=16, iterated up to 850000, local_sum=0.00000001838235
my_rank=11, iterated up to 600000, local_sum=0.00000003787879
my_rank=14, iterated up to 750000, local_sum=0.00000002380952
my_rank=10, iterated up to 550000, local_sum=0.00000004545455
my_rank=8, iterated up to 450000, local_sum=0.00000006944444
my_rank=3, iterated up to 200000, local_sum=0.00000041666667
my_rank=6, iterated up to 350000, local_sum=0.00000011904762
my_rank=4, iterated up to 250000, local_sum=0.00000025000000
my_rank=5, iterated up to 300000, local_sum=0.00000016666667
my_rank=1, iterated up to 100000, local_sum=0.00000250000000
my_rank=17, iterated up to 900000, local_sum=0.00000001633987
my_rank=2, iterated up to 150000, local_sum=0.00000083333333
my_rank=0, iterated up to 50000, local_sum=0.78539316339745
my_rank=13, iterated up to 700000, local_sum=0.00000002747253
my_rank=18, iterated up to 950000, local_sum=0.00000001461988
my_rank=7, iterated up to 400000, local_sum=0.00000008928571
my_rank=9, iterated up to 500000, local_sum=0.00000005555556
my_rank=19, iterated up to 1000000, local_sum=0.00000001315789
my_rank=15, iterated up to 800000, local_sum=0.00000002083333
my_rank=14, iterated up to 750000, local_sum=0.00000002380952
my_rank=16, iterated up to 850000, local_sum=0.00000001838235
my_rank=12, iterated up to 650000, local_sum=0.00000003205128
my_rank=11, iterated up to 600000, local_sum=0.00000003787879
my_rank=10, iterated up to 550000, local_sum=0.00000004545455
my_rank=8, iterated up to 450000, local_sum=0.00000006944444
my_rank=3, iterated up to 200000, local_sum=0.00000041666667
my_rank=6, iterated up to 350000, local_sum=0.00000011904762
my_rank=4, iterated up to 250000, local_sum=0.00000025000000
my_rank=5, iterated up to 300000, local_sum=0.00000016666667
my_rank=2, iterated up to 150000, local_sum=0.00000083333333
my_rank=13, iterated up to 700000, local_sum=0.00000002747253
my_rank=7, iterated up to 400000, local_sum=0.00000008928571
my_rank=12, iterated up to 650000, local_sum=0.00000003205128
20 decimals of pi  =3.14159265358979323846
C math library pi  =3.14159265358979
Madhava-Leibniz pi =3.14159165358978, ppb error=1000.00001390654347
Euler modified pi  =3.14159215358982, ppb error=499.99997076000113

real    0m2.272s
user    0m0.466s
sys     0m0.176s
addollar@o244-11:~/551/HW5/Q2$
```



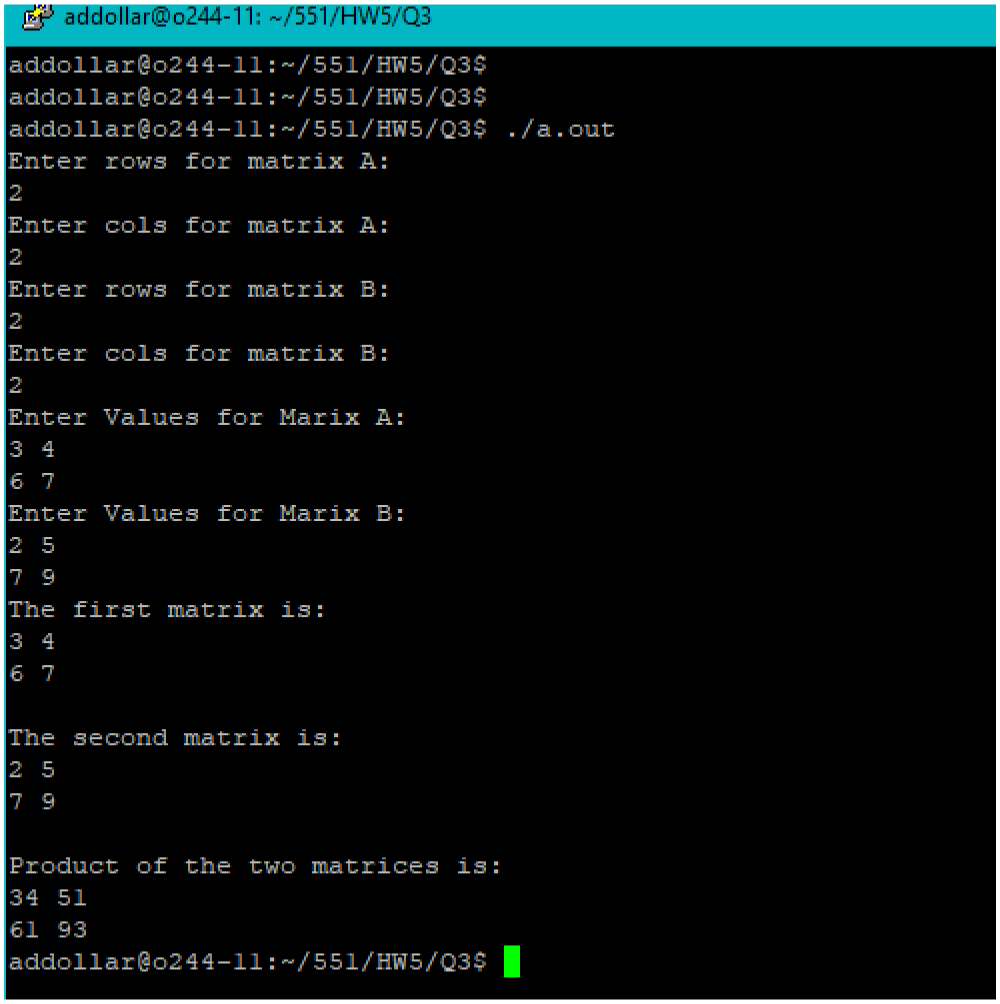
```
addollar@o244-11:~/551/HW5/Q2$ time ./a.out 1000000
TEST COMPLETE: pi_approx=3.141592

real    0m0.103s
user    0m0.014s
sys     0m0.000s
addollar@o244-11:~/551/HW5/Q2$
```

For this level of scaling, we have slightly different results. Our MPI Code is again significantly slower, likely due to network speeds rather than software speeds. And again, we have significantly less units of precision, with only 7 digits as opposed to 15. However, the plus side of this upscaling is that our 7 units of precision that we were able to maintain with OpenMp are all accurate.

Question 3

[10 pts] Multiply any dimension square matrix from 2x2 up to NxN by another matrix of the same dimensions, or by a column vector of dimension 1 up to N. Make your program interactive, but such that you can test it by re-directing input to it for faster test automation.

The Serial Implementation for matrix multiplication is included in the corresponding Q3 folder, labeled "Serial_Mult.cpp" , and can be compiled by g++. A sample of output is included belowfor a 2X2 matrix:

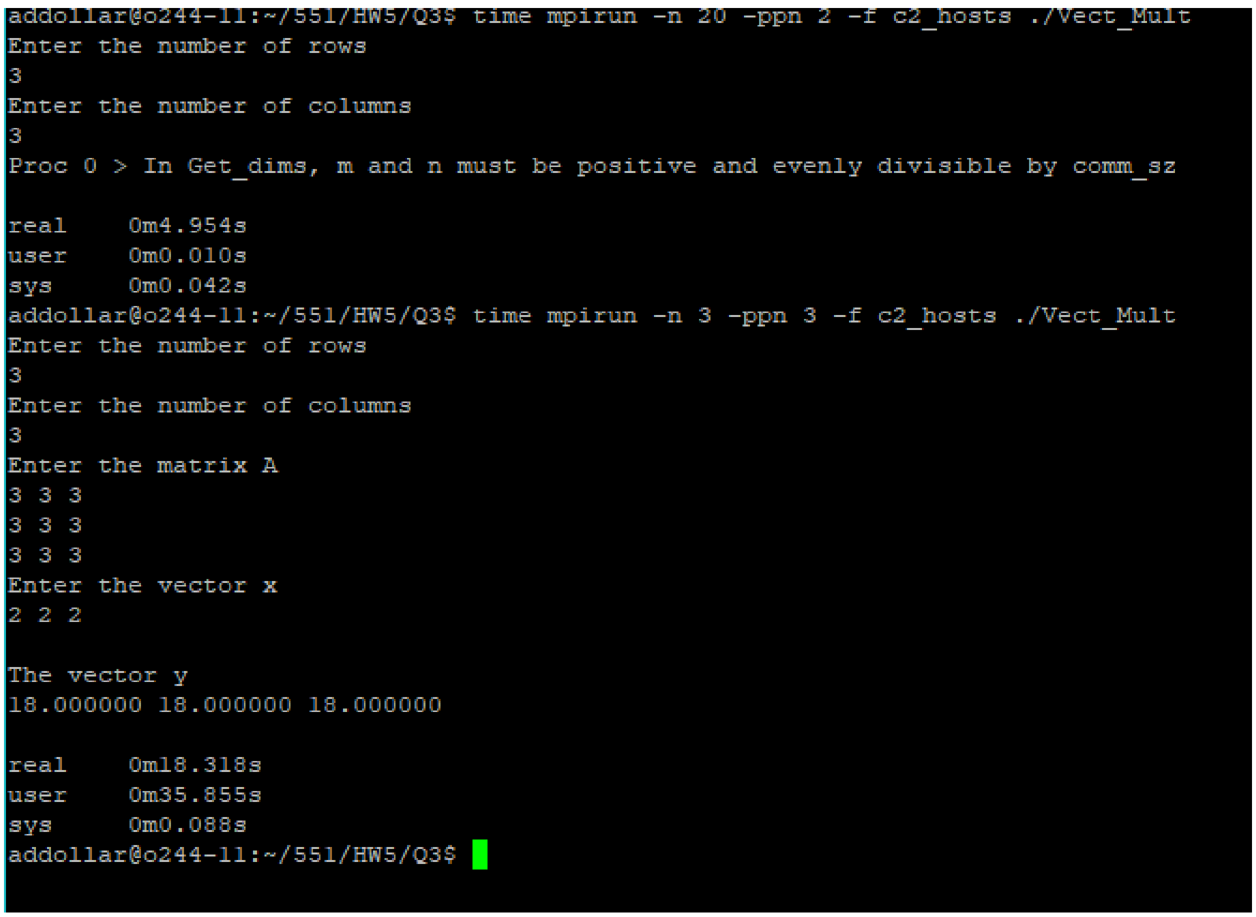


```
addollar@o244-11: ~/551/HW5/Q3
addollar@o244-11:~/551/HW5/Q3$
addollar@o244-11:~/551/HW5/Q3$
addollar@o244-11:~/551/HW5/Q3$ ./a.out
Enter rows for matrix A:
2
Enter cols for matrix A:
2
Enter rows for matrix B:
2
Enter cols for matrix B:
2
Enter Values for Marix A:
3 4
6 7
Enter Values for Marix B:
2 5
7 9
The first matrix is:
3 4
6 7

The second matrix is:
2 5
7 9

Product of the two matrices is:
34 51
61 93
addollar@o244-11:~/551/HW5/Q3$
```

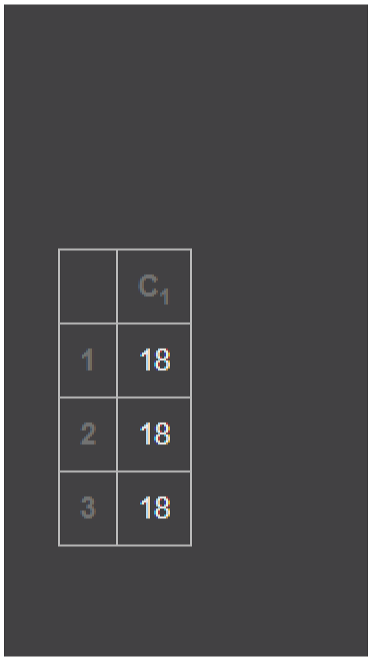
[10 pts] Create an OpenMP or MPI version of your program above (your pick) that allows for multi-process and multi-node parallelism to speed up the square matrix multiplication above as well as the matrix and column vector multiplication. Test it for functional correctness with a 3x3 matrix and dimension 3 vectors for simplicity. However, please also test it for scaling with a 10x10, 100x100 and larger NxN for speed-up achieved with your parallel implementation. Report on the speed-up achieved for 3x3, 10x10, 100x100, and even 1000x1000 if this is feasible.

I could not get full matrix multiplication working for the MPI implementation, so when comparing the MPI starter code with my serial version, I simply just used 1 column for my column value to simulate a column vector.

When testing for correctness for a 3x3 matrix, we can confirm that the starter code is indeed functionally correct, as the output is the same as an online matrix calculator:



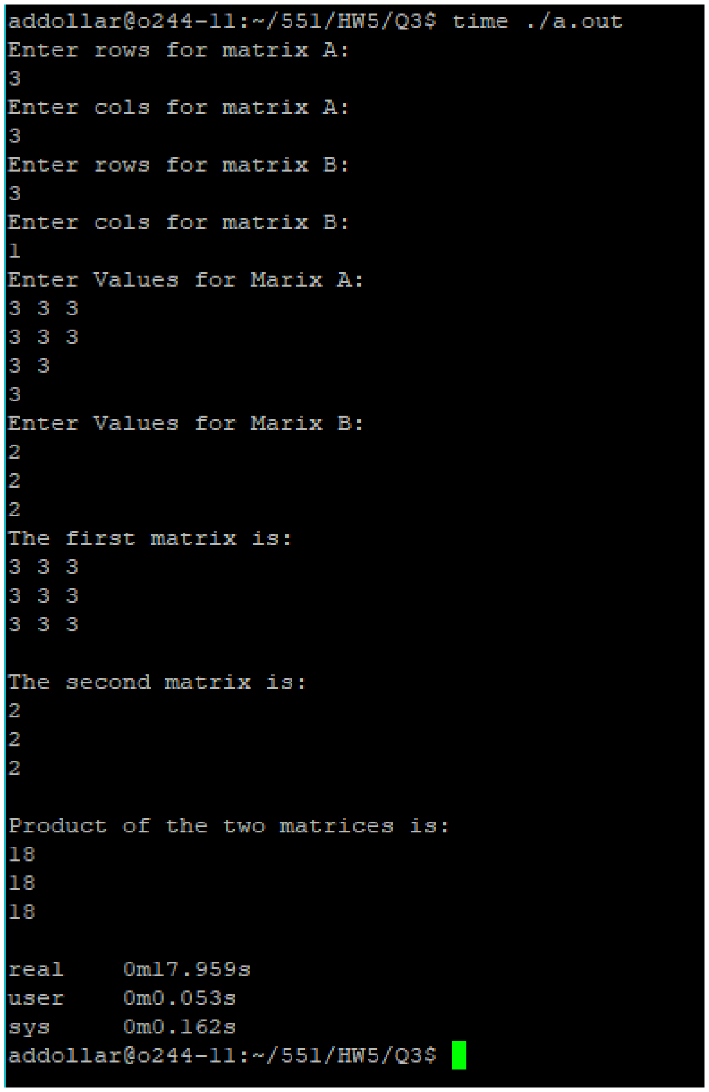
```
addollar@o244-11:~/551/HW5/Q3$ time mpirun -n 20 -ppn 2 -f c2_hosts ./Vect_Mult
Enter the number of rows
3
Enter the number of columns
3
Proc 0 > In Get_dims, m and n must be positive and evenly divisible by comm_sz

real    0m4.954s
user    0m0.010s
sys     0m0.042s
addollar@o244-11:~/551/HW5/Q3$ time mpirun -n 3 -ppn 3 -f c2_hosts ./Vect_Mult
Enter the number of rows
3
Enter the number of columns
3
Enter the matrix A
3 3 3
3 3 3
3 3 3
Enter the vector x
2 2 2

The vector y
18.000000 18.000000 18.000000

real    0m18.318s
user    0m35.855s
sys     0m0.088s
addollar@o244-11:~/551/HW5/Q3$
```



	C_1
1	18
2	18
3	18

As you can see in the above screenshots, the code is functionally correct, as it matches the output of an online vector matrix calculator.

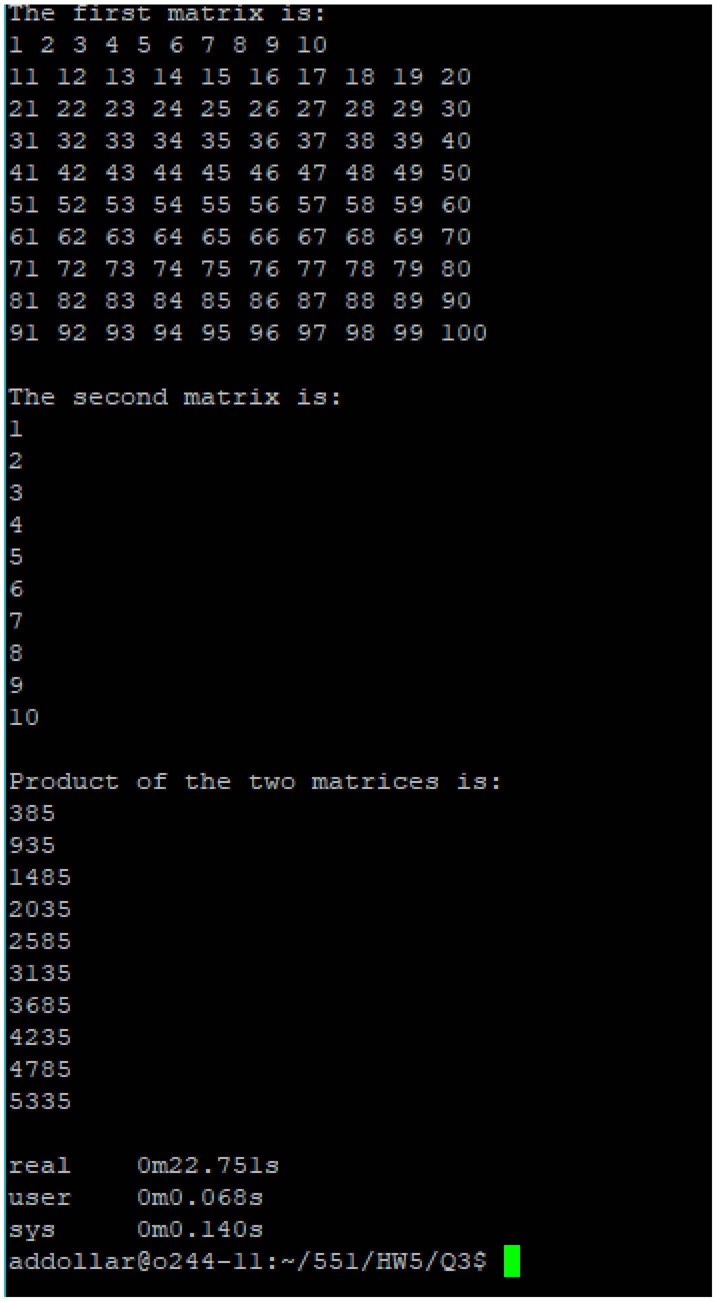
Also, you can see that we timed the MPI implementation, with a system time of 0.88s. We can run the serial version of this same multiplication operation and compare the time of execution:



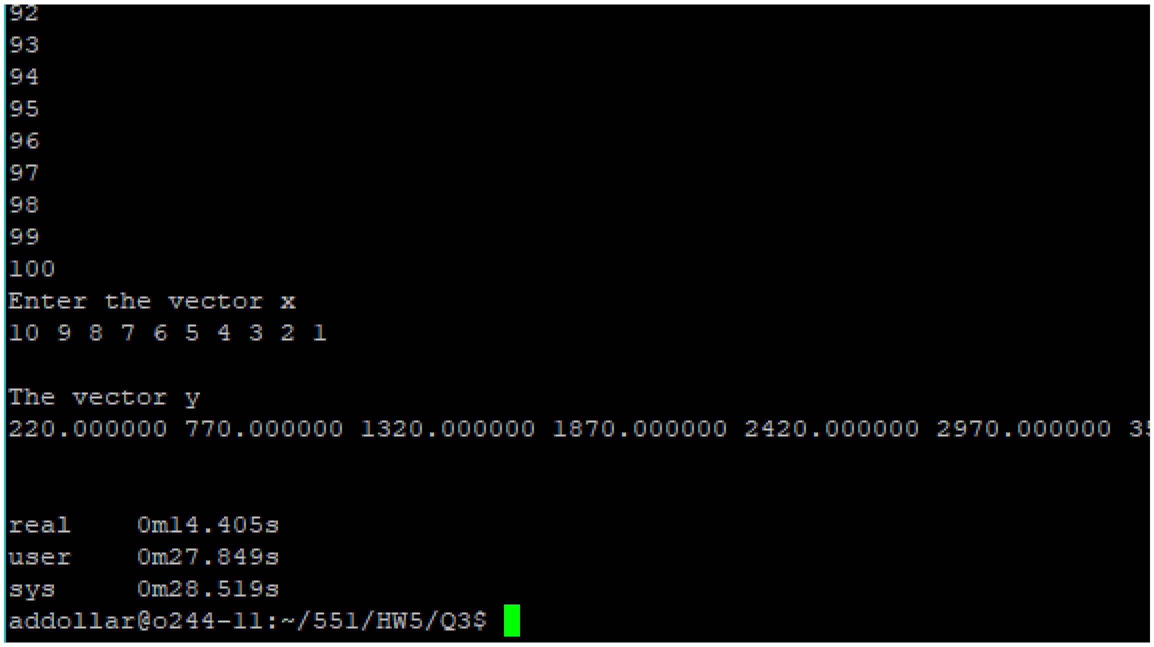
```
addollar@o244-11:~/551/HW5/Q3$ time ./a.out
Enter rows for matrix A:
3
Enter cols for matrix A:
3
Enter rows for matrix B:
3
Enter cols for matrix B:
1
Enter Values for Marix A:
3 3 3
3 3 3
3 3
3
Enter Values for Marix B:
2
2
2
The first matrix is:
3 3 3
3 3 3
3 3 3

The second matrix is:
2
2
2

Product of the two matrices is:
18
18
18

real    0m17.959s
user    0m0.053s
sys     0m0.162s
addollar@o244-11:~/551/HW5/Q3$
```

As you can see above, the serial system runtime for the same program is .162 s. We can calculate this as a speedup factor of 1.84, which is a pretty decent speedup for a program such as this.

We can test this at scale as well, with a 10x10 matrix as well, with the serial program execution screenshotted below, followed by the parallel version, and then the calculated speedup:



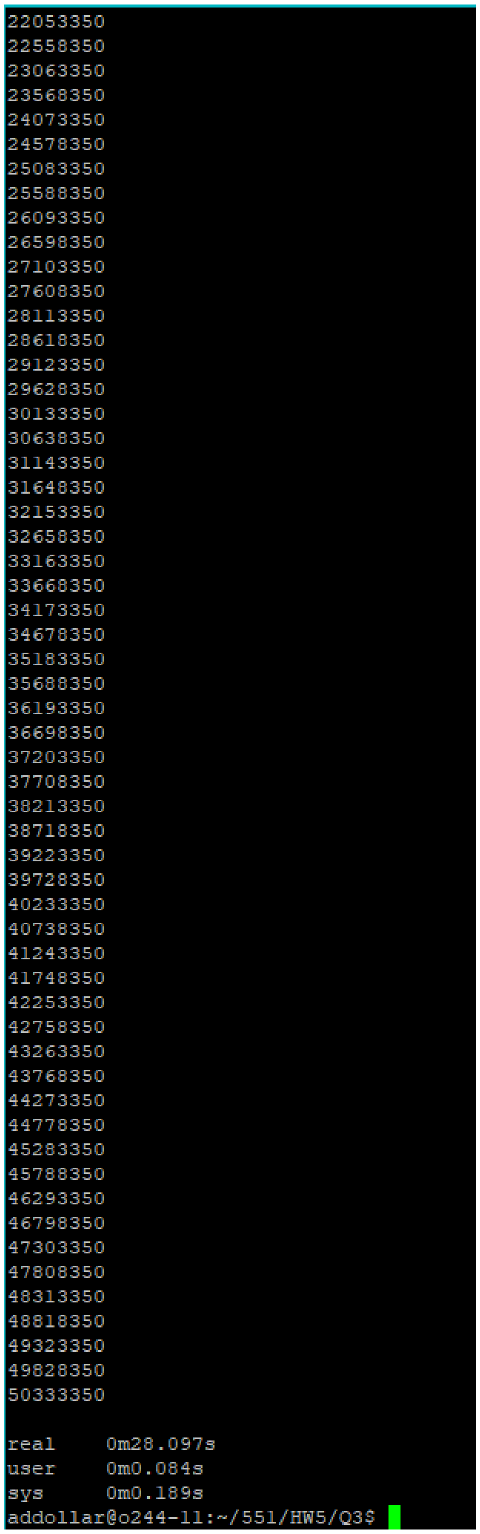
```
The first matrix is:
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100

The second matrix is:
1
2
3
4
5
6
7
8
9
10

Product of the two matrices is:
385
935
1485
2035
2585
3135
3685
4235
4785
5335

real    0m22.751s
user    0m0.068s
sys     0m0.140s
addollar@o244-11:~/551/HW5/Q3$
```



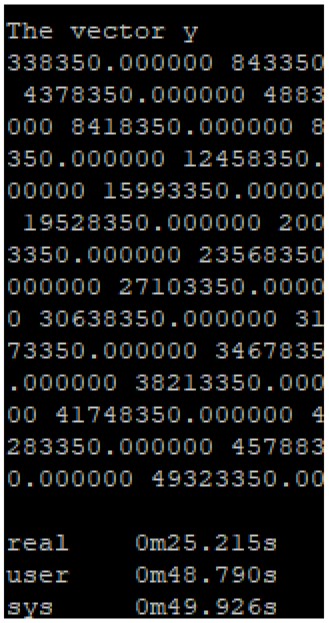
```
92
93
94
95
96
97
98
99
100
Enter the vector x
10 9 8 7 6 5 4 3 2 1

The vector y
220.000000 770.000000 1320.000000 1870.000000 2420.000000 2970.000000 3


real    0m14.405s
user    0m27.849s
sys     0m28.519s
addollar@o244-11:~/551/HW5/Q3$ 
```

Calculated speedup: .140/ 28.519= speedup factor of 0.004.
Surprisingly, the parallel implementation here is slower, perhaps due to network speed with the cluster?

Next, we will do the same with a 100x100 matrix, with the same output format below:



```
22053350
22558350
23063350
23568350
24073350
24578350
25083350
25588350
26093350
26598350
27103350
27608350
28113350
28618350
29123350
29628350
30133350
30638350
31143350
31648350
32153350
32658350
33163350
33668350
34173350
34678350
35183350
35688350
36193350
36698350
37203350
37708350
38213350
38718350
39223350
39728350
40233350
40738350
41243350
41748350
42253350
42758350
43263350
43768350
44273350
44778350
45283350
45788350
46293350
46798350
47303350
47808350
48313350
48818350
49323350
49828350
50333350

real    0m28.097s
user    0m0.084s
sys     0m0.189s
addollar@o244-11:~/551/HW5/Q3$
```



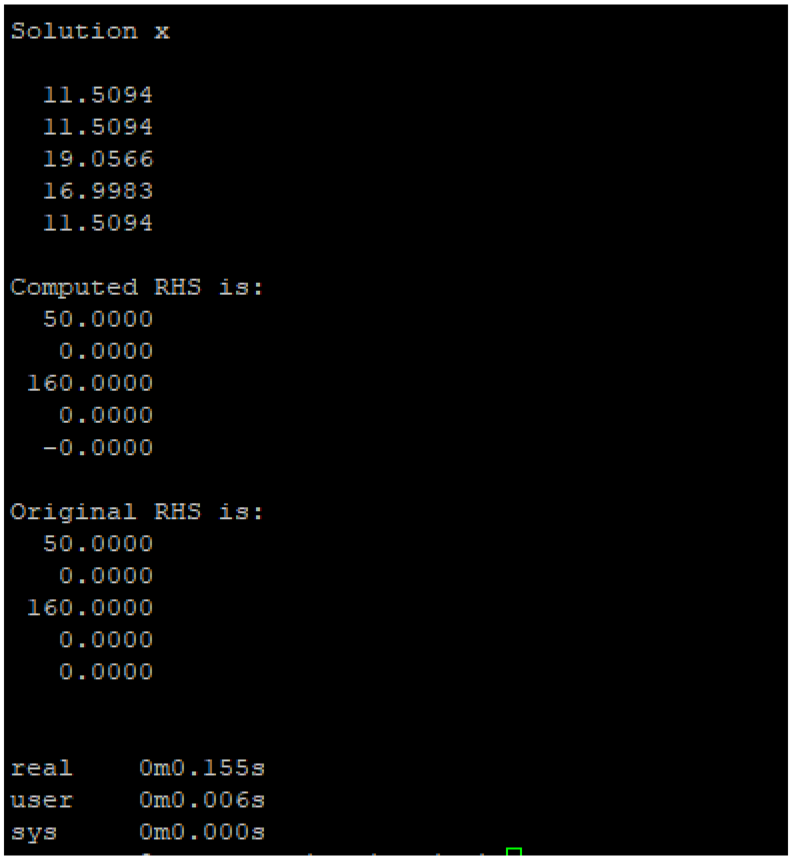
```
The vector y
338350.000000 843350
 4378350.000000 4883
000 8418350.000000 8
350.000000 12458350.
00000 15993350.00000
 19528350.000000 200
3350.000000 23568350
000000 27103350.0000
0 30638350.000000 31
73350.000000 3467835
.000000 38213350.000
00 41748350.000000 4
283350.000000 457883
0.000000 49323350.00

real    0m25.215s
user    0m48.790s
sys     0m49.926s
```

Calculated speedup: .189/49.926= speedup factor of 0.003.
Surprisingly, the parallel implementation here is slower even still, and somehow my serial program, which is just a simple nested for loop, has very minimal slow down at scale? This is worth possibly exploring further in my final parallel program.

Question 4

[20 pts] Solve for the 5 unknown concentrations in the system c1 … c5 with a sequential program.

As instructed via the class announcement, I was able to solve for these via the given gewpp.c code, with output shown below, which is also timed to help calculate speedup in the next part:



```
Solution x

  11.5094
  11.5094
  19.0566
  16.9983
  11.5094

Computed RHS is:
  50.0000
   0.0000
 160.0000
   0.0000
  -0.0000

Original RHS is:
  50.0000
   0.0000
 160.0000
   0.0000
   0.0000


real    0m0.155s
user    0m0.006s
sys     0m0.000s
```

[20 pts] Solve for the 5 unknown concentrations in the system c1 … c5 with an MPI or OpenMP parallel program. Show that the program is equivalent and measure execution with “time” on the command line to determine if it runs any faster.

The parallel implementation via openmp of this program is the file “ParG.c”, however, it is segfaulting and I do not have any more time to debug it, as I have already spent multiple hours doing so, and am already using the three days grace because of family issues that limited the time I was able to allocate to this project. Thank you for understanding.

All corresponding code is included in the attached “HW5.zip” folder