MODIFIED CODE LISTING

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int main( int argc, char
*argv[]) {

    int n, i;
    double PI25DT = 3.14159265358979323846264338327950288419716939937510;
    double pi, h, sum, x;

    int numprocs, myid;
    double startTime, endTime;

   /* Initialize MPI and get number of processes and my number or rank*/
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Status status;

    /* Processor zero sets the number of intervals and starts its clock*/
    if (myid==0)
    {
       n=800000000;
       startTime=MPI_Wtime();
    }
    /* Use MPI_Send and MPI_Recv to share number of intervals to all processes */
    if (myid == 0 ) {
        for (i = 1; i < numprocs; i = i + 1) {
            MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    }

    if (myid != 0) {
        MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }

    /* Calculate the width of intervals */
    h   = 1.0 / (double) n;

    /* Initialize sum */
    sum = 0.0;
    /* Step over each inteval I own */
    for (i = myid+1; i <= n; i += numprocs)
    {
            /* Calculate midpoint of interval */
            x = h * ((double)i - 0.5);
            /* Add rectangle's area = height*width = f(x)*h */
            sum += (4.0/(1.0+x*x))*h;
    }
```

```
    /* Get sum total on processor 0 using MPI_Send and MPI_Recv*/
    if (myid != 0 ) {
        MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }

    /*Use an array to collect results and print the approximate value of pi and runtime*/
    double sumArray[numprocs - 1];
    if (myid == 0 ) {
        for (i = 1; i < numprocs; i += 1) {
            MPI_Recv(&sumArray[i-1], 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
        }

        for (i = 0; i < numprocs - 1; i += 1) {
           sum += sumArray[i];
        }


       printf("pi is approximately %.16f, Error is %e\n",
                       sum, fabs(sum - PI25DT));
       endTime=MPI_Wtime();
       printf("runtime is=%.16f",endTime-startTime);
    }
    MPI_Finalize();
    return 0;
}
```

ORIGINAL CODE TEST

Speedup Test

Problem Size (n)	Number of Procs (p)	Runtime (t)	Speedup (S)
800000000	1	15.7339279651641846	1
800000000	2	8.2913751602172852	1.898
800000000	4	4.2844271659851074	3.672
800000000	8	2.3002870082855225	6.840
800000000	16	1.1537427902221680	13.637

Table 1: Speed up test (Original Code)

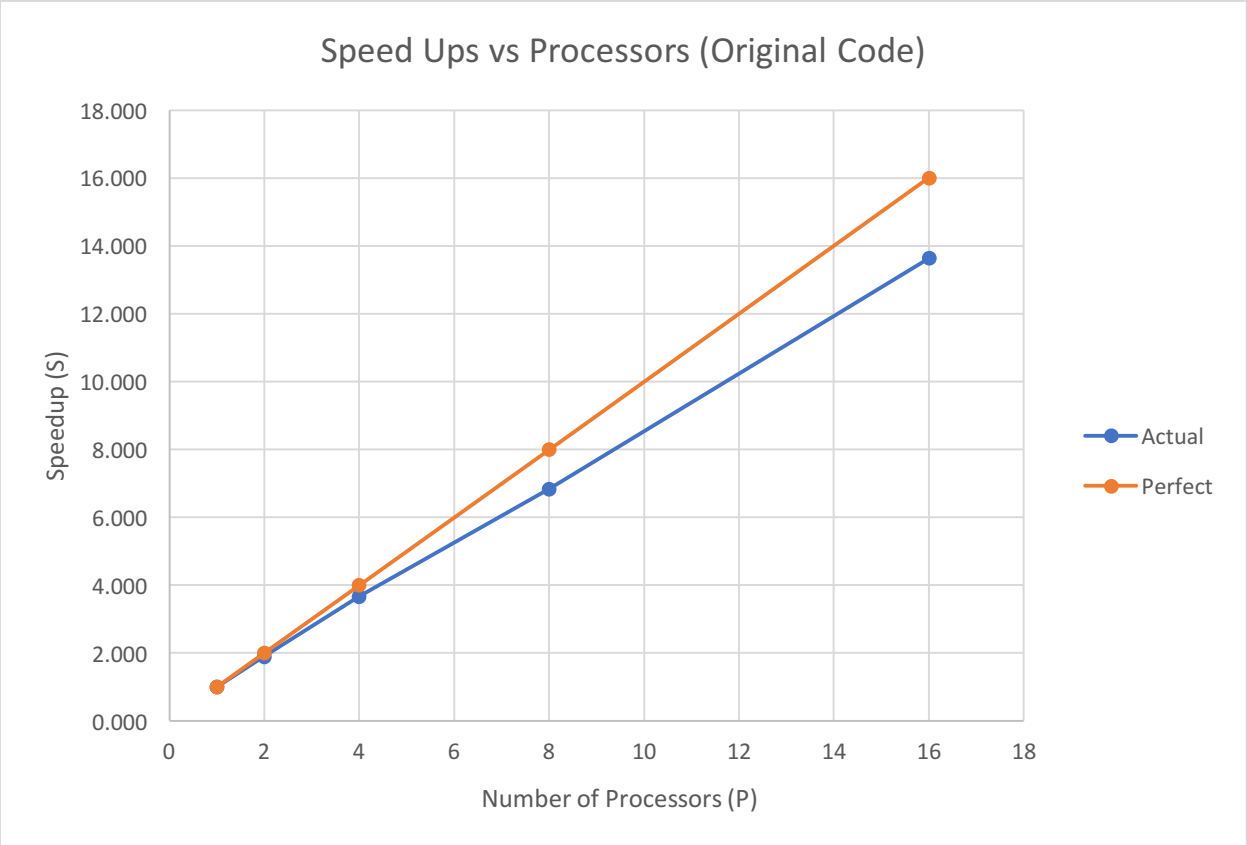

Figure 1: Speed Ups Vs Processors (Original Code)

Scaled Efficiency Test

Problem Size (n)	Number of Procs (p)	Runtime (t)	Efficiency (E)
100000000	1	2.2458469867706299	1
200000000	2	2.3018140792846680	0.976
400000000	4	2.3087430000305176	0.973
800000000	8	2.2567610740661621	0.995
1600000000	16	3.2702661918640137	0.687

Table 2: Scaled Efficiency test (Original Code)

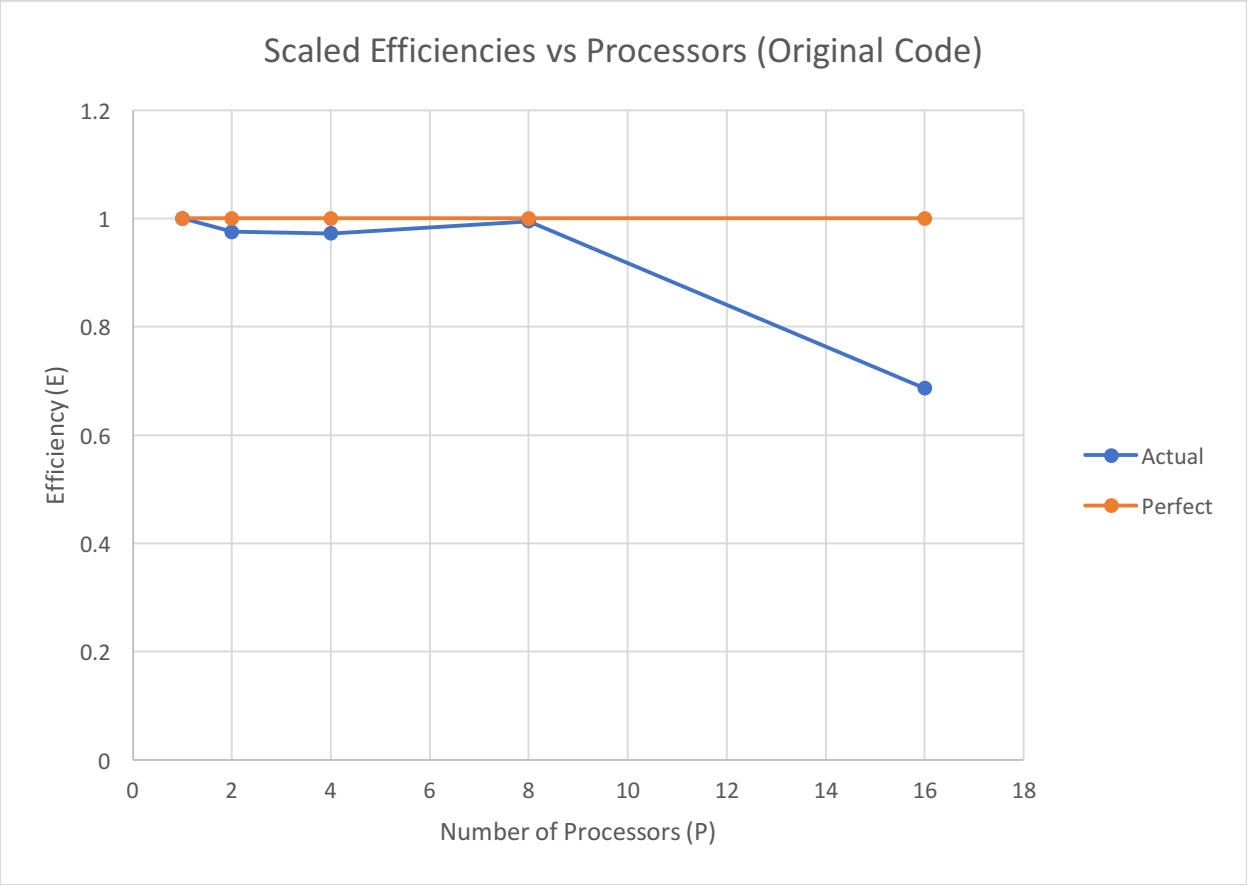

Figure 2: Scaled Efficiencies Vs Processors (Original Code)

MODIFIED CODE TEST

Speedup Test

Problem Size (n)	Number of Procs (p)	Runtime (t)	Speedup (S)
800000000	1	15.7437441349029541	1
800000000	2	9.3191010951995850	1.689
800000000	4	4.2741761207580566	3.683
800000000	8	2.2732160091400146	6.926
800000000	16	1.9014768600463867	8.280

Table 3: Speed up test (Modified Code)

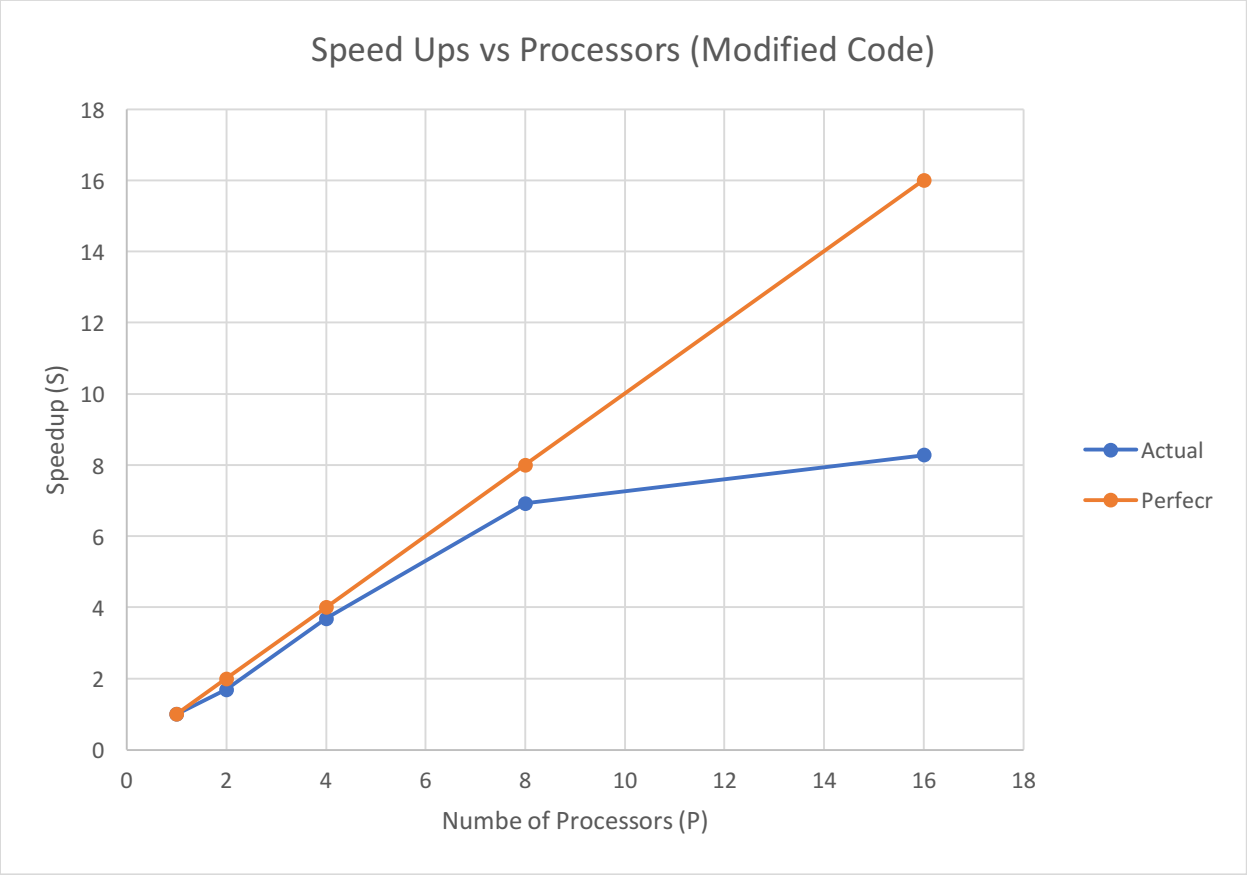

Figure 3: Speed Ups Vs Processors (Modified Code)

Scaled Efficiency Test

Problem Size (n)	Number of Procs (p)	Runtime (t)	Efficiency (E)
100000000	1	2.2430450916290283	1
200000000	2	2.3224040004272461	0.966
400000000	4	2.3050940036773682	0.973
800000000	8	2.2881898880004883	0.980
1600000000	16	3.3292701244354248	0.674

Table 4: Scaled Efficiency test (Modified Code)

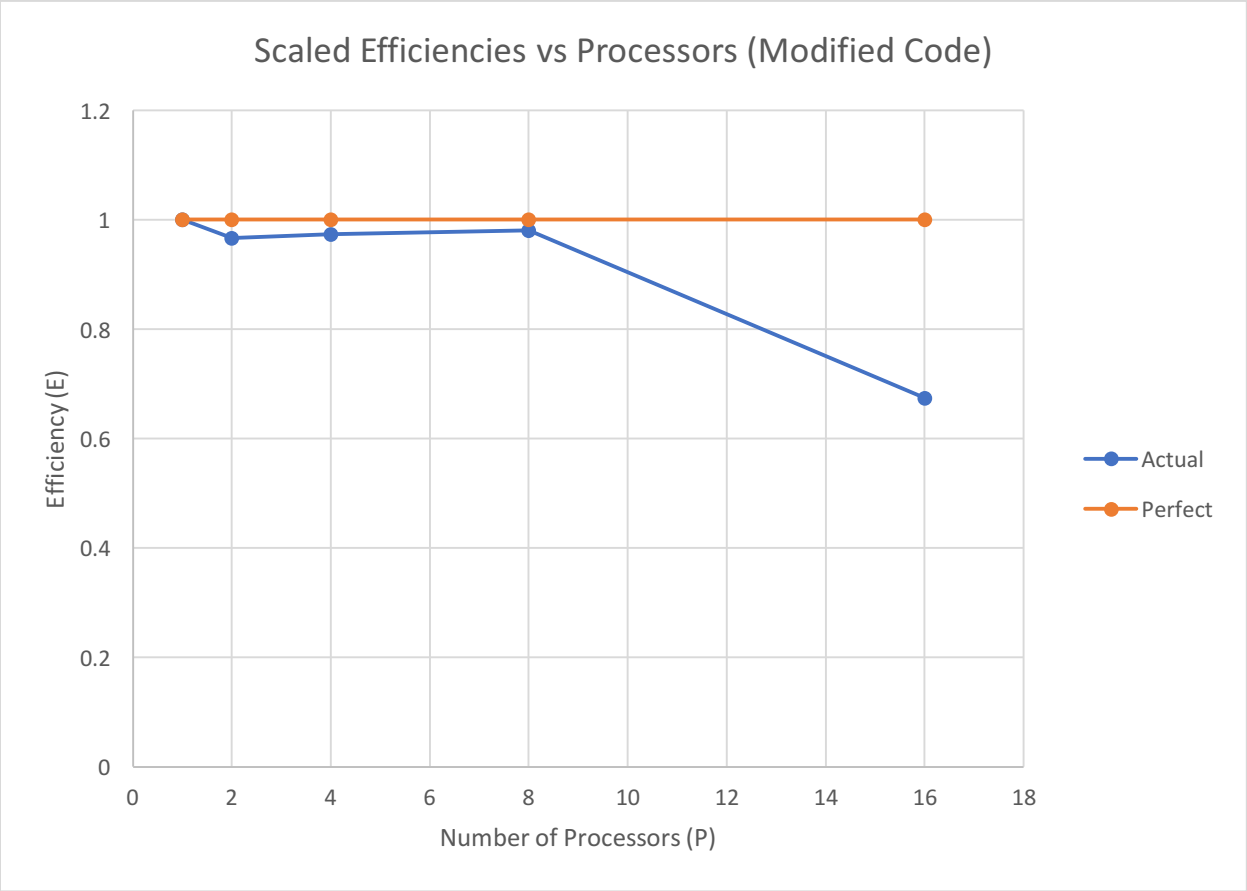

Figure 4: Scaled Efficiencies Vs Processors (Modified Code)

METHOD

To obtain the runtime, an MPI timer (MPI_Wtime()) was used. MPI_Wtime() returns a floating number of seconds representing elapsed wall clock time on the calling processor. The start time of the computation is measured at the beginning of the process. At the end of the process, the current time is measured as well. The computational time taken for the entire process is the difference between the start time and end time.

To reduce the error in the computational time, each test involved 3 -5 trials. The final reported runtime is the average of all the runtime observed for all trials in a given processor. For speedup test (original and modified code), 5 trials were used for all number of processors observed. For scaled efficiency test (original and modified code), 3 trials were used for all number of processors observed.

DISCUSSION OF RESULTS

As shown in the graphs above, both original and modified code performed below perfect speed up {S (N, P) = P}. However, the original code had a better performance than the modified code in terms speedup. As the problem size increases, the difference in speed up across both codes

increases as well. A close observation of the graph also reveals that speed up plot for the original code is shaped linearly and root-function shaped for the modified code. Based on this, we expect to see an increasing amount of speedup as the problem size increases for the original code whereas for the modified code, the speed up is expected to behave as if bounded by a horizontal asymptote.

For scaled efficiency, the graph for both code looks strikingly similar. However, a close look at the actual data reveals a not so significant better efficiency in the original code than the modified code. For both codes, the scaled efficiencies are below perfect efficiency {E (N, P) = 1} and decreases with increasing problem size.