Max Le
ID: 901223283
MTH 5050: Parallel Process
Assignment 2: Using MPI with Blueshark

1 Problem Statement

The goal of this assignment is to modify the original code, where we try to numerically integrate $\dfrac{4}{1+x^2}$ from 0 to 1. This original code was given on Canvas.

The original code has 2 main steps that involve MPI: first, we use MPI BROADCAST to send the problem size from process 0 to the other processes; next,MPI REDUCE is used in order to collect the partial sum from all of the other processes back to process 0 and then sum them up to get our result.

We need to replace the BROADCAST and REDUCE functions by a series of SEND/RECV function. Then, we are going to perform speed up test and scaled efficiency test on both the original code and the modified code.

2 Approach

From assignment 1, we know that our result should be close to $\pi \approx 3.1416....$ Firstly, in order to replace BROADCAST with SEND/RECV, this is the pseudo code of how I am going to do it:

- If process is 0, then send ”n”, the problem size, to other processes. This is done in a loop because we need process 0 to send ”n” to do the SEND function multiple times.
- If process is not 0, then we just need one RECV call. It will receive ”n” from process 0.

In code, this looks something like this:

```
if (rank == 0 ) then
  ! SENDING FROM 0 to OTHERS
  do i = rank+1,nproc-1
        call MPI_SEND(n,1,MPI_INT,i,1,&
                MPI_COMM_WORLD,ierr)
  end do
else
    ! OTHERS RECEIVE
        call MPI_RECV(n,1,MPI_INT,0,1,&
                MPI_COMM_WORLD,status1,ierr)
end if
```

The ”1” after the ”i” in the SEND call and after ”0” in the RECV call is for the tag. This tag means that we are going to perform communications ”1” where 0 send ”n”, and other processes receive ”n” from 0. In contrast, in order to replace REDUCE with SEND/RECV, we have to do the opposite. In pseudocode, it looks something like this:

- If process is 0, we are going to receive the partial sum: ”mypi” in this case. Then we are going to sum up.
- Else, we are all going to send ”mypi” to process 0

In code, this looks something like this:

```
if (rank == 0) then
 ! RECEIVE mypi
 do i = 1,nprocs -1
   call MPI_RECV(mypi_recv,1,MPI_DOUBLE_PRECISION,i,&
                 2,MPI_COMM_WORLD,status1,ierr)
   pi = pi + mypi     ! SUMMING UP
 end do

else
  ! SEND mypi
   call MPI_SEND(mypi,1,MPI_DOUBLE_PRECISION,0,&
                    2,MPI_COMM_WORLD,ierr)
end if
```

The tag is now ”2”. This is to indicate that the communication is to send ”mypi”, not ”n” from before. We are sending/receiving ”mypi”. It is also important to note that if we want to use ”pi” as the final storage, then we need an extra step before this. Basically, if process is 0, then ”pi” is ”mypi” and then if we happen to fall into the above algorithm (have more than 1 process), then ”pi” would get updated.

```
if (rank == 0) then
        pi = mypi   ! STORE mypi into pi at process 0
end if
```

The time is calculated with MPI WTIME, the clock starts when we set problem size for process 0, and end when we finish. In order to get better timing, the code is executed 5 times for each case and for each processor. Then the average time is used to compute speed up/ scale efficiency.

The speed up is calculated as: $\dfrac{\text{Time takes to run fixed N for 1 process}}{\text{Time takes to run fixed N for multiple processes}}$

Meanwhile, the scale efficiency $= \dfrac{\text{Time to run for 1 process at N0 problem size}}{\text{Time to run for P processes at NP problem size}}$.

where the ratio of interval per process is kept to be 100,000,000.

3 Results

3.1 FIXED N

3.1.1 Time results for fixed N = 800,000,000 for different number of processors

nprocs	Time Original(s)	Time Modified(s)
1	16.056570053101	15.779712915421
1	15.873580932617	15.597221851349
1	15.854604005814	15.61110496521
1	15.823721170425	15.476547002792
1	15.899166107178	15.554542064667
	15.901528453827	**15.6038257598878**
	AVG TIME ORG	**AVG TIME MOD**

nprocs	Time Original(s)	Time Modified(s)
2	8.201547145844	8.139535903931
2	7.985216856003	8.199460983276
2	7.98254609108	7.980792045593
2	8.000113964081	7.985908985138
2	7.960376024246	7.989322900772
	8.0259600162508	**8.059004163742**
	AVG TIME ORG	**AVG TIME MOD**

nprocs	Time Original(s)	Time Modified(s)
4	4.207649946213	4.243479013443
4	4.170487880707	4.178691148758
4	4.197046995163	4.26014995575
4	4.18248295784	4.159827947617
4	4.140428066254	4.141149044037
	4.1796191692354	**4.196659421921**
	AVG TIME ORG	**AVG TIME MOD**

nprocs	Time Original(s)	Time Modified(s)
8	2.212038993835	2.217353820801
8	2.183537006378	2.183438062668
8	2.211024999619	2.183103084564
8	2.226603984833	2.181988954544
8	2.195514917374	2.219805955887
	2.2057439804078	**2.1971379756928**
	AVG TIME ORG	**AVG TIME MOD**

nprocs	Time Original(s)	Time Modified(s)
16	1.810024023056	1.937111139297
16	1.786022901535	2.182522058487
16	1.821040868759	1.864782094955
16	1.795344114304	1.112848043442
16	1.749959945679	1.862988948822
	1.7924783706666	**1.7920504570006**
	AVG TIME ORG	**AVG TIME MOD**

3.1.2 Average time, errors, numerical values, and speed up calculation for fixed N

ORIGINAL CODE

nprocs	Numerical	Error	Time (s)	SPEEDUP
1	3.141592640189	1.00824E-07	15.901528453827	1
2	3.141592640192	1.00821E-07	8.0259600162508	1.98126185797461
4	3.141592640184	1.00829E-07	4.1796191692354	3.80454003342509
8	3.141592640183	1.00829E-07	2.2057439804078	7.20914512068038
16	3.141592640186	1.00827E-07	1.7924783706666	8.87125262655941

MODIFIED CODE

nprocs	Numerical	Error	Time (s)	SPEEDUP
1	3.141592640189	1.00824E-07	15.6038257598878	1
2	3.141592640192	1.00821E-07	8.059004163742	1.93619775382304
4	3.141592640184	1.00829E-07	4.196659421921	3.71815393891202
8	3.141592640183	1.00829E-07	2.1971379756928	7.10188706058281
16	3.141592640186	1.00827E-07	1.7920504570006	8.70724688522683

3.1.3 Plots comparing original and modified code fixed N

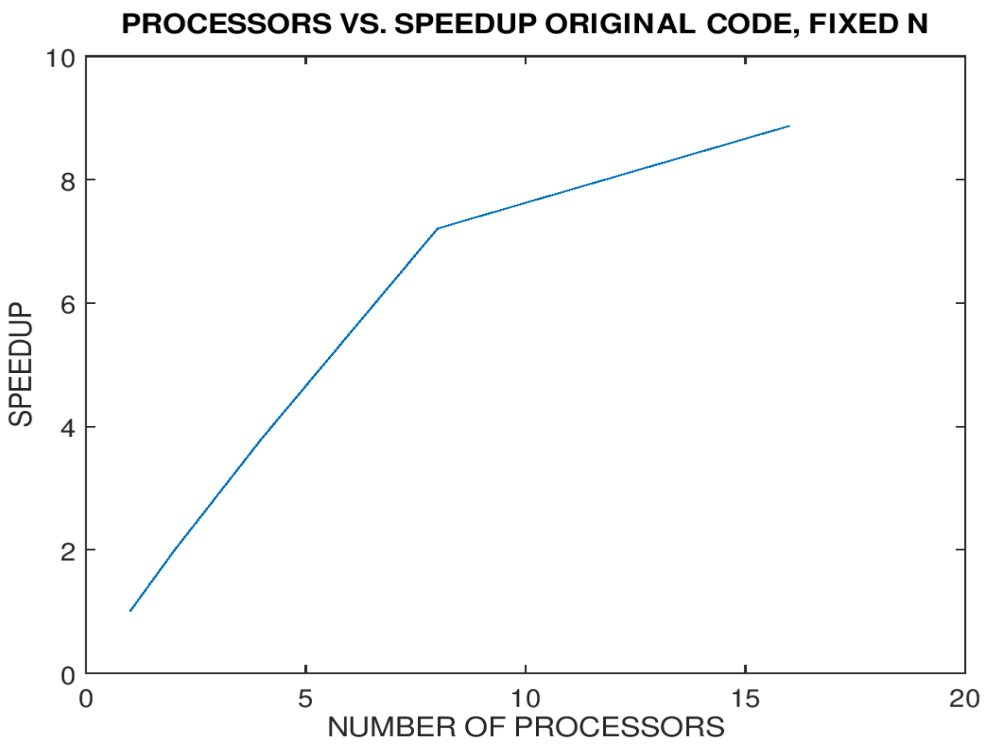

Figure 1: Original code speed up fixed N

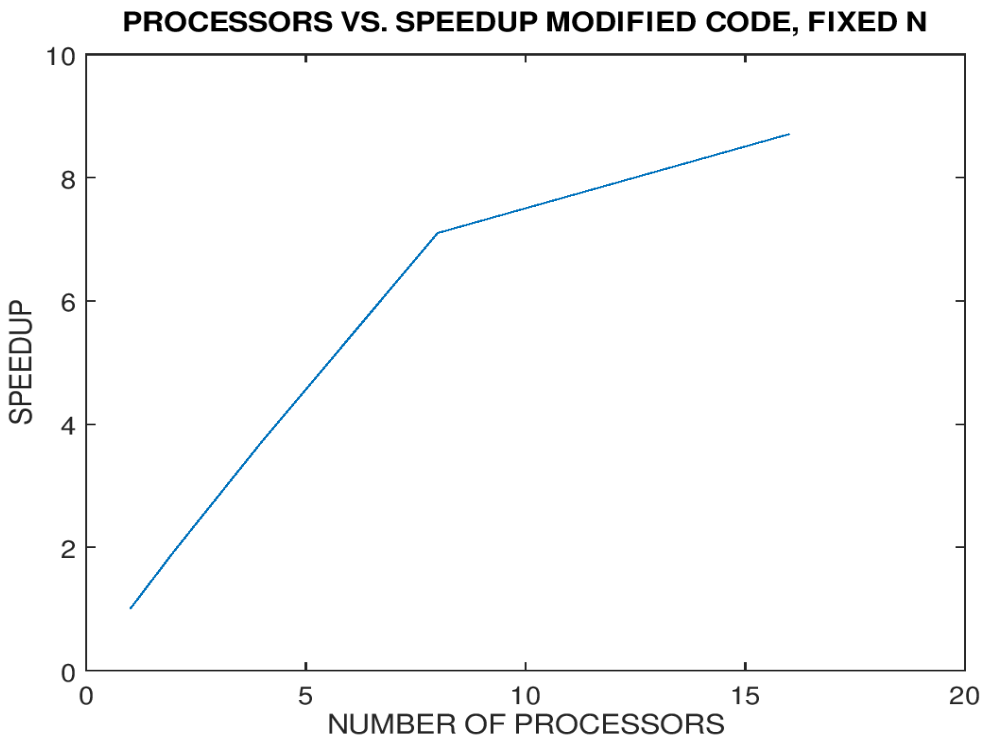

Figure 2: Modified code speed up fixed N

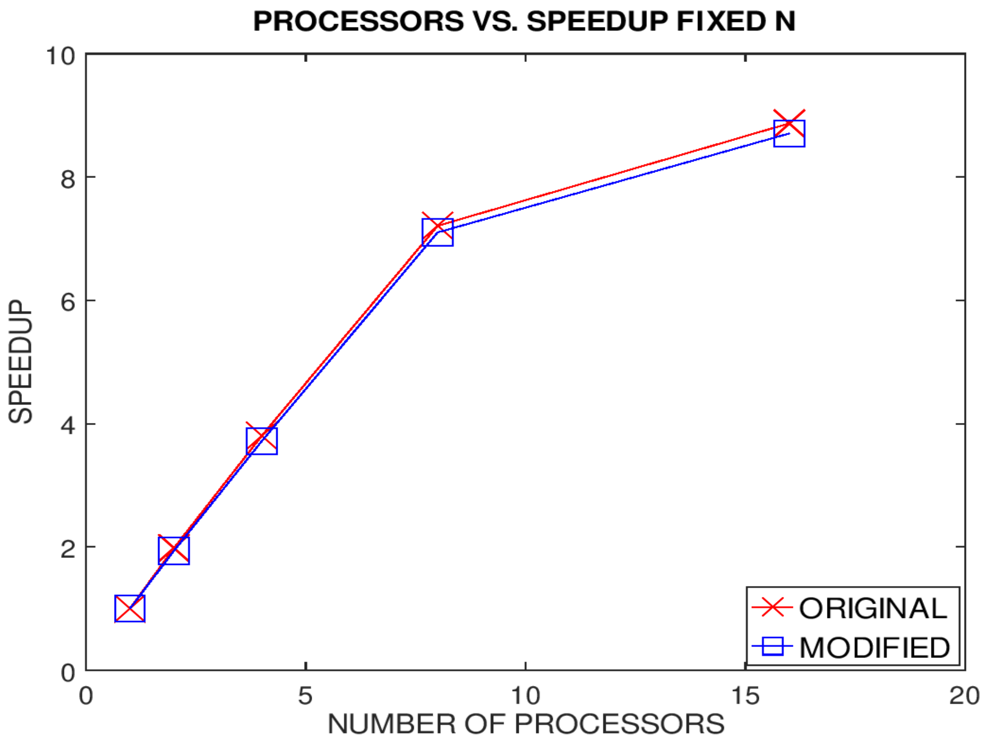

Figure 3: Original vs. Modified speed up fixed N

3.2 VARIED N

3.2.1 Time results for varied N for different number of processors

nprocs	Time Original(s)	Time Modified(s)	N PROBLEM
1	2.210348844528	2.180599212646	100000000
1	2.170042991638	2.174518108368	100000000
1	2.200505018234	2.16720199585	100000000
1	2.206801891327	2.175127029419	100000000
1	2.194341897964	2.16855096817	100000000
	2.1964081287382	**2.1731994628906**	
	AVG TIME ORG	**AVG TIME MOD**	

nprocs	Time Original(s)	Time Modified(s)	N PROBLEM
2	2.266933917999	2.24723482132	200000000
2	2.265223026276	2.265326023102	200000000
2	2.251295089722	2.230370998383	200000000
2	2.258610010147	2.221760034561	200000000
2	2.277946949005	2.243576049805	200000000
	2.2640017986298	**2.2416535854342**	
	AVG TIME ORG	**AVG TIME MOD**	

nprocs	Time Original(s)	Time Modified(s)	N PROBLEM
4	2.25540804863	2.213683128357	400000000
4	2.207736968994	2.241452932358	400000000
4	2.206567049026	2.22324681282	400000000
4	2.213709115982	2.21227812767	400000000
4	2.212677955627	2.226238012314	400000000
	2.2192198276518	**2.2233798027038**	
	AVG TIME ORG	**AVG TIME MOD**	

nprocs	Time Original(s)	Time Modified(s)	N PROBLEM
8	2.214080095291	2.227915048599	800000000
8	2.184514045715	2.192301034927	800000000
8	2.204570055008	2.190049171448	800000000
8	2.180907964706	2.199234008789	800000000
8	2.182548046112	2.196539878845	800000000
	2.1933240413664	**2.2012078285216**	
	AVG TIME ORG	**AVG TIME MOD**	

nprocs	Time Original(s)	Time Modified(s)	N PROBLEM
16	3.140465021133	3.430103063583	1600000000
16	3.142266988754	3.244676113129	1600000000
16	3.457314968109	3.254930973053	1600000000
16	3.231369972229	3.524397850037	1600000000
16	3.297169923782	2.878983974457	1600000000
	3.2537173748014	**3.2666183948518**	
	AVG TIME ORG	**AVG TIME MOD**	

3.2.2 Average time, errors, numerical values, and speed up calculation for varied N

ORIGINAL CODE

nprocs	Numerical	Error	N PROBLEM	Time (s)	SCALE
1	3.14159263198	1.09033E-07	100000000	2.196408129	1
2	3.141592636502	1.04511E-07	200000000	2.264001799	0.97014416246
4	3.141592638941	1.02072E-07	400000000	2.219219828	0.98972084756
8	3.141592640183	1.00829E-07	800000000	2.193324041	1.00140612482
16	3.141592640807	1.00205E-07	1600000000	3.253717375	0.67504576327

MODIFIED CODE

nprocs	Numerical	Error	N PROBLEM	Time (s)	SCALE
1	3.14159263198	1.09033E-07	100000000	2.173199463	1
2	3.141592636502	1.04511E-07	200000000	2.241653585	0.96946266676
4	3.141592638941	1.02072E-07	400000000	2.223379803	0.97743060374
8	3.141592640183	1.00829E-07	800000000	2.201207829	0.9872759104
16	3.141592640807	1.00205E-07	1600000000	3.266618395	0.66527497253

3.2.3 Plots comparing original and modified code varied N

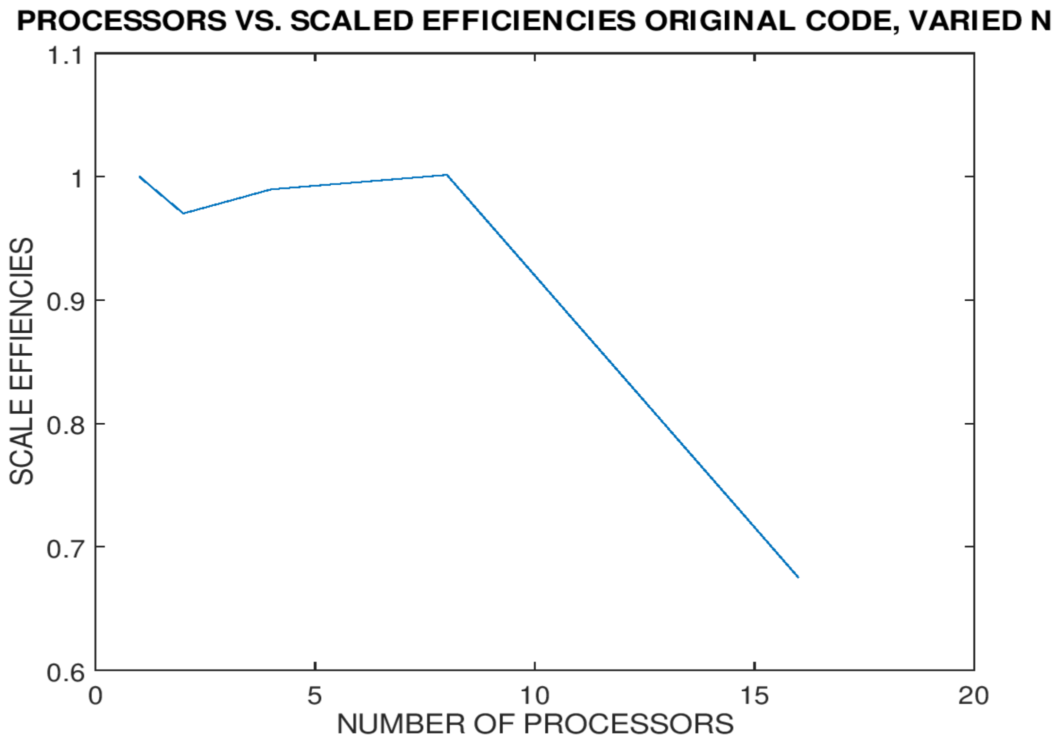

Figure 4: Original code speed up varied N

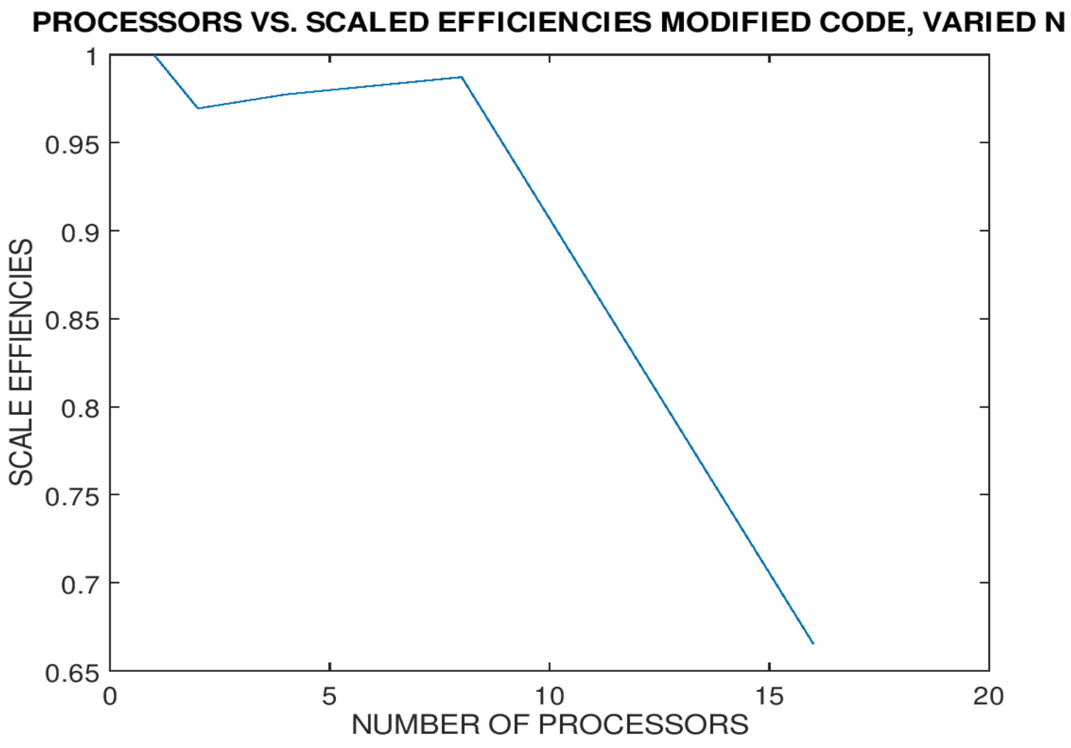

Figure 5: Modified code speed up varied N

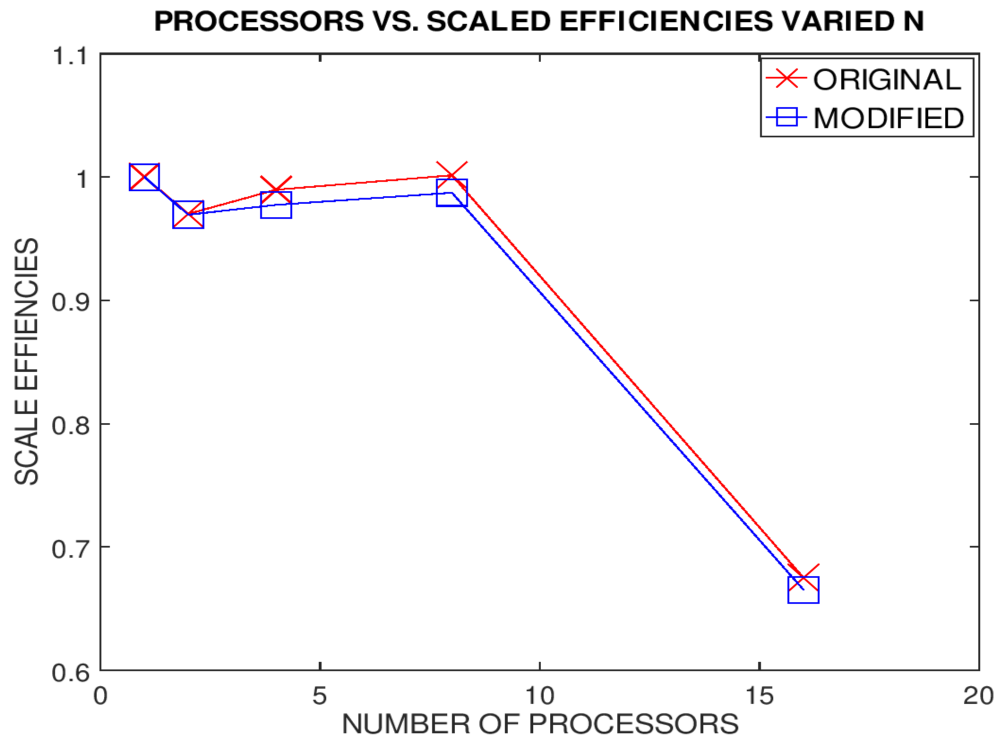

Figure 6: Original vs. Modified speed up varied N

4 Discussion

4.1 Speed Up Study

For the speed up study, theoretically, perfect speed up is achieved when Speed Up = Number of Processors. We can see from Figure 1 to 3 that perfect speed up is almost achieved when using 2,4 and 8 processors. For example, for 8 processors, we got 7.1 as a speed up factor, this is close to 8, making it almost a perfect speed up. The same thing can be said about 2 and 4 processors. Unfortunately, 16 processors do not show the same trend. We are getting around 8.7 as speed up factor for 16 processors. This is not very efficient because we can just get the same speed up factor as 8 instead, without spending more resources on using 16 processors.

4.2 Scaled Efficiency

In contrast to the speed up test, perfect scaling is achieved when Scaling = 1. This means that our work rate to solve a similar problem is the same. In other words, because the number of intervals per process is kept constant; we should expect the time it takes to solve the problem to be similar. This is true if we look at Figure 4,5 and 6. For 2, 4, and 8 processors, the scaled efficiencies flunctuate around 1. Now, this is clearly not the case for 16 processors, the scaled efficiency is only 0.67 for both original code and modified code. It looks as if our work rate to solve the problem gets diminished. One possible explanation for this outlier in both Speed Up and Scaled Efficiency has to do with hardware problems. The Blueshark cluster has 2 times Hexa-Core Intel Xeon X5650, this means 12 cores per node. Therefore, if we are requesting more than 12 processors, in this case 16, then the cluster might not request the correct number of processors. Thus, for the Speed Up test, we don't have enough power (processors) to solve. On the other hand, for the Scaled Efficiency test, our code requests a large problem size; however, because of the hardware, we do not have enough resources to solve such large problem.

4.3 Modified vs Original

In both test cases, the original is better. For example, using 8 processors, the original code has a closer value of speed up factor to being ”perfect”, 8, than the modified code. The same thing can be said about the scaled efficiency test, the original code has closer efficiency value to 1.0 than the modified one. One possible explanation would be that the Broadcast and Reduce functions are well written for numerical integrations, i.e sending problem size and then collect them all back. The send/receive functions can replace the original bcast/reduce; however, it is not very efficient. One way to think about this is that to we need a Do/For loop for sending problem size ”N” from process 0 to all of the other processors. It could be that the Do/For loop in Bcast/Reduce is written better and therfore is optimized. Although, the difference is not that great: original code is only slightly better than the modified one. If optimization is true for Bcast/Reduce, then we should see a more drastic difference between the original and modified code when we solve large problems where there are needs to send and receive large amount of data between processors.

5 Code

```fortran
program midpoint_speedup

        include 'mpif.h'

        integer :: ierr, rank, nprocs,n,i, isend, nsend, nreceive
        integer,dimension(MPI_STATUS_SIZE) :: status1
        double precision :: real_PI
        double precision :: mypi, pi, h, sum, x
        double precision startwtime, endwtime
        double precision :: mypi0, mypi_recv

        startwtime = 0.0
        real_PI = 3.14159265358979323846264 3

        !Start OPENMP
        call MPI_INIT(ierr)
        call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
        call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)


        n = 0
        ! SET PROBLEM SIZE AT PROCESS 0
        if (rank == 0) then
                ! Change this for varied N, or fixed N
                !n =  800000000
                n = nprocs*100000000
                startwtime =  MPI_WTIME()
        end if

    !************** SEND/RECEIVE PROBLEM SIZE FROM 0  ********** !
    ! SEND CALL
    if (rank == 0) then
        do i = 1,nprocs-1
            call MPI_SEND(n,1,MPI_INT,i,1,MPI_COMM_WORLD,ierr)
        end do
    ! RECEIVE CALL
    else
        call MPI_RECV(n,1,MPI_INT,0,1,MPI_COMM_WORLD,status1,ierr)
    end if
    !***************** MID POINT CODE ************************!
    ! COMPUTE PARTIAL SUM
    h = 1.0/(1.0*n)
    sum = 0.0
    do i = rank+1,n,nprocs
        x = h*((1.0)*i-0.5)
        sum = sum + 4.0/(1.0+x*x)
    end do
    mypi = h*sum
    if (rank == 0) then
        pi = mypi
    end if

    !*************** SEND MY PI TO ZERO TO SUM UP  ****************!
    if (rank == 0) then
        do i = 1,nprocs-1
            ! receive my pi
            call MPI_RECV(mypi_recv,1,MPI_DOUBLE_PRECISION,i,2,&
                                    MPI_COMM_WORLD,status1,ierr)
            pi = pi + mypi_recv
        end do
    else
        ! send mypi
        call MPI_SEND(mypi,1,MPI_DOUBLE_PRECISION,0,2,&
                                    MPI_COMM_WORLD,ierr)
    end if


    !**********  WRITE RESULT *******************************!
    if (rank == 0) then
        endwtime =  MPI_WTIME()
        write (6,200) pi, abs(pi-real_PI), nprocs
        200 format('␣','pi␣is␣approximately␣',f16.12,',Error␣is␣',&
                        f16.12,'␣nprocs␣=␣', I2)
        write (6,300) endwtime-startwtime
    300 format('␣','wall␣click␣time␣=␣', f16.12)
        write(6,400) n
    400 format('␣','N␣SIZE␣=␣', I10)

    end if

    ! Close OPENMP
    call MPI_FINALIZE(ierr)
end program midpoint_speedup
```