CS546 Parallel and distributed Processing

Programming Assignment - 4

<u>Performance Evaluation</u>

In this assignment, we have to improve the performance of two applications, Conjugate Gradient and Multi-Grid .we know that Conjugate Gradient and Multi-Grid are two kernel applications in NAS Parallel Benchmarks (NPB). NPB is a small set of programs designed to help evaluate the performance of parallel supercomputers.

We have to Downloaded the benchmark, that can be found http://www.nas.nasa.gov/publications/npb.html. And I have to improve the performance of CG and MG by making changes in OpenMP pragmas, add/delete OpenMP constructs and, change environment variables and other compiler tags in the code.

1. Conjugate Gradient (CG): Initially we have run the OpenMP (C) version of CG for 1,2,4,8 processors Next, we need to improve the Performance of conjugate gradient by making necessary changes. I have tried various approaches to improve performance, but it doesn't. . And it gives on later stages. Initially, I added environment variables and deleted few pragmas for norm_temp1,norm_temp2 , removed parallel version for reduction of norm_temp1 and norm_temp2,removed parallel version of conjugate gradient routine. In conj_grad(), added pragamas for rho,sum,q,z,r,p intilizations ,reduction of rho,reduction of d and partition submatrix vector, and in makea(), added pragmas for vecset,aprnvc. By making these changes, it doesn't give better performance(1.2% increases).

Then, I added three implementations of the sparse matrix-vector multiply and default matrix-vector multiply is not loop unrolled. Added pragmas for conju_grad(),makes(),and implementations are unrolled to a depth of 2 and unrolled to a depth of 8.we can observe that, fastest performance and reporting timing results, any of these three may be used without penalty.

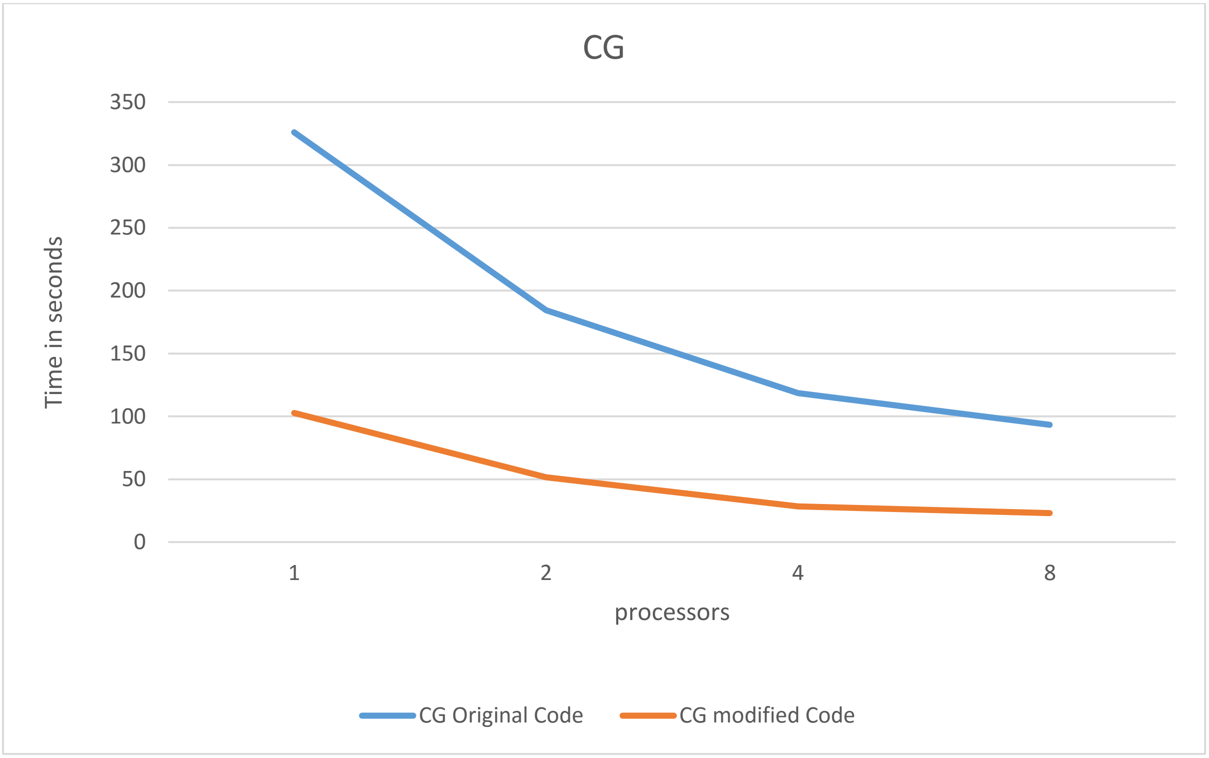
I've evaluated the OpemMP (C) version of parallel code up to 8 process. All the results listed in Results folder.

```
CG Benchmark Completed.
Class           =                        C
Size            =                   150000
Iterations      =                       75
Time in seconds =                    28.30
Total threads   =                        4
Avail threads   =                        4
Mop/s total     =                  4438.19
Mop/s/thread    =                  1109.55
Operation type  =           floating point
Verification    =               SUCCESSFUL
Version         =                    3.3.1
Compile date    =              10 Oct 2016
```

The below tables and plots are drawn based on Total time(seconds) time and number of processes.

Number of processors	1	2	4	8
CG Original Code	325.94s	184.47s	118.4s	93.2s
CG modified Code	102.74s	51.64s	28.3s	22.93s

Graph:



Observation: we can observe that, the modified version of CG gives better Performance compared to Original version. In modified version CG, the performance increases as the number of processors increases because changes made in sparse matrix-vector multiplication reflects.

2. Multi-Grid (MG): Initially we have run the OpenMP (C) version of CG for 1,2,4,8 processors Next, we need to improve the Performance of Multi-Grid by making necessary changes. I have tried various approaches to improve performance, but it doesn't. And it gives on later stages. Initially, I added global parameters T_BENCH AND T_INIT, parallelize norm2u3(),zero3().and added pragmas for setup() functions. In psinv(),deleted pragma for private(i1,i2,i3,r1,r2) and added for rep_nrm(). In resid(),added one pragma for r = v - Au and this vectorizes. In rprj3(),added pragma because it projects onto the next coarser grid, using a trilinear Finite Element projection: s = r' = P r. In interp(),Added #pragma for calculation of arrays z1,z2,z3.In norm2u3(),removed no wait in homogeneous boundaries. Here, performance doesn't change,onlt it changes by 0.6%.

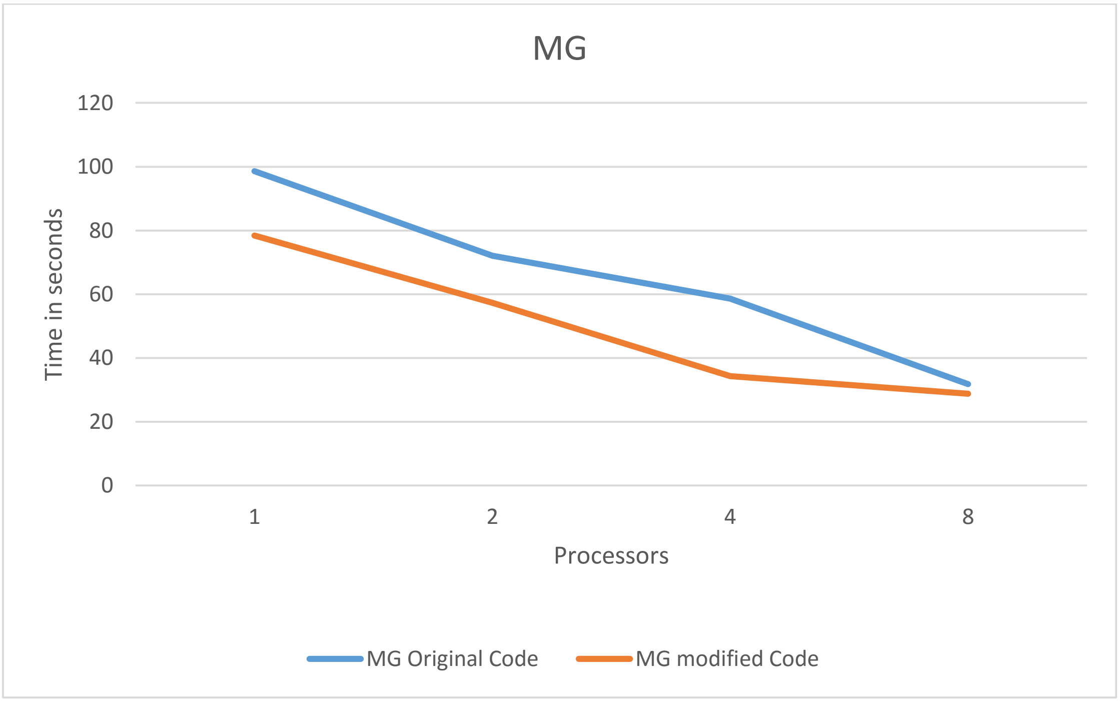
Then, I added few pragmas in comm3,it organizes the communication on all borders, parallelizes all the vectors and including above changes. It gives better performance then pervious stages compared to original version.

I've evaluated the OpemMP (C) version of parallel code up to 8 process. All the results listed in Results folder.

```
MG Benchmark Completed.
Class           =                        C
Size            =            512x 512x 512
Iterations      =                       20
Time in seconds =                    34.25
Total threads   =                        4
Avail threads   =                        4
Mop/s total     =                  4248.49
Mop/s/thread    =                  1062.12
Operation type  =           floating point
Verification    =               SUCCESSFUL
Version         =                    3.3.1
Compile date    =              16 Oct 2016
```

Number of processors	1	2	4	8
MG Original Code	98.64s	72.12s	58.6s	31.76s
MG modified Code	78.4s	57.3s	34.25s	28.74s

Graph:



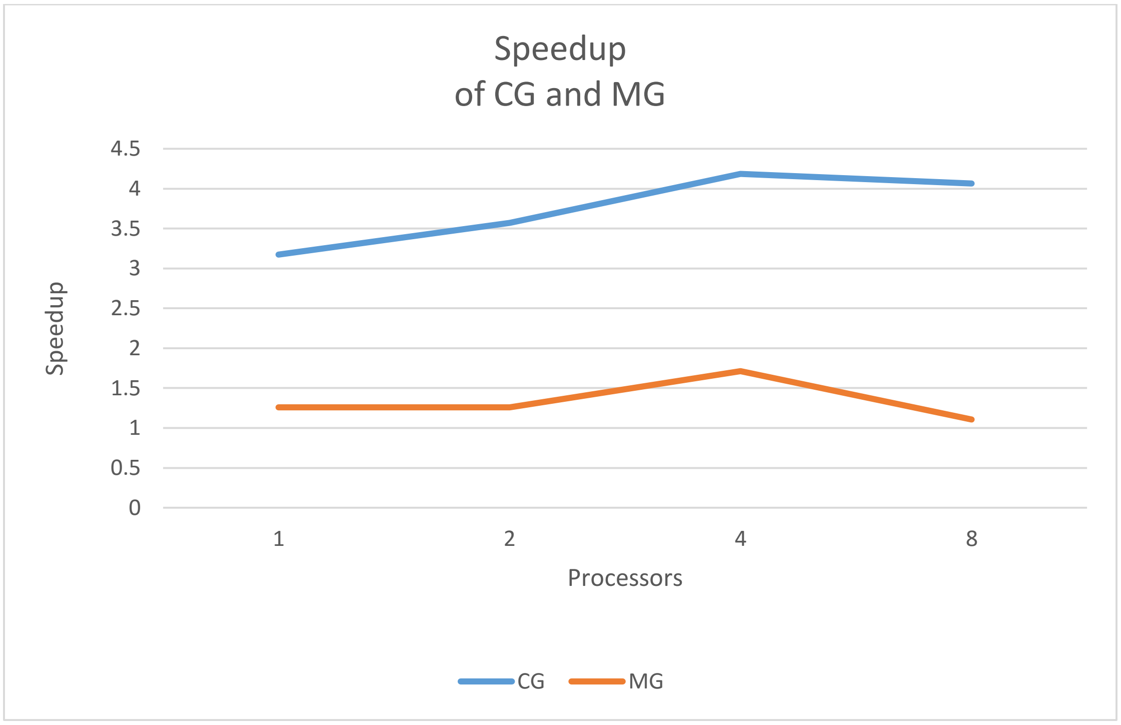
Observation: We can observe that, the modified version of MG gives better performance than the original version., As the number of processors increases , performance increases because changes made in comm3,basically it organizes the communication on all borders, and parallelizes all the vectors

Speedup: Now, we need to calculate speedup of CG AND MG by varying the number of Processors.

The below table shows the speedup of both CG and MG for processors 1,2,4 and 8.

		Speedup		
Number of processors	1	2	4	8
CG	3.172474207	3.572230829	4.183745583	4.064544265
MG	1.258163265	1.258638743	1.710948905	1.105080028

Graph :



Observation: The modified version of CG gives better speedup compared to MG. In CG, the speedup increases as the number of processors increases up to 4 processors and slightly decreases in case of 8 processors because changes made in sparse matrix-vector multiplication. And decrease in 8 processors results due to more computation in multiplication of sparse matrix.

In MG, the speedup increases as the number of processors increases up to 4 processors and decreases in case of 8 processors because changes made in comm3,basically it organizes the communication on all borders, parallelizes all the vectors. And decrease in 8 processors results due to vectorization of all three dimensions.