Homework 1
Collective Communication and Coalesced Access

Jatin Gupta- 13572

1 Collective Communication

1.1 Summary of Code

Message size is generated as the array of integer,which means for 1k message size each array must have $\frac{2^{10}}{2^2} = 2^8$ number of elements.
Random number is generated on all the processors in send_buffer and MPI_Reduce function is used to sum all the array elements . also MPI_Scatter is used to scatter the array and store it in the recv_buffer.

```
55  // reduce using mpi call
56  MPI_Reduce((void *)send_buffer,final_sum,TOTAL_ARRAY_LENGTH,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
57
58   //TOdo changed RECV_ARRAY_LENGTH
59  MPI_Scatter(final_sum,RECV_ARRAY_LENGTH,MPI_INT,recv_buffer,RECV_ARRAY_LENGTH,MPI_INT,0,MPI_COMM_WORLD);
60
```

1.2 MPI_Reduce and MPI_Scatter

MPI_Reduce is used to perform the reduction operation which are associative and commutative,here on 'sum operator' to reduce all on 0 rank. while scatter will distribute piecewise the results to all the nodes . Reduce in MPICH is implemented as binomial tree,which takes logp steps . Therefore $T_{tree} = \lceil logp \rceil(\alpha + n\beta)$ while MPI_Scatter is implemented in MPICH as ring algorithm.

1.3 MPI_Reducescatter using Recursive halving

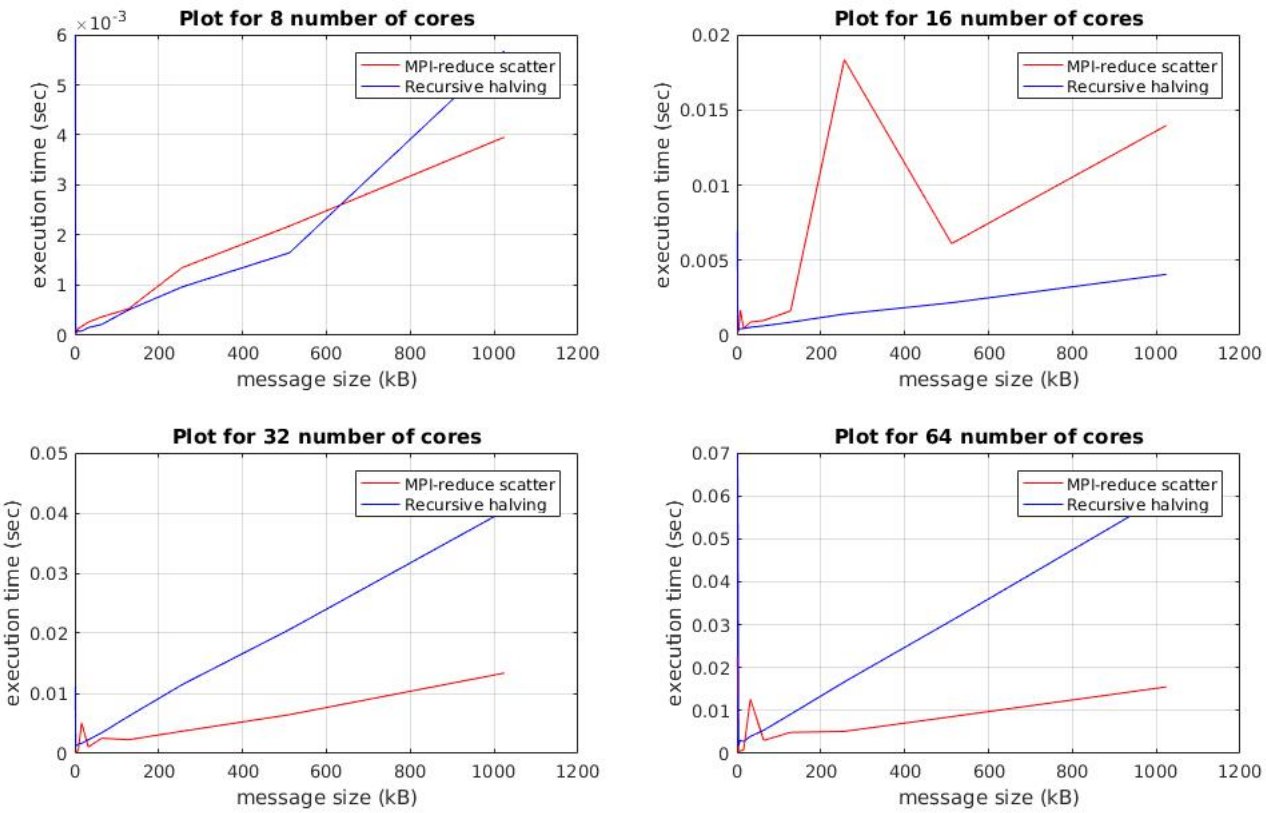
ReduceScatter is implemented through recursive halving. Here each processor will send message to the other processor which are $\frac{p}{2}$ distance away from the i node. Here all processors assumed to be divided into groups and all the processors are in one group communicates with each other. At each stage the group size will decrease by half. These communication will proceed for log(n) where n is number of processor.

```
79      while(q-1 != 0){
80          q = q/2;          // communication group size
81          p = myrank/q;     // pseudo communication group number
82          g = p*q;      // communicating group leader (1st node in every pseudo communication group)
83
84
85          sendr = myrank;
86          recvr = ((myrank-g)+(q/2))%q + g;
```

1.4 Mathematical Comparison

The data is communicated for logp steps, So for power of 2 processors the time taken by recursive halving is
$T_{rec\_half} = O(lgp\alpha + \frac{p-1}{p}n\beta)$
While for reduction using binomial tree and scatter using linear/ ring algorithm will take total of
$T_{Reduce\_scatter} = O((lgp + p - 1)\alpha + (lgp + \frac{p-1}{p})n\beta))$

1.5 Plots



1.6 Analysis of plots

Plot implies that Recursive halving is good algorithm for less number of processors. while for larger number of processors MPICH algorithm execution time is lesser.

2 Advantages of Coalesced Access

2.1 Summary of Code

Two files are created for Array of structure as (hw1-b-AOS.cu) and Structure of Array implementation as (hw1-b-SOA.cu).

2.1.1 Array of Structure

Structure is named as atom and 3 variable pos,vel and force are created of double3 type. double3 implies that each variable has x,y,z components.
kernel simply gets the value of the array located at the its thread id and updates it to twice the value and outputs in new location d_out.

```
21 // Array of structure a structure occupies 72 B
22 typedef struct {
23     double3 pos;
24     double3 vel;
25     double3 force;
26 }atom;
```

Figure 1: Array of structure

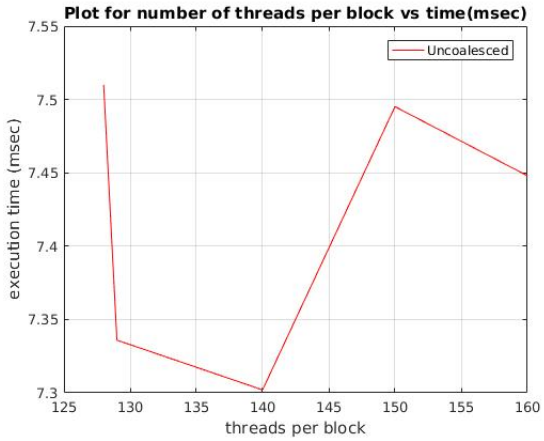
```
28 ////////////////////////////////////////////////////
29 //      KERNEL
30 ////////////////////////////////////////////////////
31 __global__ void updateAtomKernel(const atom *d_in,atom *d_out ,const int N){
32
33     //int t_idx = threadIdx.x;  // thread index
34     int idx = threadIdx.x + blockIdx.x*blockDim.x;
35
36     //Colesced memory access.
37
38     d_out[idx].pos.x =  2*d_in[idx].pos.x ;
39     d_out[idx].pos.y =  2*d_in[idx].pos.y ;
40     d_out[idx].pos.z =  2*d_in[idx].pos.z ;
41
42     d_out[idx].vel.x =  2*d_in[idx].vel.x;
43     d_out[idx].vel.y =  2*d_in[idx].vel.y;
44     d_out[idx].vel.z =  2*d_in[idx].vel.z;
45
46     d_out[idx].force.x = 2*d_in[idx].force.x;
47     d_out[idx].force.y = 2*d_in[idx].force.y;
48     d_out[idx].force.z = 2*d_in[idx].force.z;
49 }
50
```

Figure 2: Image showing kernel of Array of structure

Output:

# threads Per block	T (msec)
—128—	7.510010
129	7.335938
140	7.302002
150	7.495117
—160—	7.447998

Table 1: UnCoalesced Access Using Different number of threads per block



Analysis

Time taken for execution of thread block size which is multiple of warp size is less than non warp size , but thread block which are very close to warp size takes lesser time. This can be explained as, the remaining threads (threads - multiple of warp size) will pre call the global memory for further thread and overall will reduce time. But there will be rise in time once this advantage to warp threads is preceded.
Also, An SM can execute one block at a time for thread per block greater than (SM size / 2) so, in these cases more threads implies more usage of SM at once. which justify decrease of time from 128 threads to 160 threads per block.

2.1.2 Structure of Array

Structure is named as atom and 3 variable have their x,y and z components. Each array (pos,vel and force) with their x,y and z is defined as posx,posy,posz and similarly for vel and force.
Each thread access the value from d_in variable from the location of thread id. and outputs the updated value in d_out.

```
14  struct atom{
15       double posx[numThreads];
16       double posy[numThreads],posz[numThreads];
17       double velx[numThreads],vely[numThreads],velz[numThreads];
18       double forcex[numThreads],forcey[numThreads],forcez[numThreads];
19  };
```

Figure 3: Structure for Structure of Array

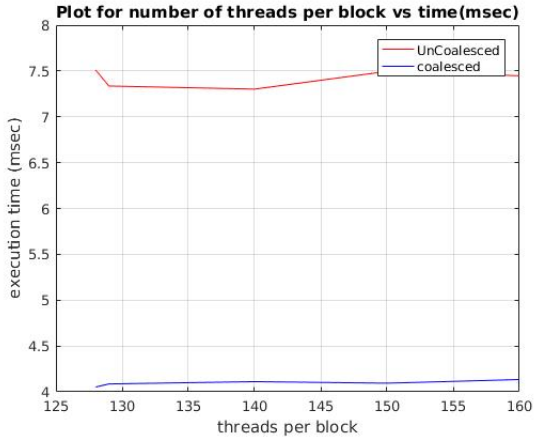
```
24  __global__ void updateAtomKernel(const atom *d_in,atom *d_out){
25
26     // thread id
27     int idx = threadIdx.x + blockIdx.x*blockDim.x;
28
29     //coalesced memory acces
30     d_out->posx[idx] = 2*d_in->posx[idx];
31     d_out->posy[idx] = 2*d_in->posy[idx];
32     d_out->posz[idx] = 2*d_in->posz[idx];
33
34     d_out->velx[idx] = 2*d_in->velx[idx];
35     d_out->vely[idx] = 2*d_in->vely[idx];
36     d_out->velz[idx] = 2*d_in->velz[idx];
37
38     d_out->forcex[idx] = 2*d_in->forcex[idx];
39     d_out->forcey[idx] = 2*d_in->forcey[idx];
40     d_out->forcez[idx] = 2*d_in->forcez[idx];
41
42  }
```

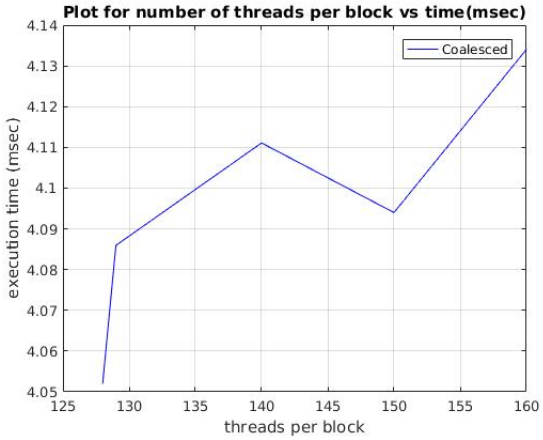
Figure 4: Kernel for Structure of Array

Output:

# threads Per block	T (msec)
—128—-	4.052002
129	4.085938
140	4.111084
150	4.093994
—160—-	4.134033

Table 2: Coalesced Access using different no. threads per block





Analysis

Coalesced memory access time is approximately half that of uncoalseced memory access for all the no. threads per block. Also when number of thread blocks are multiple of warp size then time taken for execution is less.

Results Collective Communication:

```
////////////////// 8 Num cores //////////////////
S.no    Size(KB)    Time(R.S.)(sec)    Time(R.H.)(sec)    Speedup (R.S./R.H.)    Validity
------  ----------  -----------------  -----------------  ------------------- ----------
  1     1                  0.000086        0.006622        0.012997        1
  2     2                  0.000050        0.000172        0.291262        1
  3     4                  0.000040        0.000047        0.852792        1
  4     8                  0.000134        0.000089        1.502674        1
  5     16                 0.000177        0.000082        2.156977        1
  6     32                 0.000263        0.000155        1.694316        1
  7     64                 0.000368        0.000217        1.696703        1
  8     128                0.000529        0.000508        1.041295        1
  9     256                0.001352        0.000967        1.398176        1
  10    512                0.002182        0.001648        1.324074        1
  11    1024               0.003956        0.005678        0.696746        1

////////////////// 16 Num cores //////////////////
S.no    Size(KB)    Time(R.S.)(sec)    Time(R.H.)(sec)    Speedup (R.S./R.H.)    Validity
------  ----------  -----------------  -----------------  ------------------- ----------
  1     1                  0.000193        0.006924        0.027891        1
  2     2                  0.000193        0.000531        0.363269        1
  3     4                  0.000256        0.000364        0.703801        1
  4     8                  0.001674        0.000443        3.778794        1
  5     16                 0.000463        0.000453        1.022105        1
  6     32                 0.000883        0.000539        1.637771        1
  7     64                 0.000991        0.000635        1.561021        1
  8     128                0.001629        0.000873        1.865920        1
  9     256                0.018347        0.001418        12.939802       1
  10    512                0.006107        0.002172        2.811745        1
  11    1024               0.013965        0.004057        3.442087        1

////////////////// 32 Num cores //////////////////
S.no    Size(KB)    Time(R.S.)(sec)    Time(R.H.)(sec)    Speedup (R.S./R.H.)    Validity
------  ----------  -----------------  -----------------  ------------------- ----------
  1     1                  0.000225        0.011219        0.020061        1
  2     2                  0.000264        0.001237        0.213336        1
  3     4                  0.000356        0.001276        0.278961        1
  4     8                  0.000437        0.001592        0.274524        1
  5     16                 0.005093        0.001682        3.027782        1
  6     32                 0.001058        0.002278        0.464469        1
  7     64                 0.002542        0.003443        0.738315        1
  8     128                0.002287        0.006176        0.370290        1
  9     256                0.003700        0.011456        0.322976        1
  10    512                0.006457        0.020622        0.313116        1
  11    1024               0.013379        0.040322        0.331806        1
```

```
////////////////// 64 Num cores //////////////////
S.no    Size(KB)    Time(R.S.)(sec)    Time(R.H.)(sec)    Speedup (R.S./R.H.)    Validity
------  ----------  -----------------  -----------------  ------------------- ----------
  1     1                  0.022889        0.084732        0.270133        1
  2     2                  0.016664        0.058545        0.284637        1
  3     4                  0.000475        0.001864        0.254925        1
  4     8                  0.000571        0.003108        0.183722        1
  5     16                 0.000847        0.002740        0.309058        1
  6     32                 0.012632        0.003974        3.178726        1
  7     64                 0.003040        0.005415        0.561377        1
  8     128                0.004922        0.009130        0.539092        1
  9     256                0.005137        0.016662        0.308307        1
  10    512                0.008569        0.030955        0.276823        1
  11    1024               0.015497        0.060210        0.257382        1
```

Coalesced Memory Access

```
Time taken in ThreadsPerBlock: 128 is 7.510010 msec
./hw1-b-AOS 129
Time taken in ThreadsPerBlock: 129 is 7.335938 msec
./hw1-b-AOS 140
Time taken in ThreadsPerBlock: 140 is 7.302002 msec
./hw1-b-AOS 150
Time taken in ThreadsPerBlock: 150 is 7.495117 msec
./hw1-b-AOS 160
Time taken in ThreadsPerBlock: 160 is 7.447998 msec
./hw1-b-SOA 128
Time taken in ThreadsPerBlock: 128 is 4.052002 msec
./hw1-b-SOA 129
Time taken in ThreadsPerBlock: 129 is 4.085938 msec
./hw1-b-SOA 140
Time taken in ThreadsPerBlock: 140 is 4.111084 msec
./hw1-b-SOA 150
Time taken in ThreadsPerBlock: 150 is 4.093994 msec
./hw1-b-SOA 160
Time taken in ThreadsPerBlock: 160 is 4.134033 msec
```