Harun Şaşmaz
59900
COMP 429 – Assignment 3

REPORT

In this assignment, we implemented an iterative sparse matrix – dense vector multiplication with 1D row partitioning. There are three parts of the assignment and my submission includes also the bonus part. All of the three parts are working properly, and comments are added step by step in source code. This assignment uses **mpich v3.2.1** and communication is collective through all three parts.

Compile and Test:

I could not figure out how to arrange Makefile to compile all parts at once, so for testing, you need to move the regarding part to src where matrix.cpp and mmio.cpp are. Then, you can call make.

Important Variables:

```
int* rowptr              // part of the matrix.csrRowptr for a process
int* colptr              // part of the matrix.csrColIdx for a process
double* valptr           // part of the matrix.csrVal for a process
int elm_count            // element count of a process
int elm_displs           // starting index of elements for a process
int row_count            // row count of a process.
int eCounts[]            // array of elm_counts to be scattered
int eDispls[]            // array of elm_displs to be scattered
int rowCounts[]          // array row_counts after partitioning
int rowDispls[]          // array of starting indexes of rows for all processes
MPI_Datatype info_type   // new introduced datatype to pack sending data in one scatter.
info_t* infos            // list of info needed for processes
info_t  my_info          // process info for a process
```

Implementation: **See source code for step by step explanations*

Part I:

This part starts with reading of input matrix and conversion of that matrix into CSR format by master process. When reading is done, the master process divides the number of rows to the number of processes to calculate how many rows each process gets. Since it is not guaranteed that number of processes divides number of rows, the division is rounded, and the last process gets the leftover rows. After this calculation done, master process fills rowCounts and rowDispls arrays. Then, it calculates number of elements and its displacements for each process and fills eCounts and eDispls arrays. Also, the master process fills infos array with needed information. Once these calculations are done, master process scatters the infos array to all processes one by one. Each process gets the information about number of rows, time steps, elm_count and elm_displs. After that master broadcasts rowCounts and rowDispls because this is needed for Allgatherv operation. Then, rhs vector is broadcasted and rowptr, colptr and valptr are scattered by master process. Notice that rowptr array has an additional element, this is for adding elm_count to the end of rowptr so that process can know where to stop. After these scatter operations are done, the multiplication kernel starts. However, it first needs two arrays for local multiplications and gathering the total vector from other processes. Once a process completes its multiplication on related indexes, all processes share their results with Allgatherv and then each process copies this gathered result into rhs and continues to next time step. After multiplication, the elapsed time of each process is gathered by master and average time is printed out.

Part II:

Everything is the same with part I except the multiplication kernel. The hybrid implementation uses two omp parallel for pragmas so make multiplication parallel with OpenMP. This first pragma distributes rows that is owned by a process into OpenMP threads and each thread updates different indexes of local result. The second pragma distributes the copying operation of gathered result and each thread updates different indexes of rhs vector.

Part III:

Everything is the same with part I except row partitioning. This part makes use a load balanced row partitioning function. This function first puts the count of non-zero elements in each row in an array. Then it applies a binary search which starts with range 0 to total non-zero element count. The aim of this is to find such a lower bound that is appropriate for row partitioning. Once a lower bound is found, rowCounts and rowDispls arrays are filled in an iteration that checks each process gets balanced amount of non-zero elements.

Bottlenecks of This Implementation

The main bottleneck of this assignment is high amount of collective communication. All the data is sent from master to all other processes. My implementation reduces the number of communications by using an additional datatype which packs 4 information such as number of rows, time steps, element counts and element displacements into one data and scatter at once. Other than this, at each iteration, program has to use Allgatherv to gather multiplication result in all processes, this leads to a communication overhead at each iteration.

Hardware: (Node: me02)

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	24
On-line CPU(s) list:	0-23
Thread(s) per core:	1
Core(s) per socket:	12
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	62
Model name:	Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
Stepping:	4
CPU MHz:	1411.500
CPU max MHz:	3200.0000
CPU min MHz:	1200.0000
BogoMIPS:	4800.03
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	30720K

Results: **20 iterations for all parts and input files*

Cube_coup_dt6
Serial: 6.04 seconds

process\exe. time	Part 1	Part 2	Part 3
1	5.34	7.55	5.13
2	2.83	3.61	2.74
4	1.63	2.01	1.56
8	1.04	1.19	1.01
16	0.82	0.85	0.80

process \ speedup	Part 1	Part 2	Part 3
1	1.13	0.80	1.17
2	2.13	1.67	2.20
4	3.70	3.00	3.87
8	5.80	5.07	5.98
16	7.36	7.10	7.55

Flan_1565
Serial: 5.63 seconds

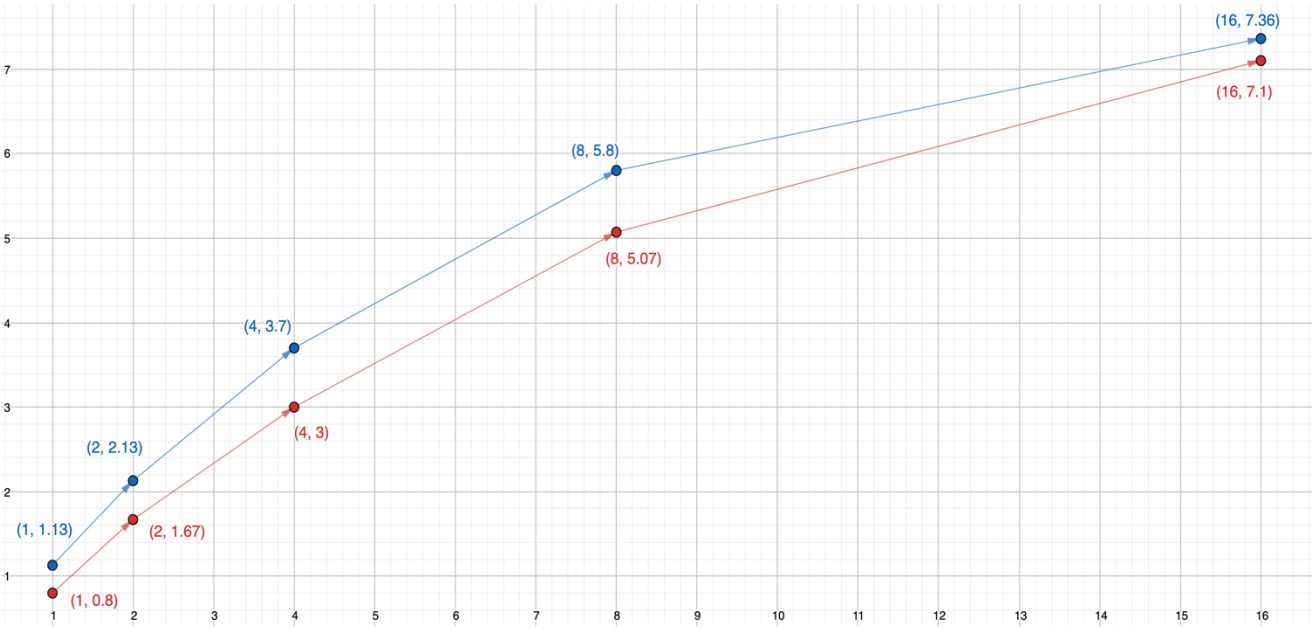
process\exe. time	Part 1	Part 2	Part 3
1	4.93	6.84	4.89
2	2.60	3.18	2.53
4	1.40	1.71	1.43
8	0.85	0.99	0.90
16	0.67	0.70	0.70

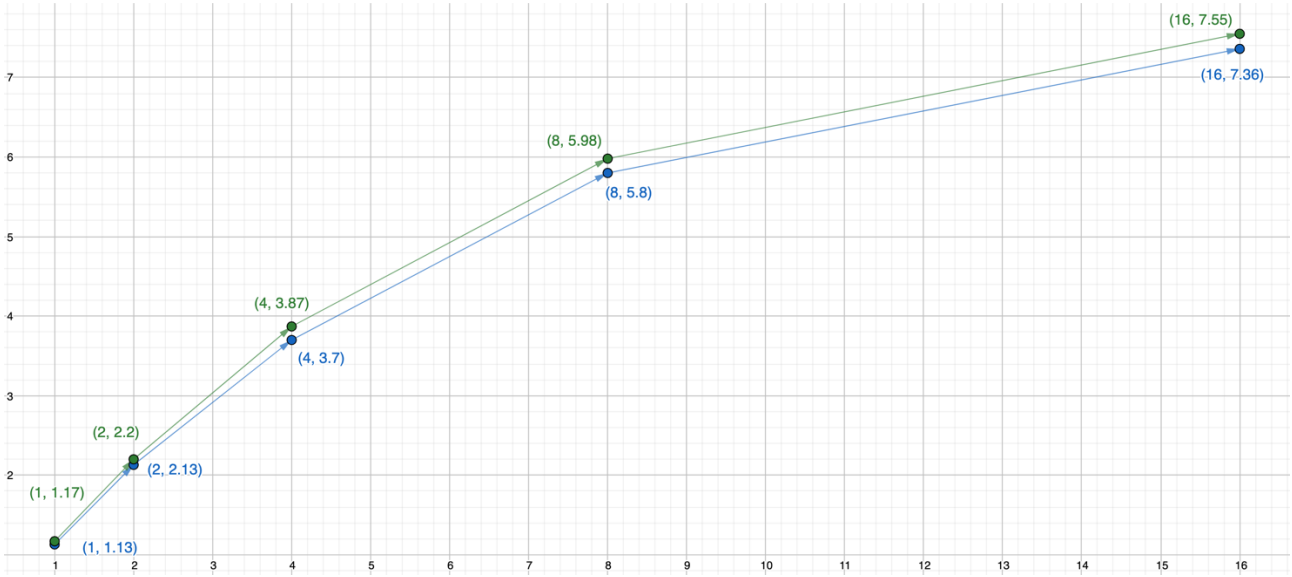
process \ speedup	Part 1	Part 2	Part 3
1	1.14	0.82	1.15
2	2.16	1.77	2.22
4	4.02	3.29	3.93
8	6.62	5.68	6.25
16	8.40	8.04	8.04

Graphical Comparison: ** x-axis is the number of processes; y-axis is speed-up*

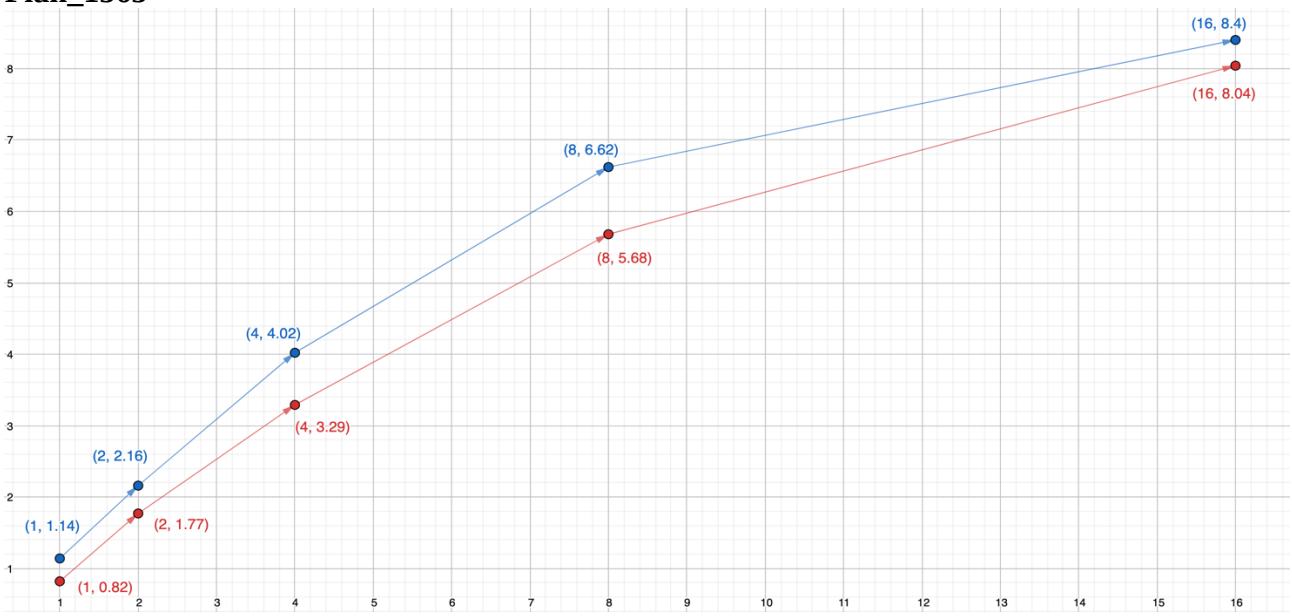
Blue represents part 1, red represents part 2, and **green represents **part 3**.*

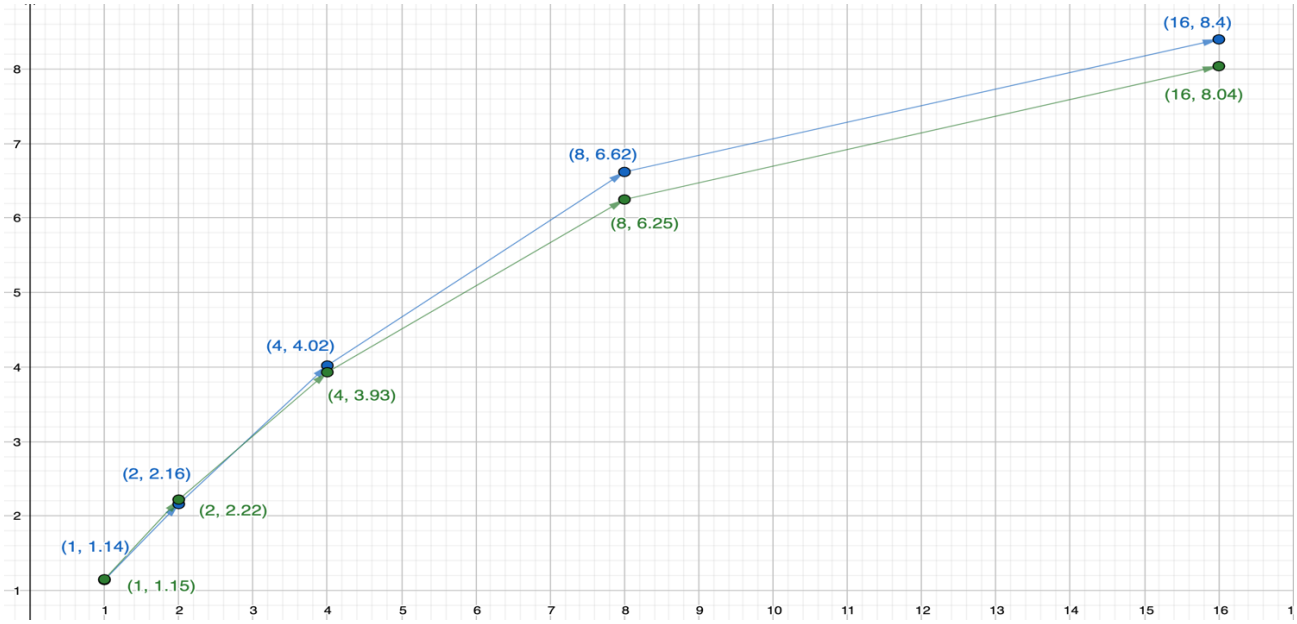
Coup_Cube_dt6




(16, 7.55)
(16, 7.36)
(8, 5.98)
(8, 5.8)
(4, 3.87)
(4, 3.7)
(2, 2.2)
(2, 2.13)
(1, 1.17)
(1, 1.13)

Flan_1565


(16, 8.4)
(16, 8.04)
(8, 6.62)
(8, 5.68)
(4, 4.02)
(4, 3.29)
(2, 2.16)
(2, 1.77)
(1, 1.14)
(1, 0.82)


(16, 8.4)
(16, 8.04)
(8, 6.62)
(8, 6.25)
(4, 4.02)
(4, 3.93)
(2, 2.16)
(2, 2.22)
(1, 1.14)
(1, 1.15)

Discussion:

When we first observe the results, it seems that the first part with 4 MPI processes has four times speedup. This may not be expected due to communication overhead. However, in the multiplication kernel, instead of reading from rowptr and updating local_res at each iteration, I declared temp variables that keeps the needed information about rowptr, and multiplication result is accumulated on a temp variable. This reduces execution time and hence the speedup is higher. Other than that, the main reason of not getting higher speedup is collective communication overhead. The higher number of processes, the higher overhead. Because at each iteration, each process sends its local result to the all other processes. In consideration of the amount of sharing data per process, communication volume and cost effect speedup.

If we compare part 1 and 3 for Cube Coup matrix, then we see that the resulting speedup is not that distinguishable because the time measurement is not precise. Elapsed times for different processes differ and we only consider the elapsed time of master process. Also, the distribution of non-zero elements may affect the results, because if the workload of the master process does not change much, then the result also does not change that much. The number of rows that master gets does not differ from part 1 to part 3, so this slight speedup is expected. However, for Flan_1565, it seems that part 3 is slower. The reason of this as mentioned above. Distribution of non-zero elements increases the workload of master process and so it takes longer time. To be more precise, we need to collect all elapsed times from all processes and analyze the deviation between them to understand the effect of load balancing precisely. In addition, we can measure the average time taken at one time step of all processes, analyze them and observe the deviation. This measurement will lead us to observe deviation of each process.

If we compare part 1 and 2, the expectation is that the hybrid implementation is slower than pure MPI. This problem is not suitable for shared memory programming because threads access to rhs vector without any pattern, and also each thread uses different rows of the matrix, so cache utilization is really low under these circumstances. This is why part 2 is slower than part 1 in all runs. Also, of course the 1 OpenMP thread version is the fastest because it is close to pure MPI except the cost of opening a parallel region. In my implementation, I avoid using parallel for loop in the inner loop where multiplication is accumulated on local result, because each process gets around 400 thousand of rows and it means opening 400 thousand parallel regions at each time step. Also, the multiplication result deviates from the expected result because IEEE floating arithmetic is not associative.

Moreover, even though it does not affect the timing of multiplication kernel, I declared an MPI datatype which consists four data that is sent to each process separately. Since the communication data is small, i.e. only 4 integers, reducing the number of collective communication function reduces the communication cost. Also, different optimizations such as 2D partition may be applied to reduce communication cost of multiplication kernel but code size would grow accordingly.