CIS 6930 Project 3 Report

Tae Seung Kang

A. Task 1 (OpenMP)

1. How to compile and run

Directory: Task1/src

Files: pagerank.c, makefile, facebook_combined.txt

I used Linux 'make' utility to compile and run. To compile and run the program, go to the directory Task1/src.

- To compile: type "make compile" in the command prompt

This will run the following command.

```
mpicc -fopenmp -o pagerank.o pagerank.c -lm
```

- To run: type "make run"

This will run the following command to execute the binary code pagerank.o.

```
time ./pagerank.o
```

- The default number of OMP threads is set to 4 and the default input file is set to facebook_combined.txt under the current working directory. You can edit makefile to change the OMP_NUM_THREADS value and the input graph file.

```
export OMP_NUM_THREADS=<numThreads>
```

usage: ./pagerank.o <input graph file>

e.g., `export OMP_NUM_THREADS=8`

```
./pagerank.o facebook_combined.txt
```

2. Implementation methods

- Undirected Facebook graph is used. I consider the edge "a - b" as both "a -> b" and "b -> a" links.

- An omp pragma directive is inserted for every "for loop" except for writing the final result to an output file.

- functions in pagerank.c: main(), pagerank(), init(), compute()

main(): master reads files and calculates the number of unique nodes N

pagerank(): calls init() and compute() functions

init(): initializes matrix A and the current vector R (R_n) and previous vector R_prev (R_{n-1}). R is initialized to a normalized identity vector using omp directives.

```
#pragma omp parallel default(none) shared(N, A, colsum, fp, R, R_prev, R0)
        {
                int ID = omp_get_thread_num();
                int nthreads = omp_get_num_threads();

                printf("ID = %d, nthreads = %d\n", ID, nthreads);

                #pragma omp for private(i, j, x)
                for (i = 0; i < N; i++) {
                        R_prev[i] = R[i] = R0;
                        A[i] = malloc(N*sizeof(double));
                        for (j = 0; j < N; j++) {
                                A[i][j] = 0;
                        }
                        colsum[i] = 0;
                }
        }
```

Also, the matrix A is initialized to a column stochastic matrix using omp directives.

```
#pragma omp parallel for default(none) \
                private(i, j, x) shared(N, A, colsum)
        for (i = 0; i < N; i ++) {
                for (j = 0; j < N; j ++) {
                        if (A[i][j] > 0) {
                                A[i][j] /= colsum[j];
                        }
                }
        }
```

compute(): performs power method iterations to compute page ranks using A and R_prev

R = (1 - d)/N + d*A*R_prev

Matrix vector multiplication (A*R_prev) is parallelized using omp directives.

```
#pragma omp parallel for default(none) \
                        private(i,j,sum) shared(N, A, R, R_prev, d)
reduction(+:totalsum)
                for (i = 0; i < N; i ++) {
                        sum = 0.0;
                        {
                                // A*R_prev
                                for (j = 0; j < N; j ++) {
                                        sum += A[i][j]*R_prev[j];
                                }
                                R[i] = (1 - d)/N + d*sum;
                                totalsum += R[i];
                        }
```

```
        }
```

The resulting vector R is normalized in parallel using omp directives.

```
#pragma omp parallel for default(none) \
                        private(i, j) shared(N, A, R, R_prev, totalsum, l1sum,
squaresum)
        for (i = 0; i < N; i ++) {
                //R[i] = ((1 - d)/N + d*R[i])/totalsum;
                R[i] = R[i]/totalsum;
                squaresum += pow(R_prev[i] - R[i], 2);
                R_prev[i] = R[i];
        }
```

- L2 norm is used for convergence threshold.

$\sum_i (R\_prev[i] - R[i])^2 < \varepsilon$ (epsilon)

3. Experimental settings

- Hardware: IBM Blade HS22 Intel Xeon 2.53GHz 8 cores with 16 threads (16 logical processors) with 24GB ram

- damping factor d = 0.85

- Convergence condition (threshold value) epsilon = 1e-7

- Input file: facebook_combined.txt

- Number of OpenMP threads: 1 to 16 (specifically 1, 4, 8, and 16)

4. Outputs and corresponding conclusions

- Output file: `Output_Task1.txt`

- Output format: node id and pagerank separated by a tab sorted by node id

```
nodeid  pagerank
0       0.006225
1       0.000236
2       0.000199
...
```

- Vector norm (L1, L2) for convergence condition is important for running time. With a single thread, L1 norm takes 60 iterations while L2 norm 40 iterations (epsilon = 1e-7).

- Experimental results

10 runs are executed for different number of threads (1, 4, 8, and 16) and their running times are averaged. L2 norm and damping factor 0.85 are used.

Running time comparison: average running time with error bars is shown below.

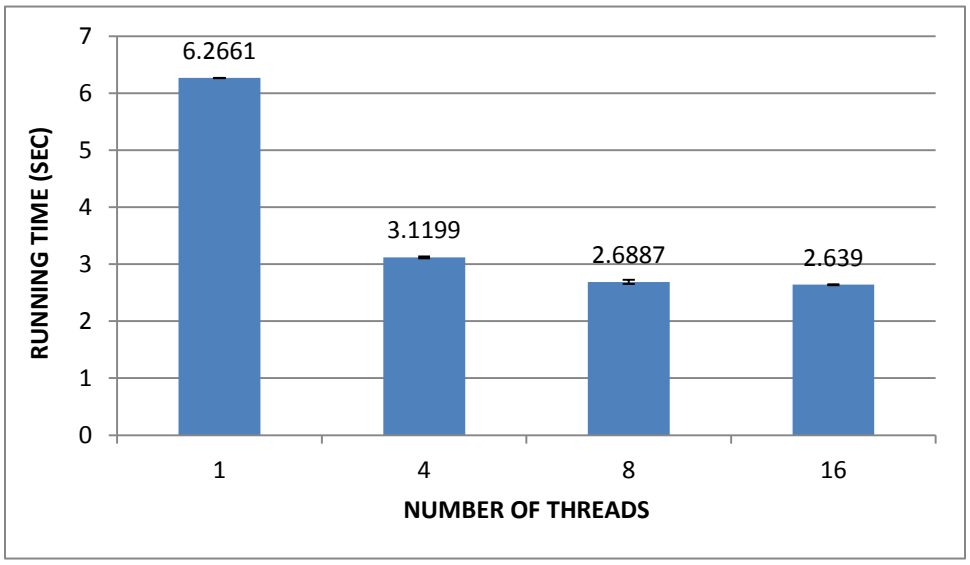


Figure 1. Running time comparison

The running time decreases with the number of threads increasing. This shows the benefits of using parallel computing. The running time is decreased by 50.21% to 57.88% compared to a single thread.

- Comparison of number of iterations: average number of iterations is shown below with error bars.

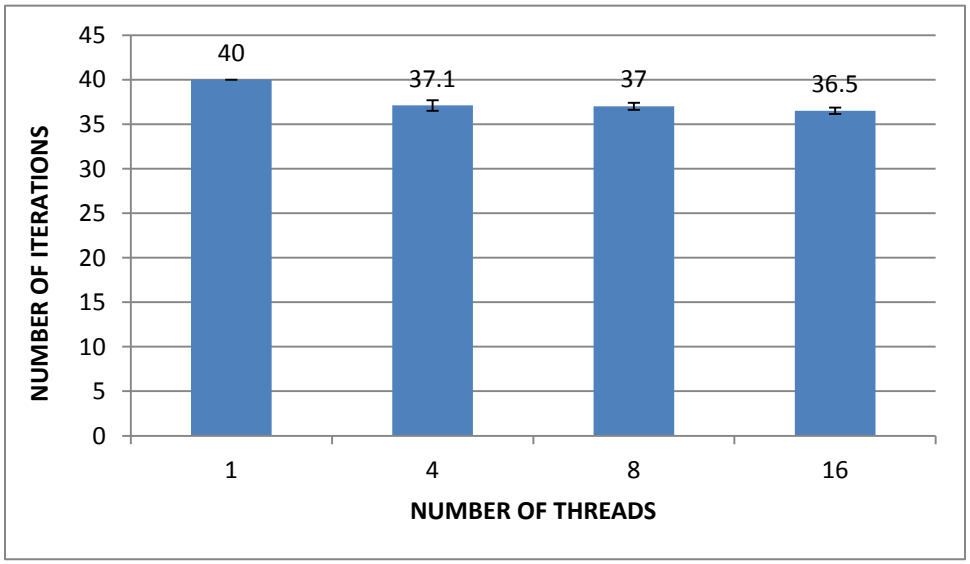


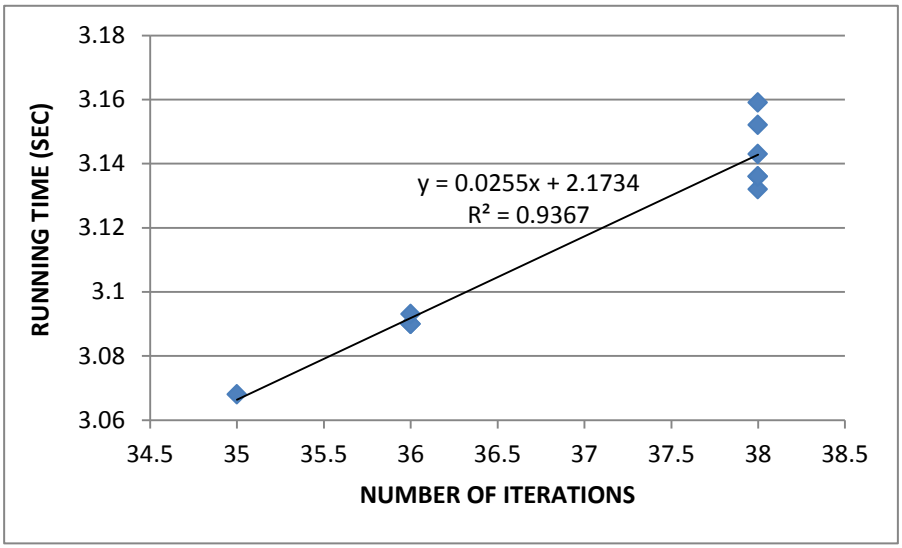
Figure 2. Comparison of number of iterations

In Figure 2, the number of iterations also decreases as the number of threads increases. An interesting observation to make here is that sequential processing yields always the same result (40 iterations) while parallel processing does not guarantee the same result for each run (35-37 iterations). This indicates the non-deterministic behavior with openmp threads. That is, every time I run with openmp multithreads, different results come out. This could possibly be because of roundoff errors and the order of computation.

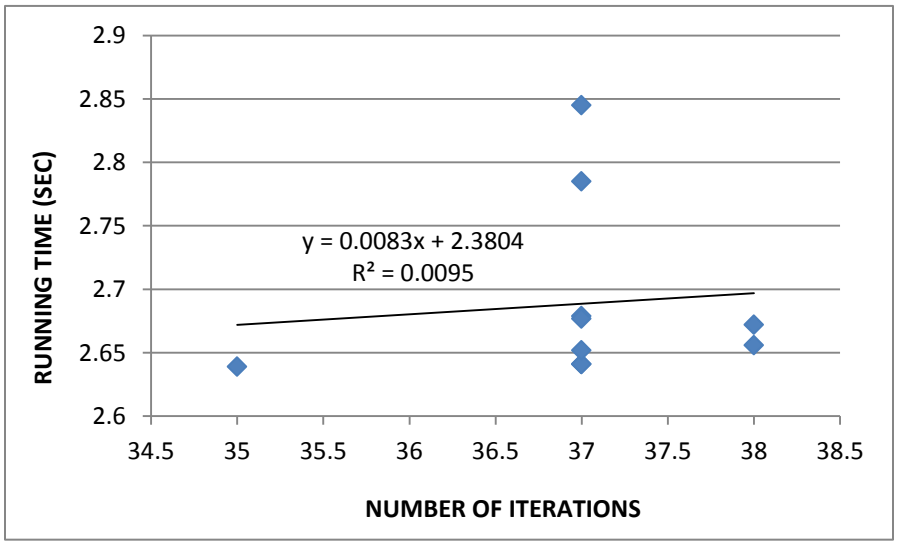
- Linear regression on number of iterations and running time

It appears that there is a linear relationship between the number of iterations and the running time. The smaller the number of iterations, the shorter the running time. Below is the results of linear regression.

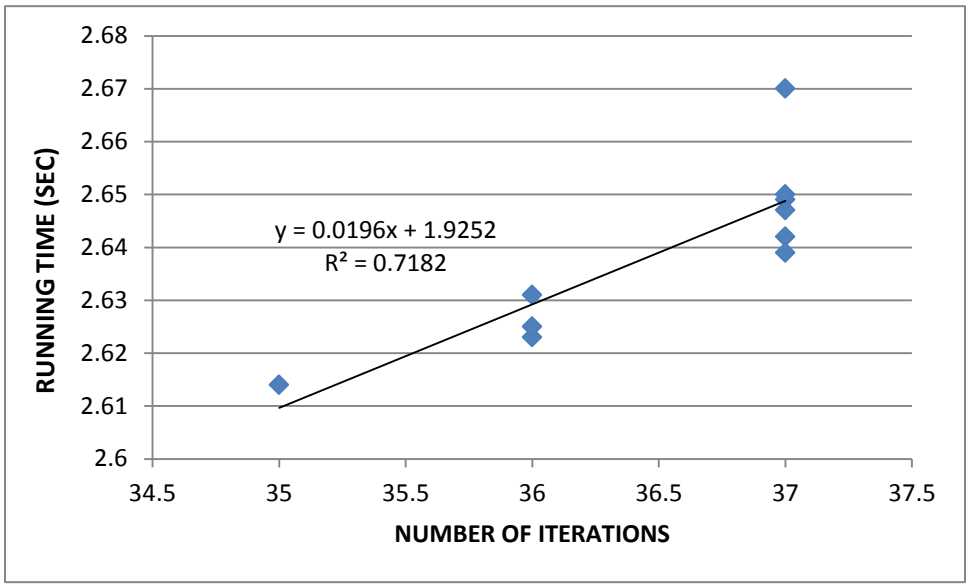
4 threads: very good fit



8 threads: poor fit



16 threads: good fit



Although not always fit on linear regression, there seems to be certain relationship between the number of iterations and the running time.

- Raw data: Below is the raw data used to plot the figures above.

# of threads	1		4		8		16	
	running time	iterations	running time	iterations	running time	iterations	running time	iterations
run1	6.265	40	3.09	36	2.785	37	2.647	37
run2	6.261	40	3.09	36	2.641	37	2.649	37
run3	6.274	40	3.093	36	2.672	38	2.65	37
run4	6.265	40	3.159	38	2.656	38	2.642	37
run5	6.267	40	3.132	38	2.641	37	2.67	37
run6	6.258	40	3.152	38	2.679	37	2.614	35
run7	6.264	40	3.068	35	2.677	37	2.623	36
run8	6.259	40	3.136	38	2.845	37	2.639	37
run9	6.273	40	3.136	38	2.652	37	2.625	36

run10	6.275	40	3.143	38	2.639	35	2.631	36
average	6.2661	40	3.1199	37.1	2.6887	37	2.639	36.5
stdev	0.006136	0	0.031558	1.197219	0.069642	0.816497	0.016316	0.707107

B. Task 2 (MPI)

1. How to compile and run

Directory: Task2/src

Files: reducer.cpp (253 lines), makefile, 100000_key-value_pairs.csv

I use Linux 'make' utility to compile and run. To compile and run the program, go to the directory Task2/src.

- To compile: make compile

This will run the following command.

```
mpic++ -o reducer.o reducer.cpp -std=c++0x
```

- To run: make run

This will run the following command to execute the binary code reducer.o with all the processors available.

```
time mpirun -np `nproc` ./reducer.o
```

- You can edit makefile to change the input graph file and/or to run MPI processes. The default input file is set to 100000_key-value_pairs.csv under the current working directory.

usage: mpirun -np <number of processes to run> ./reducer.o <input graph file>

Examples) `mpirun -np 3 ./reducer.o`

`mpirun -np 4 ./reducer.o 100000_key-value_pairs.csv`

2. Implementation methods

- functions in reducer.cpp file: main()

Steps in main() function: initialization, first step, second step, and final step

INITIALIZATION: read file and insert key-value pairs into a table

FIRST STEP: partition and perform local reduction on own table. Values for the same key are summed.

SECOND STEP: exchange the results of local reduction with other processors. Processor maps pairs to corresponding processor and inserts into an array list (vector type). Each processor sends the selective key-value pairs to corresponding processors and receives the keys assigned to it with associated sum values from others. The following is the code snippet for MPI communication.

```
MPI_Sendrecv(&data[0][0],
        size * 2,
        MPI_INT, i, 1,
        &recv[0][0],
        recvsize * 2,
        MPI_INT, i, 1,
        MPI_COMM_WORLD, NULL
 );
```

Upon receiving the pairs, processor merges them into the existing table.

FINAL STEP: perform second local reduction and write into file. Processor writes only its own pairs into the output file which requires synchronization between processes. This is resolved by taking turn using a barrier (MPI_Barrier). More specifically, a turn variable increases from 0 to the number of processors – 1 by one. Each processor writes only when the turn variable becomes its rank. The writes are appended.

```
while(turn < nprocs) {
        MPI_Barrier(MPI_COMM_WORLD);
        if(turn == myrank) {

                outfile.open (OUTFILE, ios::app);

                for (auto it = partable.begin(); it != partable.end(); ++it) {
                        key = it->first;
                        value = it->second;

                        if (first <= key && key <= last) {
                                outfile << key << '\t' << value << endl;
                        }
                }

                outfile.close();
        }
        turn ++;
}
```

- For data structures, C++ standard library (STL) is used. For hash tables, unordered_map is used at first. Later, however, I realized that the output keys are not sorted. So, I changed to use map data structure in STL.

- Hard to send vector type: When sending a key-value pair list to each processor, I had to convert the vector of key-value pairs to a 2-dimensional int array.

3. Experimental settings

- Hardware: IBM Blade HS22 8 cores with 16 threads (16 logical processors) with 24GB ram

- Input file: 100000_key-value_pairs.csv

- Number of processes: 1 to 16 (1, 4, 8, and 16)

- Number of runs for each process: 5

4. Outputs and corresponding conclusions

- Output file: Output_Task2.txt

- Output format: key and sum of values separated by a tab sorted by key

```
Key     sum-of-values
0        63
1        79
2        75
...
```

- In order to identify the benefits of using MPI, an experiment has been performed. I varied the number of MPI processes to measure the running times. The results of 5 runs are averaged and standard deviation is also computed.

Running times according to the number of processes with error bars:

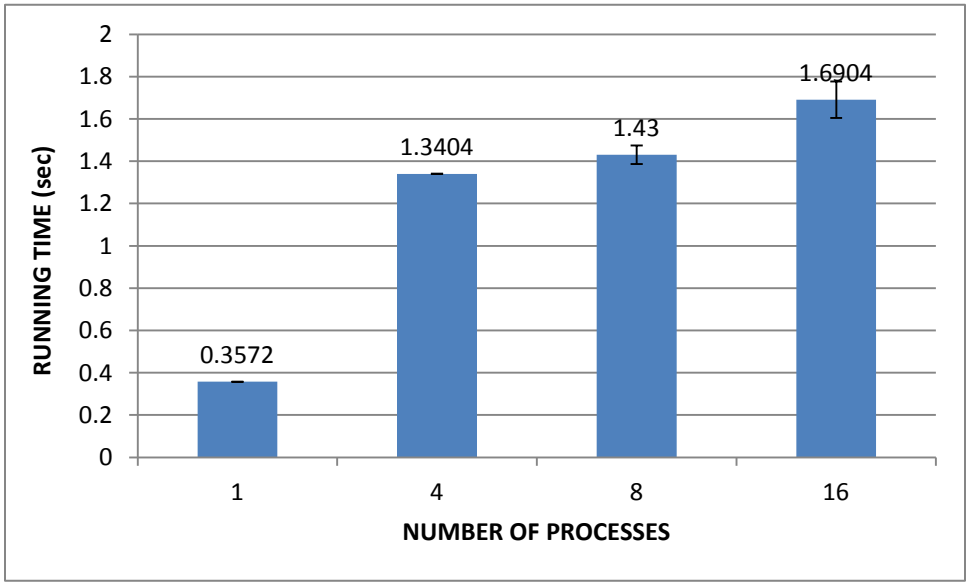


Figure 3. Average running time according to the number of MPI processes

Standard deviations of running times:

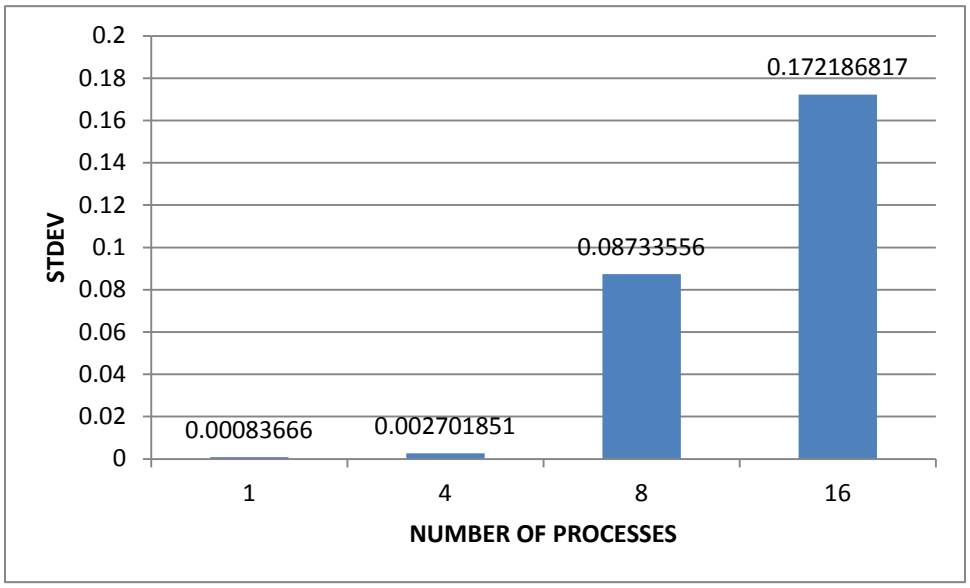


Figure 4. Standard deviation of the running times

Raw data for the above experiment:

Number of processes	1	4	8	16
run1	0.358	1.34	1.36	1.582
run2	0.358	1.336	1.531	1.584
run3	0.356	1.342	1.382	1.691
run4	0.357	1.343	1.519	1.988
run5	0.357	1.341	1.358	1.607
average	0.3572	1.3404	1.43	1.6904
stdev	0.000837	0.002702	0.087336	0.172187

- Conclusion: First off, it is observed that the running time increases as the number of MPI processes increases as shown in Figure 3. Put another words, a single processor performs faster than multiprocessors. This is because the dataset size is too small and the computation is too simple to take advantages of parallel processing. Rather, communication overhead (MPI_Sendrecv) would've dominated the running time. Also, synchronization control like MPI_Barrier should be the bottleneck. To take performance gain from parallel processing using MPI, the dataset size should be much larger and

the job should preferably be computation-oriented so that the communication overhead can be overwhelmed by the gain from parallel processing.

Another interesting observation is the deviation of the running times. In Figure 4, the standard deviation increases as the number of MPI processes increases. This is because the extent of nondeterministic behavior of multiple processes increases with the increasing number of processes whereas that of a single process is almost none.