Final Project Report

Bihua Yu

December 6, 2018

1 Project description

- Write a program to generate an $N \times N$ matrix A. Randomly selected 95% of the matrix's elements are strictly zero while the reminders are floating-point random numbers uniformly distributing in (0.0, 10.0).
- Write a parallel program to compute the dominant eigenvalue and the associated eigenvector on $P = 2^0 \times 28, 2^1 \times 28, 2^2 \times 28$ cores for $N = 28 \times 2^{10}, 28 \times 211, 28 \times 2^{12}$. Obviously, the linear algebra results do not depend on P for a given N. If power method is used, you need to compute A^k where $k \geq 6$.
- Collect performance results for the above 3×3 cases and tabulate them.
- Plot the speedup curves.

2 Algorithm description

Power method is used in this task. A sequential power method algorithm is described as Algorithm 1 shows.

In order to parallelize this algorithm, first we need to distribute the data. Considering the matrix-vector multiplication is a dominate computation in this algorithm, we may have two good options for a partition of the data M, x, y, in order to parallelize the matrix-vector multiplication.

The first way is:

- M: row-wise block distributed. Each process has about N/P rows of M
- x: block distributed. Each process has N/P elements of x
- y: block distributed. Each process has N/P elements of y

The second way for a partition is

Algorithm 1 Sequential algorithm

```
1: M = mat_gen(N);                          ▷ Generate a random $N \times N$ matrix $M$
2: $|x| = 1$;                               ▷ Make an initial guess $x$ and normalize it
3: $\lambda_0 = 0$; $i = 1$;
4: while true do
5:     $y = Ax$;
6:     $\lambda_i = (y, y)/(y, x)$;         ▷ Compute an approximate eigenvalue
7:     if $|\lambda_i - \lambda_{i-1}| < eps$ then
8:         break;                           ▷ It converges, and exit
9:     else
10:        $|y| = 1$;                       ▷ Normalize x
11:        x = y;
12:        i = i+1;
13: $|y| = 1$;                              ▷ Normalize y and get the eigenvector
```

- M: row-wise distributed. Each process has about N/P rows of M
- x: Allocate redundant vector with N dimension for all processes
- y: block distributed. Each process has N/P elements of y

If we want to randomly initialize the vector x, the first way is preferred. This is because in the first way, each process can generate the elements of x simultaneously. However, with the second partition, communication among processes is necessary in order to keep the same x in all processes.

Having the partition in the first way, we can parallelize the algorithm in several places. They have been marked by Algorithm 2. In the following, I will explain how each part is parallelized.

2.1 Parallelized 1: parallelized generation of a random matrix

Each element of M, m, has two status — zero or nonzero. If m follows Bernoulli distribution with a probability $p = 5\%$ of not being a zero, which is, $m \sim B(1, p)$, then, the number of non-zeros in M, n, follows Binomial distribution, which is $n \sim Bin(N^2, p)$. Thus, the expectation of $n = N^2 p = 0.05N^2$, which is exactly what we want. Therefore, we can just generate each element m individually and simultaneously in each process, with a Bernoulli distribution $B(1, p)$. However, if the generated m is non-zero, we need to generate it again in the range (0,10) uniformly, in order to decide the value of m.

2.2 Parallelized 2, 4, 5, 6: parallelized vector-vector dot product and vector normalization

In order to normalize a vector x, we need to obtain the dot-product (x, x) first. Since both x and y are block distributed, they take the same strategy to be normalized. Taking normalizing x as an example. On each process, we have $x_i \times x_i$. Then, we sum up all x_i^2 together and distribute the summation s_x using $MPI\_Allreduce()$. On each process, do x_i/s_x. Thus, normalizing x is completed.

2.3 Parallelized 3: parallelized matrix-vector multiplication

To parallelize the matrix-vector multiplication, we take the strategy talked in class. We rotate the x among different processes. And for each rotation, we multiply corresponding elements in M by x on the same process. After one cycle, we will get an updated y which is block distributed on processes.

Algorithm 2 Sequential algorithm

```
 1: M = mat_gen(N);                          ▷ Parallelized 1
 2: $|x| = 1$;                               ▷ Parallelized 2
 3: $\lambda_0 = 0$; $i = 1$;
 4: while true do
 5:     $y = Ax$;                            ▷ Parallelized 3
 6:     $\lambda_i = (y, y)/(y, x)$;         ▷ Parallelized 4
 7:     if $|\lambda_i - \lambda_{i-1}| < eps$ then
 8:         break;
 9:     else
10:         $|y| = 1$;                       ▷ Parallelized 5
11:         x = y;
12:         i = i+1;
13: $|y| = 1$;                               ▷ Parallelized 6
```

3 compile- and run-time options

There are two folders: “program” and “timehw4”.

program: It contains the program which the project question asked for.
Files contained are: fun.h, fun.cc, pow.cc

compile-option: mpiicpc -std=c++11 -c fun.c pow.cc; mpiicpc -std=c++11 -o pow fun.o pow.o

run-time option: mpirun -n P ./pow N

timehw4: It is for performance analysis. In order to analyze the performance, instead of

generating matrix randomly, I wrote a separate program"mat.cc" to control the input. In this way, we can compare the program with difference numbers of processes but the same input.
To generate a random $N \times N$ matrix in a file "myfile":
compile-option: mpiicpc -std=c++11 -c mat.cc; mpiicpc -std=c++11 -o mat mat.o
run-time option: ./mat N myfile
To test the program with myfile:
compile-option: mpiicpc -std=c++11 -c fun.c pow.cc; mpiicpc -std=c++11 -o pow fun.o pow.o
run-time option: mpirun -n P ./pow N myfile

4 Results

Noting in the tables and figures $pi = 28 \times 2^i, ni = 28 \times 2^{10+i}, 1pmeansasingleprocess$. "set" indicates the number of process P and input N. #iter indicates with this input N, after how many iterations, the solution convergences and the program stops. I test 10 examples for each setting. So $A(t)$ means the average time of the 10 runs for one setting — one specific combination P and N. A(t)/iter means the average time costs for each iteration. $speedup = \frac{A(t)/iterwith1process}{A(t)/iterwithPprocess}$

set	#iter	A(t)/iter	speedup
p0n0	39	0.000343409769230769	0.0195249660997802
p1n0	39	0.000233079897435897	0.0287672346535512
p2n0	39	0.000322449102564103	0.0207941782105942
p0n1	29	0.000361264965517241	0.0234830219180038
p1n1	29	0.000398878172413793	0.021268632104159
p2n1	29	0.000402627344827586	0.0210705835369456
p0n2	16	0.00051988025	0.0180230807960102
p1n2	16	0.0005188059375	0.0180604019205158
p2n2	16	0.0005224555625	0.0179342405795517
1pn0	39	0.0000067050641025641	1
1pn1	29	0.00000848359310344828	1
1pn2	16	0.00000936984375	1

5 Analysis

Overall, we can see that sequential algorithm performs much better than the parallel algorithm. This could be caused by too much communications in parallel algorithm. According to the design of the algorithm, except parallelized 3, which has synchronous communication, the other 5 places(parallelized 1, 2, 4, 5, 6) all need $MPI\_Reduce()$, which results in

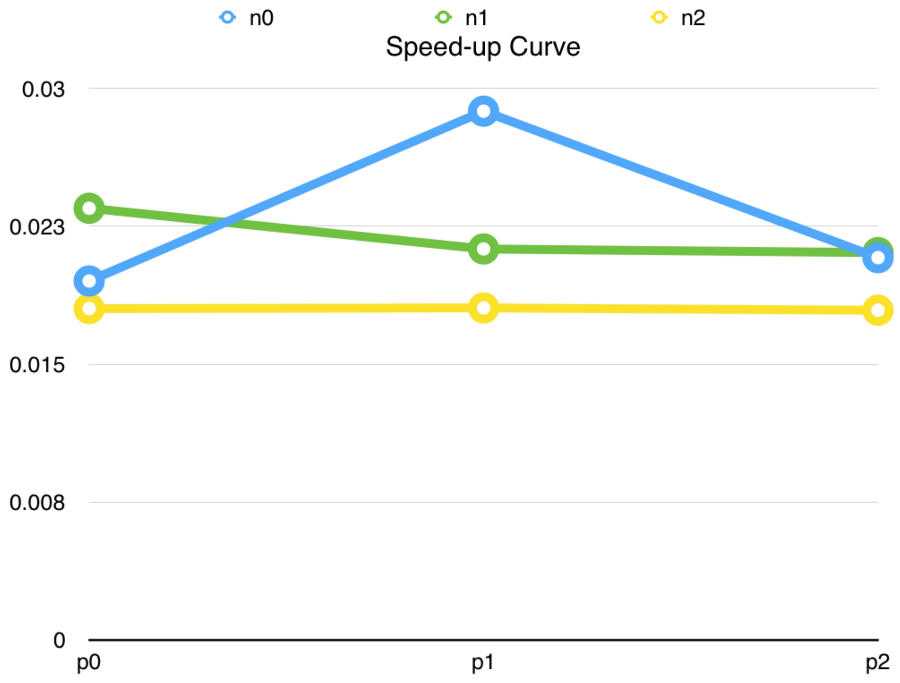

Figure 1: Speed up curve

a big time cost. Because of the big overhead, speed up ratio is much smaller than 1, rather than greater than 1.

On the other hand, the overhead is not only heavy, but grows as N grow, which makes even increasing N is not necessary increase speed up ratio. In our p1 test case, it even decreases the speed up ratio as N grows, which implies the communication also grows with N.