Parallelization of Power Method

Bihua

December 6, 2018

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 $k \geq 6$.

Project description

- Collect performance results for the above 3×3 cases and tabulate them.
- Plot the speedup curves.

Data Partition 1

- 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

Data Partition 2

- 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

Sequential algorithm

Algorithm 1 Sequential algorithm

```
1:  M = mat_gen(N);                          ▷ Generate a random N × N matrix M
2:  |x| = 1;                                 ▷ Make an initial guess x and normalize it
3:  λ_0 = 0; i = 1;
4:  while true do
5:      y = Ax;
6:      λ_i = (y, y)/(y, x);                 ▷ Compute an approximate eigenvalue
7:      if |λ_i − λ_{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
```

Figure: Sequential algorithm

Parallel algorithm

Algorithm 2 Sequential algorithm

```
1:  M = mat_gen(N);                                   ▷ Parallelized 1
2:  |x| = 1;                                          ▷ Parallelized 2
3:  λ_0 = 0; i = 1;
4:  while true do
5:      y = Ax;                                       ▷ Parallelized 3
6:      λ_i = (y, y)/(y, x);                          ▷ Parallelized 4
7:      if |λ_i − λ_{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
```

Figure: Parallel algorithm

Parallel algorithm

Algorithm 2 Sequential algorithm

```
1:  M = mat_gen(N);                                   ▷ Parallelized 1
2:  |x| = 1;                                          ▷ Parallelized 2
3:  λ_0 = 0; i = 1;
4:  while true do
5:      y = Ax;                                       ▷ Parallelized 3
6:      λ_i = (y, y)/(y, x);                          ▷ Parallelized 4
7:      if |λ_i − λ_{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
```

Figure: Parallel algorithm

Parallelized 1: parallelized generation of a random matrix
Element of M: $m \sim B(1, p)$
The number of non-zeros in M: $n \sim Bin(N^2, p)$
$E(n) = N^2 p = 0.05 N^2$

Parallel algorithm

Algorithm 2 Sequential algorithm

```
1:  M = mat_gen(N);                                   ▷ Parallelized 1
2:  |x| = 1;                                          ▷ Parallelized 2
3:  λ_0 = 0; i = 1;
4:  while true do
5:      y = Ax;                                       ▷ Parallelized 3
6:      λ_i = (y, y)/(y, x);                          ▷ Parallelized 4
7:      if |λ_i − λ_{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
```

Figure: Parallel algorithm

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

Parallel algorithm

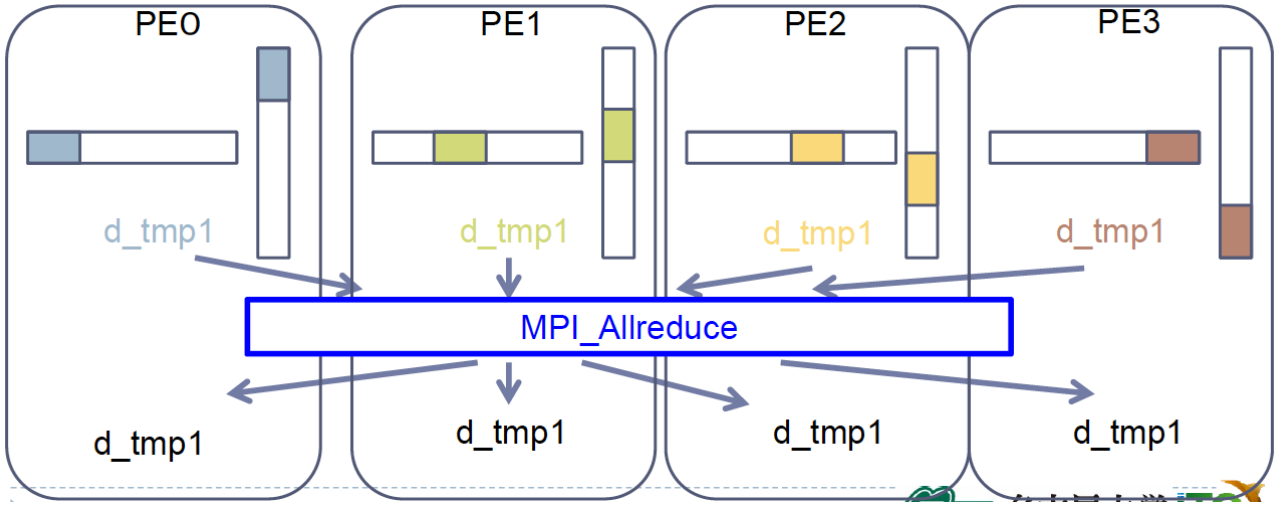

Figure: Parallel algorithm

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

Picture is from: Takahiro Katagiri, Professor, Information Technology Center, Nagoya University

Parallel algorithm

Algorithm 2 Sequential algorithm

```
1:  M = mat_gen(N);                                   ▷ Parallelized 1
2:  |x| = 1;                                          ▷ Parallelized 2
3:  λ_0 = 0; i = 1;
4:  while true do
5:      y = Ax;                                       ▷ Parallelized 3
6:      λ_i = (y, y)/(y, x);                          ▷ Parallelized 4
7:      if |λ_i − λ_{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
```

Figure: Parallel algorithm

Parallelized 3: parallelized matrix-vector multiplication

Parallel algorithm

Method 2: Row Partition Both A & B

Row-partition matrix A:

$$\begin{pmatrix} A_{11}@1 & \cdots & A_{1n}@1 \\ \vdots & \ddots & \vdots \\ A_{m1}@P & \cdots & A_{mn}@P \end{pmatrix}$$

Row-partition vector B:

$$\begin{pmatrix} B_1@1 \\ B_2@2 \\ \vdots \\ B_n@P \end{pmatrix}$$

Figure: Parallel algorithm

Parallelized 3: parallelized matrix-vector multiplication

Picture is from: Yuefan Deng, Professor, Applied Math and Statistics, Stony Brook University

Result

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	0.0000048359310344828	1
1pn2	16	0.00000936984375	1

Figure: Case Comparison

$pi = 28 \times 2^i, ni = 28 \times 2^{10+i}, 1p = asingleprocess$. #iter indicates after how many iterations, the solution convergences and the program stops. A(t)/iter means the average time costs for each iteration. $speedup = \frac{A(t)/iter\ \ with\ \ 1\ \ process}{A(t)/iter\ \ with\ \ P\ \ process}$

Result

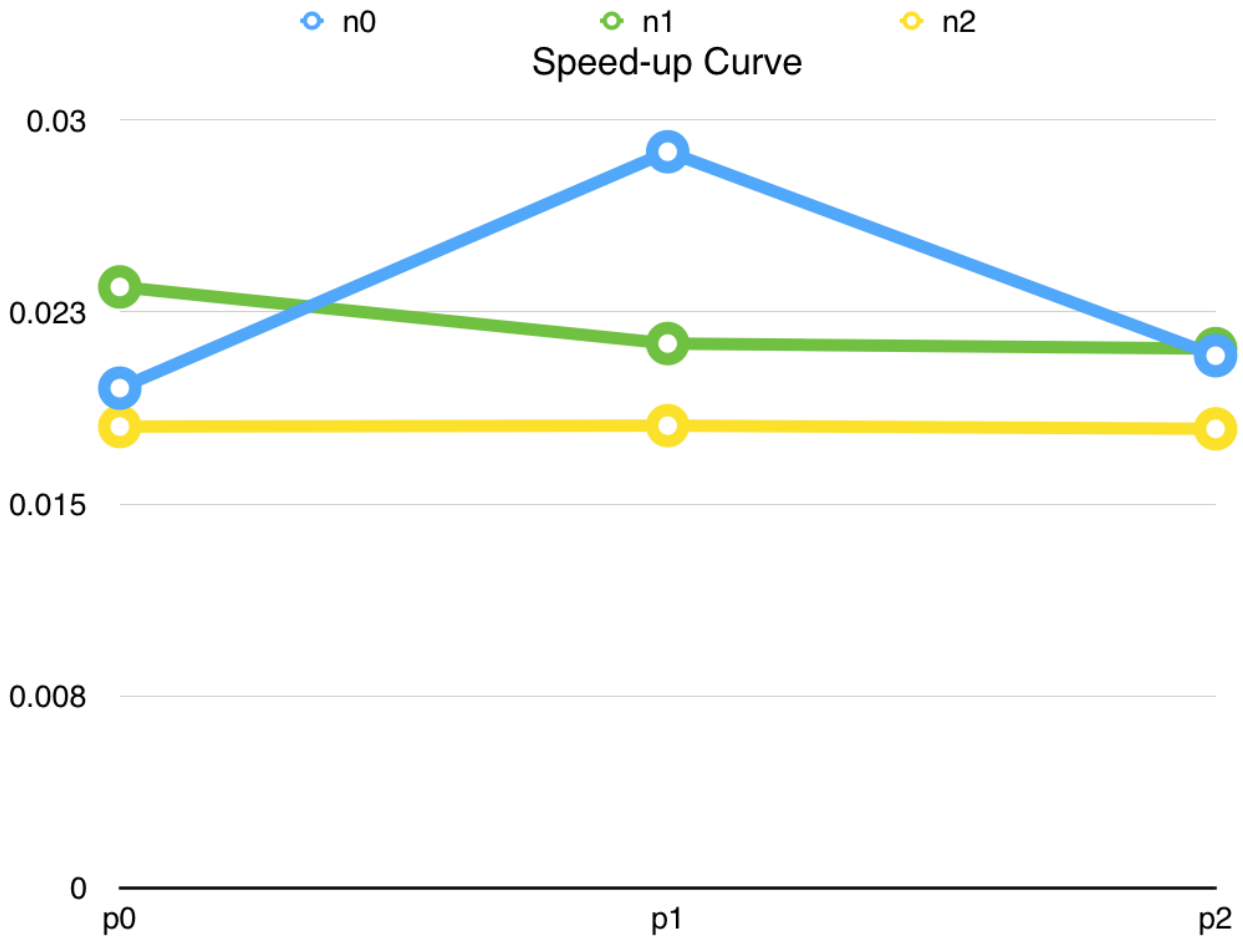

Figure: Speedup curves