MPI 矩阵-向量乘法

一、学生信息

姓名：柳建国
学号：2022Z8017782089
专业：电子信息
所部：数字所

二、问题描述

对于矩阵 A（如图 1）和向量 $x = (1,2,3,1,2,3,\ldots)$，计算向量 $y = A \times x$（如图 2）。

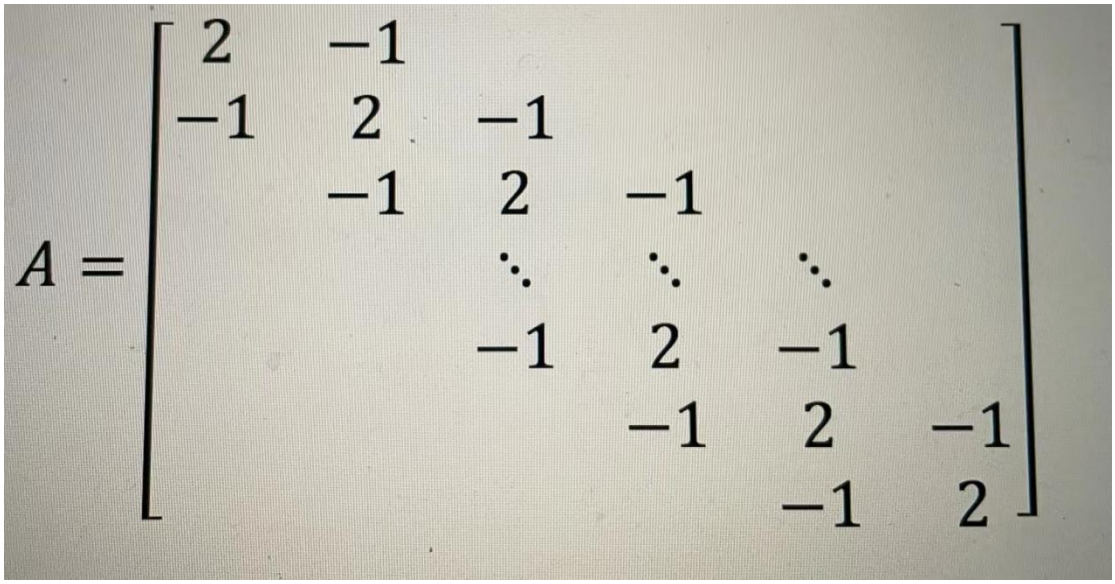

图 1 矩阵 A 形式

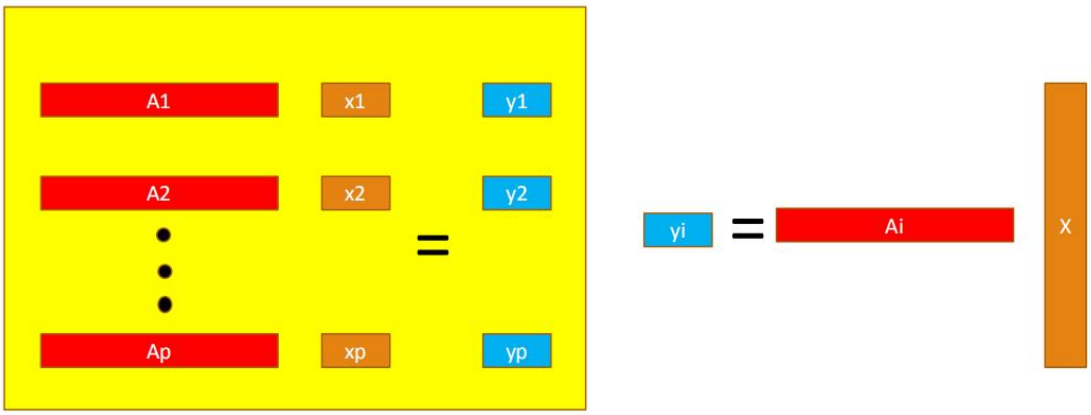

图 2 向量乘示意图

三、测试环境（系统，CPU，核心）

测试环境配置见表 1。

表 1 运行环境配置

配置	参数
系统	Ubuntu 18.04
CPU	Inter(R) Core(TM)i5-10505 CPU@5.20GHz 3.20Ghz
核心	虚拟机配置 4 核心
内存大小	8G

四、OpenMP 程序的性能评估

1. 运行时间

表 2 不同数量进程和各种大小矩阵下，程序运行时间

Comm_sz	Order of Matrix				
	1024	2048	4096	8192	16384
1	0.01178775	0.018478	0.0381435	0.152634	0.60850075
2	0.005616	0.01059925	0.019324	0.07757425	0.31698025
4	0.004279	0.00474175	0.01009075	0.0401635	0.15757725

2. 加速比

表 3 不同数量进程和各种大小矩阵下，程序运行加速比

Comm_sz	Order of Matrix				
	1024	2048	4096	8192	16384
1	1.00	1.00	1.00	1.00	1.00
2	2.10	1.74	1.97	1.97	1.92
4	2.75	3.90	3.78	3.80	3.86

3. 效率

表 4 不同数量进程和各种大小矩阵下，程序运行效率

Comm_sz	Order of Matrix				
	1024	2048	4096	8192	16384
1	1.00	1.00	1.00	1.00	1.00
2	1.05	0.87	0.99	0.98	0.96

4	0.69	0.97	0.95	0.95	0.97

4. 可扩展性

如果一个技术可以处理规模不断增加的问题，那么它就是可扩展的。对于并行程序而言，可扩展性有明确定义。

假设我们运行一个并行程序，固定进程或线程数目、固定问题规模，得到一个效率值 E。现在我们增加程序所用的进程或线程数目，如果在问题规模也同比例增加情况下，该程序效率值一直都是 E，那么我们就称该程序是可扩展的。

如果在增加进程或线程个数时，可以维持固定效率，却不增加问题规模，那么程序成为强可扩展(strongly scalable)。如果在增加进程或线程个数时，只有以相同倍率增加问题规模才能保持效率值，那么程序就称为弱可扩展(weakly scalable)。

在不考虑进程间通信的情况下，通过以上实验结果，可以看到，随着进程数量增加，程序执行的时间在线性减小，所以该程序算法是强可扩展性的。

五、源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

//Matrix number N*N
#define N 16384

//claim function
int ** makeMatrixA(int matrixsize);
int * makeMatrixx(int matrixsize);

//this is main function
int main(int argc, char *argv[]){
    //make Matrix
    int ** MatrixA = makeMatrixA(N);
    int * Matrixx = makeMatrixx(N);

    //start time
    double start_time, end_time;
    start_time = omp_get_wtime();

    //malloc memory to store result
    int * y = (int *)malloc(sizeof(int)*N);
```

```
	#pragma omp parallel for default(shared) num_threads(4)
	for(int i=0; i<N; i++){
		y[i]=0;
		for(int j=0; j<N; j++){
			y[i] += MatrixA[i][j] * Matrixx[j];
		}
	}

	//end time
	end_time = omp_get_wtime();

	//print result
	printf("Execute time is %lf.\n", end_time-start_time);
	printf("The result is:");
	for(int i=0; i<N; i++){
		printf("%d ", y[i]);
	}
	printf("\n");

	// free memory
	free(MatrixA);
	free(Matrixx);
	free(y);
}

int ** makeMatrixA(int matrixsize){
	int **MatrixA = (int **)malloc(sizeof(int *)*matrixsize);
	for(int i=0; i<N; i++){
		MatrixA[i] = (int *)malloc(sizeof(int)*matrixsize);
	}
	for(int i=0; i<matrixsize; i++){
		for(int j=0; j<matrixsize; j++){
			if(j==(i-1)||j==(i+1)){
				MatrixA[i][j]=-1;
			}
			else if(j==i){
				MatrixA[i][j]=2;
			}
			else{
				MatrixA[i][j]=0;
			}

		}
	}
```

```
	return MatrixA;
}

int * makeMatrixx(int matrixsize){
	int * Matrixx=(int *)malloc(sizeof(int)*matrixsize);
	for(int i=0; i<matrixsize; i++){
		if(i%3 == 0){
			Matrixx[i] = 1;
		}
		else if(i%3 ==1){
			Matrixx[i] = 2;
		}
		else{
			Matrixx[i] = 3;
		}
	}
	return Matrixx;
}
```