Homework 4

School of Data and Computer Science
17341015 Hongzheng Chen

1.Problem 1

Consider a sparse matrix stored in the *compressed row format* (you may find a description of this format on the web or any suitable text on sparse linear algebra). Write an OpenMP program for computing the product of this matrix with a vector. Download sample matrics from the Matrix Market (`http://math.nist.gov/MatrixMarket`) and test the performance of your implementation as a function of matrix size and number of threads.

Answer. Since the matrix files downloaded from the Matrix Market are not in compressed row format, we have to transform the format first. One matrix is stored using three arrays including `row_offset`, `col_index`, and `val`. When we obtain the three arrays, we can do the multiplication as the pseduocode shown below.

```
for (int i = 0; i < numRows; ++i) {
    int offset = (i == numRows-1 ? numNonZeros : row_offset[i+1]);
    for (int j = row_offset[i]; j < offset; ++j)
        res[i] += vec[col_index[j]] * val[j];
}
```

The `vec` is a predefined vector

$$\mathbf{v} = \begin{bmatrix} 1 & 2 & \cdots & n \end{bmatrix}^{\mathrm{T}}$$

used for matrix-vector multiplication.

The code shown above is a sequential program for baseline comparison. To obtain the OpenMP parallel program, we can directly add `#pragma omp parallel for` before the outer for loop, since no dependency between two loops.

The experiment leverage ten datasets varying sizes collected from the Matrix Market, shown in Table 1.

Table 1: Benchmarks for experiments

Benchmark	# of rows	# of cols	# of non-zeros
gre	115	115	421
saylr	238	238	1128
bus	1138	1138	2596
nos	960	960	8402
orsreg	2205	2205	14133
sherman	1080	1080	23094
orani	2529	2529	90158
bcsstk	4884	4884	147631

And the machine used for the experiment equips with a Intel Core i7-7700HQ processor (8 cores) running at 2.80GHz and a 8GB memory. The programs are compiled with gcc 7.3.0 and running on Ubuntu 18.04(LTS).

The experimental results[1] are shown in Fig 1. Notice the scalability results of number of threads are for the biggest benchmark `bcsstk`.

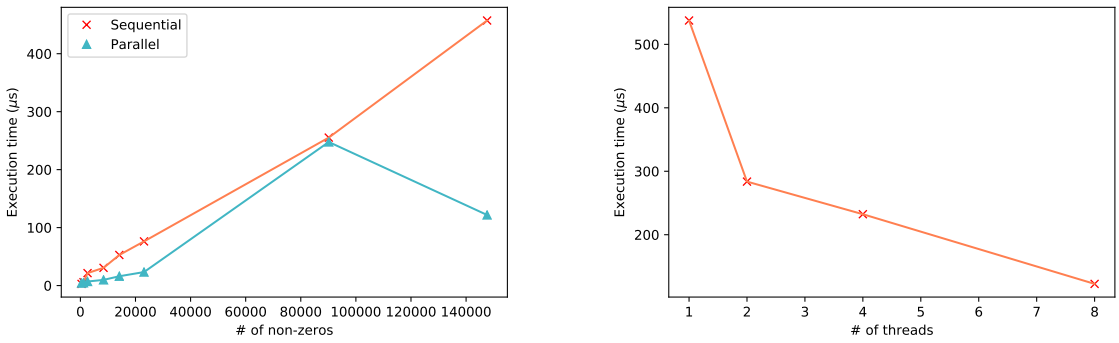

Figure 1: Scalability of matrix size and number of threads

From Fig. 1, we can see that as the size of the matrix scales, the difference of the execution time between parallel and sequential programs becomes larger. When more threads are considered, the speedup of the parallel program becomes larger, which is expected.

But we can also see that for the `orani` benchmark, the running time of parallel program is similar to the sequential program, which results from the specific structure of the input matrix. The matrix is asymmetric and unstructured (shown in Fig. 2), thus using compressed row format to store and compute the matrix may not be a good choice.

[1]The figures of the experiments are created with Python Matplotlib, and the notebook *plot.ipynb* is also in the attachment.

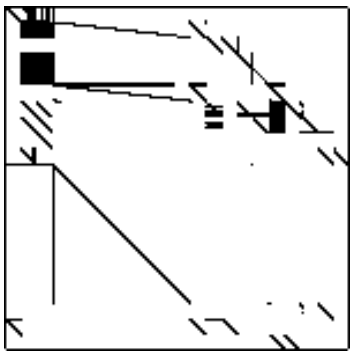

Figure 2: Orani's matrix structure

I also add other features into my program for high productivity.

- Use `template` and `typedef` for flexibility of different value types.
- Leverage `<chrono>` header for high-accuracy timing.
- Write a `Makefile` for auto compiling and experimenting.

The program `matmul.cpp` can be found in the attachment.

2.Problem 2

Implement a producer-consumer framework in OpenMP using `sections` to create a single producer task and a single consumer task. Ensure appropriate synchronization using locks. Test your program for a varying number of producers and consumers.

Answer. Similar to Project 1, we can emulate a queue by a fixed-size array, whose head and tail are connected as a ring. Two operations `push` and `pop` are implemented for the queue. In each operation. we leverage `#pragma omp critical` to ensure the mutuality.

Each producer produces one number and push it into the queue. The consumers pop the numbers from the queue and add them together.

The producers and consumers are encapsulated in `omp section`. All the sections are put in a large block `omp parallel sections`, which enables the sections to run in parallel.

The experimental results are shown in Table 2. Due to time limitation, only several experiments are conducted here.

Table 2: Running time of the producer-consumer framework

# of producers	# of consumers	Running time (ns)
2	2	2499500
2	6	3709500
4	4	2148600
6	2	1785100
8	8	2210600

From Table 2, we can see that more consumers lead to slower running time, which may result from the heavy conflicts of popping. Once the number is produced, it may immediately be consumed by the consumers. From these limited cases, we can also conclude that the more producers may be better for the performance. Since no larger cases are tested, the conclusion may be different when the input size scales.

The program `prod-cons.cpp` can be found in the attachment. Notice at this time, the number of producers and consumers cannot be set automatically. The default setting is two producers with two consumers.