CS 314 Principles of Programming Languages

OpenMP Parallel Programming Project

In this project, you will implement a parallelized version of sparse matrix vector multiplication (*spmv*). You are given a sequential C implementation of the *spmv* code and your job is to parallelize it using OpenMP. The following four sections will explain the Sequential C implementation and clarify what kind of parallelism you should be restricted to during this project.

1 Background

1.1 Matrix-vector Multiplication

Matrix-vector multiplication is an important and fundamental operation in many computation problems. Given $A \in R^{M \times N}$, $b \in R^{N \times 1}$, the matrix-vector multiplication result vector $res \in R^{M}$ is obtained by

$$res = Ab \tag{1}$$

where $res_i = \sum_{j=1}^{N} A_{ij} b_j$, res_i is the i-th entry of res, b_j is the j-th entry of b, A_{ij} is the entry of A at row i and column j.

The complexity of calculating res is $O(MN)$. However, if a matrix has a large number of zeros, we can reduce the number of multiplication operation because any number multiplied by zero is zero. This type of matrices are called sparse matrices. For example, if

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{pmatrix},$$

regardless whether b is sparse, only five multiplications are needed to calculate res.

Apparently the calculation of each $res_i, i = 1, 2..., M$ is independent. In this project you will exploit the parallelism in sparse matrix vector multiplication and parallelize it using **OpenMP**.

1.2 Handling Sparse Matrix Input

In this project you work with sparse matrices stored in Coordinate Form (COO). Specifically, we use the sparse matrices from matrix market [1] with *mtx* file extension name. COO stores a list of (row, column, value) tuples.

In the source code, you are provided with matrix I/O functions and the code to load the sparse matrix in COO format. The sparse matrix A is loaded and stored into a one-dimension array $val[\,]$, which only contains non-zero element of the matrix A. The row and column index of every entry in $val[\,]$ are represented using two arrays: $rIndex[\,]$ and $cIndex[\,]$, for instance, the non-zero value $val[i]$ is located at row $rIndex[i]$ and column $cIndex[i]$ in the matrix A.

Ideally, the tuples (row, column, value) in the COO format are sorted by row index, then column index. However, in *mtx* files, it is not always the case. In the source code, we pre-process the loaded matrix entries and sort the $val[\,]$ values such that after pre-processing, the elements in the array $val[\,]$ are reordered by row index, and then by column index, both in ascending order.

Two 1-dimension arrays, $rsIndex$ and $reIndex$ represent the beginning and ending positions of each row of matrix A in the array $val[\,]$. For instance, the non-zero values of the i-th row in matrix A are stored in the $val[\,]$ array in locations

$$rsIndex(i), rsIndex(i) + 1, ..., reIndex(i)$$

An input vector is stored in a file in the format such that the first element in the file is the vector size, and the rest of the file contains the values of the vector sorted by row index in ascending order, illustrated below:

$$\begin{pmatrix} \text{Vector Size} \\ \text{b[0]} \\ \text{b[1]} \\ \ldots \\ \text{b[N]} \end{pmatrix}$$

In the source code, we also provide vector I/O functions to load and store the vector b values from a file. The 1-dimension array $vec[\,]$ stores the vector for sparse matrix vector multiplication. Please do not modify the matrix and vector I/O code!

[1] http://math.nist.gov/MatrixMarket/formats.html

2 Sequential SPMV Code

With the above variable definitions, the following sequential algorithm is used to calculate $A \times b$ where A is the sparse matrix and b is the input vector. The result is stored in $res[\,]$.

```
int i, j;
for (i = 0; i < M; i++) {
    res[i] = 0;
    for (j = rsIndex[i]; j <= reIndex[i]; j++) {
        int tmp = cIndex[j];
        res[i] += val[j]*vec[tmp];
    }
}
```

The sequential code has a two-level nested loop. The outermost i loop corresponds to the multiplication of a row vector in the matrix A with the column vector b. The innermost j loop corresponds to the accumulation of the multiplication results in a row and column vector multiplication. The non-zero entries in A are stored in $val[\,]$. The $rsIndex[i]$ and $reIndex[i]$ represents the starting and ending position of row i's non-zero elements in the array $val[\,]$. The code is in "spmv_seq.c".

To compile this program, you can type:

```
make sequential
```

To run this program, you type:

```
./spmv_seq [martix-market-filename] [input-vector-filename]
```

3 Parallelization

In this project, you are asked to only exploit loop-level parallelism in the given sequential *spmv* program. You will express loop-level parallelism through OpenMP pagmas, i.e., #pragma omp parallel variations. Detailed OpenMP pragma description can be found in https://computing.llnl.gov/tutorials/openMP/ .

You will need to choose the right loop level(s) to parallelize, and use the best loop scheduling parameters for each parallelized loop and learn how to improve the parallelization by feedback.

You will need to submit three OpenMP versions of the sparse matrix vector program, named spmv_N_thread_static.c, spmv_N_thread_dynamic.c, and spmv_N_thread_newalg.c.

The requirement for each is as follows:

1. spmv_N_thread_static.c: A version that uses static schedule for parallelization. You can exploit parallelism in only a single loop level. And you need to choose the right loop level based on dependence analysis. The "reduction" pragma is not allowed in this version.

2. spmv_N_thread_dynamic.c: A version that uses dynamic schedule for parallelization. Similarly, you can exploit parallelism in only a single loop level. You need to choose the right loop level based on dependence analysis. The "reduction" pragma is not allowed in this version.

3. spmv_N_thread_newalg.c: A version that uses your own strategy for parallelization. You can exploit parallelism in a single loop level or two loop level as long as the parallelized program produce correct output.

   Feel free to add preprocessing steps before this loop for optimization purpose, for instance, you can add code to determine the best chunk size for any given sparse matrix, or you can modify the loop for better load balancing and locality strategy. You can use other parallelization strategy such as "reduction" type parallelism. The goal is to make the parallel version of *spmv* as fast as possible.

In all three cases, the number of threads is set as a runtime parameter. That is, the parallel program will not only take an input matrix and vector as input, but also the number of threads. For instance, the dynamic schedule version takes three input parameters.

./spmv_parallel_dynamic [martix-filename] [vector-filename] [thread-num]

The thread number is stored in the "thread_num" variable in the program. If you need to use or apply the number of the threads in any other place of the program, please make sure it is consistent with the "thread_num" variable.

In OpenMP, in a static schedule, loop iterations are divided into pieces of size chunk and then statically assigned to threads, that is, the workload schedule is determined at compile time. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

In dynamic schedule, loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another from the work queue shared by all threads. If you do not specify the chunk size, it is 1 by default.

The static schedule reduces runtime overhead for accessing shared work-queue, however, it may not provide the best load balancing, since the number of iterations in the inner loop (j loop) is unknown at compile time, and may be different across different i iterations – the number of non-zero entries for each rows may be different.

The dynamic schedule provides better load balancing, since it determines at runtime, how many task (iterations) every thread needs to be assigned. A task is always assigned to the least idle thread during the execution. However, the dynamic schedule requires extra code generated at compile to communicate and update the shared work-queue.

In the third file you need to submit, the parallelization strategy you designed, you are encouraged to transform the loop and/or add additional code for better load balancing results. You are also encouraged to try other optimization strategies as mentioned above. The current source code includes timing functionality. You can check the running time for each version of the code.

Hints: to make your parallel program work efficiently, shared and thread-private variable have to be explicitly specified by using data-sharing attribute clauses in OpenMP.

We will grade your projects mainly based on correct parallelization parameters and correct program output (after parallelization). However, for better performance testing, we encourage you to run the program on a machine that has less interference from other users, for instance, your own computer. The ilab machines usually have multiple users active at the same time, and the parallel performance measured on a busy machine may be noisy. To design better parallelization strategy in the third version of your code, you might want to obtain accurate performance numbers. However, you still have to make sure your code runs on an ilab machines. In the end, we will use an idle ilab machine to test your program.

4 File Description

In this part we provide detailed description of the provided files, as well as the test cases. You only need to modify the following files when you do this project.

spmv_N_thread_static.c is provided for you to implement the static schedule parallel strategy. Its usage is
./pstatic [martix-market-filename] [input-vector-filename] [thread-num]

spmv_N_thread_dynamic.c is provided for you to implement the dynamic schedule parallel strategy. Its usage is
./pdynamic [martix-market-filename] [input-vector-filename] [thread-num]

spmv_N_thread_newalg.c is provided for you to implement your own parallel strategy. Its usage is
./pnewalg [martix-market-filename] [input-vector-filename] [thread-num]

The computation result, the output vector, will be saved in "output.txt" for grading purpose. We will grade based on both correctness and efficiency of your parallelized code.

Several other provided files include

mmio.h, mmio.c those two files are Matrix Market I/O library, which can be downloaded from http://math.nist.gov/MatrixMarket. We call the functions defined in this library to load the .mtx matrix file.

utils.h, utils.c those two files provide two utility functions."getmul" calculates the matrix-vector multiplication after loading the data. We compare its result with the one we get by parallel computing to check the result by "checkerror" function. The program will print "Calculation Error!" if the two results are not the same.

spmv_seq.c is the sequential implementation of matrix-vector multiplication. Its usage is
./spmv_seq [martix-market-filename] [input-vector-filename]

vector_generator.sh is used for generating the vector file. Its usage is
./vector_generator.sh [vector-size]
The generated file follows the format described in Section 1.

testcases folder stores the matrix and vector examples for testing. You are only given one test case: cop20k_A.mtx as the input matrix, cop20k_A_vector.txt as the input vector. More test matrices in mtx format can be found at http://math.nist.gov/MatrixMarket/

However, the input vector is not provided in the link above. Use the vector_generator.sh script above to generate an input vector of any size. The vector size has to match the number of columns in the input matrix. To find the dimension of the input matrix, in the mtx file, the first line specifies the number of rows, columns, and non-zero entries for the given matrix. The detailed format description is here: http://math.nist.gov/MatrixMarket/formats.html

.

Do not change the Makefile otherwise your code can not be compiled!