Report for CS427 project

Xuangui Huang

5100309715

Overview

In this project, my main responsibility is to implement the Min-Max Modular Network concurrently using OpenMP. My program will read results from my group partners, which consist of prediction results for different classes and different subproblems, then output results by M3 method computing parallel.

Details

Runtime Behavior

By invoking the m3 program without parameters, you can view its usage information:

```
$ ./m3
A M3 Program for SVMlinear and KNN
Usage: ./m3 <predict_output_file_folder> <output_file_folder> [A|B|C|D] <M=6> <
N=20> <threshold=0.85>
Reading prediction output files from <predict_output_file_folder>/[A|B|C|D]/{po
sitive numbers}-{negative numbers}.txt,
outputing results to <output_file_folder>/[A|B|C|D].txt.
$ ./m3_res
Getting M3 results
Usage: ./m3_res <M3_output_file_folder> <output_file_name> <test_file_name=test
.txt>
Reading M3 output files from <M3_output_file_folder>/[A|B|C|D].txt,
outputing results to file <output_file_name>
```

where M is the number of positive classes, N is the number of negative classes. This

program will read prediction output datum from sub-folder of `<predict_output_file_folder>` with respect to the class you choose, then output M3 result to file in `<output_file_folder>` with respect to that class.

Principal

The principal of Min-Max Modular Network is not difficult to understand: for each test data, just doing `Min` operations on prediction results obtained from the same positive class and different negative classes, then dong `Max` operation on these results for all the positive classes to obtain the final result.

Implementation

Basics

In `m3.cpp`, use `argc` and `argv` to parse command-line parameters.

```
  char* predict_output_file_folder = argv[1];
  char* output_file_folder = argv[2];
  int M = 6, N = 20;
  if (argc >= 5) M = atoi(argv[4]);
  if (argc >= 6) N = atoi(argv[5]);
  if (argc >= 7) threshold = atof(argv[6]);
  char sub_class = argv[3][0];
```

In `m3_result.cpp`, use the following program to combine results from different classes into a multi-class result and obtain prediction precision:

```
  for (i = 0; i < MAX_CLASS; ++i) {
    snprintf(M3_output_file[i], BUF_LEN, "%s/%c.txt", M3_output_file_folder, ch
ar('A' + i));
    M3_output_file_handle[i] = fopen(M3_output_file[i], "r");
  }

  line = (char*) malloc(sizeof(char) * max_line_len);
  int right_count = 0;
  for (l = 0; l < MAX_TEST; ++l) {
    double max = 0.0f, tmp;
    int max_i = 0;
    for (i = 0; i < MAX_CLASS; ++i) {
      fscanf(M3_output_file_handle[i], "%lf", &tmp);
      if (tmp > max) {
        max = tmp;
        max_i = i;
      }
    }
```

```
    fprintf(output_file_handle, "%c\n", char('A' + max_i));

    readline(test_file_handle);
    if (line[0] == char('A' + max_i)) right_count++;
  }

  printf("precision: %lf\n", (double) right_count / MAX_TEST);
```

Without Parallelism

Without parallelism, M3 is easy to implement using nested `for` loops. Simply loop on positive classes and negative classes, read prediction result files, then do `Min` and `Max` operation:

```
  for (i = 0; i < M; ++i) {
    for (j = 0; j < N; ++j) {
        \\ read files then do Min operation
    }
    \\ do Max operaion
  }
  \\ output
```

There are some details to make program neat and readable.

- To obtain a prediction file name and open it, `snprintf` and `fopen` are used:

```
snprintf(predict_output_file[i][j], BUF_LEN, "%s/%c/%d-%d.txt", predict_output_
file_folder, sub_class, i, j);
predict_output_file_handle[i][j] = fopen(predict_output_file[i][j], "r");
```

Note that we store them in arrays for different positive classes and negative classes, making them independent on each other, which is convenient to become parallel.

- To do `Min` and `Max` operation, `std::min` and `std::max` are used, and temporary results are stored in `_min` and `_max`, preventing them from polluting the namesapce:

```
\\ ...
    _min[l] = std::min(tmp[l], _min[l]);
\\ ...
    _max[l] = std::max(_min[l], _max[l]);
\\ ...
```

With Parallelism using OpenMP

By inserting `#pragma omp` clauses into the previous program, we can easily obtain a parallel M3 program using OpenMP:

```
#pragma omp parallel for num_threads(M) schedule(static) default(shared) privat
e(i, j, l, tmp)
  for (i = 0; i < M; ++i) {
#pragma omp parallel for num_threads(N) schedule(static) default(shared) privat
e(j, l, tmp)
    for (j = 0; j < N; ++j) {
        \\ read files then do Min operation
    }
    \\ do Max operation
}
\\ output
```

Notice that we will invoke M threads for the outer loop, each thread will then invoke N threads for the inner loop. To use this kind of nested parallel computation in OpenMP, we have to first enable it by the following subroutine call before this parallel region:

```
omp_set_nested(1);
```

Files are read sequently in each threads, however there may have data races when storing temporary `Min` and `Max` results. Therefore critical regions are created to protect them:

```
\\ ...
#pragma omp critical(crimin)
    {
      _min[i][l] = std::min(tmp, _min[i][l]);
    }
\\ ...
#pragma omp critical(crimax)
    {
      _max[l] = std::max(_min[i][l], _max[l]);
    }
\\ ...
```

However, this kind of protection have some impact on time performance. I've tried to use locks to lessen time dissipation on dispensable block, since not all the threads will have data races when accessing those areas concurrently. However, the total number of locks in OpenMP is very limited, hence I don't use it at last.

Why uses OpenMP?

In the original M3 paper, large cluster and MPI are used to implement the M3 network.

Data are distributed in different cluster nodes, subroutines for subproblems are also invoked in different cluster nodes, `Min` and `Max` operations are implemented using `MPI_Reduce()` method. However, in this project, data are not so large so that they can be stored in just one machine. Besides, we don't have a large cluster to use to exploit Task Level Parallelism. The underlying training and prediction algorithm we use, both kNN and SVMlinear, can't be invoked simultaneously for each subproblems, though they have been speed up by CUDA and OpenMP inside themselves. Hence we choose OpenMP to implement M3 network, instead of OpenMPI.

Acknowledgment

Thanks to our TA and my group partners: Lingxiao Jia, Libo Shen, Yaxiang Wang.