CS 395T **Scalable Machine Learning** **Fall 2014**

Solutions to Homework 2

Lecturer: Inderjit Dhillon Date Due: Sep 22, 2014

Keywords: *OpenMP, Netflix, Parallel ALS*

Problem 1

1. Run time in seconds, for matrix-matrix multiply, with gcc + OpenMP, on a Stampede computing node with 16 cores, 32 GB shared memory:
   (a) Parallelizing outer **for** loop: Add the following line just before outer loop.
   `pragma omp parallel for shared(A,B,C) private(i,j,k)`

Size	Sequential	1 thread	4 threads	8 threads	16 threads
50 x 50	676 x 10^{-6}	954.6 x 10^{-6}	323 x 10^{-6}	271.2 x 10^{-6}	211 x 10^{-4}
500 x 500	1.0105	1.1666	0.2981	0.1512	0.1669

   (b) Parallelizing inner **for** loop: Add the following line just before inner loop.
   `pragma omp parallel for shared(A,B,C,i) private(j,k)`

Size	Sequential	1 thread	4 threads	8 threads	16 threads
50 x 50	676 x 10^{-6}	954.7 x 10^{-6}	656.2 x 10^{-6}	961.6 x 10^{-6}	1.1142
500 x 500	1.0105	1.2363	0.3164	0.1593	11.5742

   (c) Parallelizing both the **for** loops: Add the following line just before outer loop.
   `pragma omp parallel for shared(A,B,C) private(i,j,k) schedule(static) collapse(2)`

Size	Sequential	1 thread	4 threads	8 threads	16 threads
50 x 50	676 x 10^{-6}	837 x 10^{-6}	316 x 10^{-6}	270 x 10^{-6}	217 x 10^{-5}
500 x 500	1.0105	1.1650	0.2952	0.1480	0.1647

2. Note that Matlab uses 16 cores by default on the Stampede computing nodes. For the 50x50 case, Matlab's multiply takes about 0.192 seconds (using 16 cores), and about one order of magnitude lesser (about 0.03 seconds) using a single core (which can be set by invoking `maxNumCompThreads(1)`). For the 500x500 case, Matlab's multiply takes about 0.02 seconds regardless of the number of cores.

Problem 2

(a) See the source code in the end of this file.

(b) `icc -O3 -fopenmp -I./eigen -o omp-als als.cpp`

(c)

#threads	1	2	4	8	16
Walltime(s)	258.6	165.1	75.2	36.5	20.5

(d) Matlab implementation took 74s to finish 10 iterations, which is about 3.7x longer than `omp-als`.

Problem 3

(a)

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + C\sum_{i=1}^{n}(\sigma(y_i\boldsymbol{w}^T\boldsymbol{x}_i) - 1)y_i\boldsymbol{x}_i,$$
$$\nabla^2 f(\boldsymbol{w}) = I + CX^TDX,$$

where $\sigma(y_i\boldsymbol{w}\boldsymbol{x}_i) = (1 + e^{-y_i\boldsymbol{w}^T\boldsymbol{x}_i})^{-1}$ and D is a diagonal matrix with $D_{ii} = \sigma(y_i\boldsymbol{w}^T\boldsymbol{x}_i)(1 - \sigma(y_i\boldsymbol{w}^T\boldsymbol{x}_i))$.

(b) $O(nd)$.

(c) $O(nd^2 + d^3)$.

Source Code for Parallel ALS using OpenMP and Eigen

```
#include <cstdio>
#include <cstdlib>
#include <vector>
#include <omp.h>
#include <Eigen/Sparse>
#include <Eigen/Eigen>

typedef Eigen::SparseMatrix<double> smat_t;
typedef Eigen::Triplet<double> triplet_t;
typedef Eigen::MatrixXd dmat_t;
typedef Eigen::VectorXd vec_t;

void exit_and_usage() { // {{{
    puts( "./omp-als rank lambda nr_threads data_dir\n");
    exit(-1);
} // }}}
void load_data(const char* data_dir, smat_t &R, smat_t &testR) { // {{{
    char buf[1024];
    char filename[1024];
    int rows, cols;
    size_t nnz;
    std::vector<triplet_t> triplet_list;
    sprintf(buf, "%s/meta", data_dir);
    FILE *meta = fopen(buf, "r");
    fscanf(meta, "%d %d", &rows, &cols);
    fscanf(meta, "%ld %s", &nnz, filename);
    sprintf(buf, "%s/%s", data_dir, filename);
    FILE *training = fopen(buf, "r");
    triplet_list.resize(nnz);
    R.resize(rows, cols);
    for(size_t s = 0; s < nnz; s++) {
        int row, col;
```

```
        double value;
        fscanf(training, "%d %d %lf", &row, &col, &value);
        triplet_list[s] = triplet_t(row-1, col-1, value);
    }
    R.setFromTriplets(triplet_list.begin(), triplet_list.end());
    fclose(training);
    fscanf(meta, "%ld %s", &nnz, filename);
    sprintf(buf, "%s/%s", data_dir, filename);
    FILE *test = fopen(buf, "r");
    triplet_list.resize(nnz);
    testR.resize(rows, cols);
    for(size_t s = 0; s < nnz; s++) {
        int row, col;
        double value;
        fscanf(training, "%d %d %lf", &row, &col, &value);
        triplet_list[s] = triplet_t(row-1, col-1, value);
    }
    testR.setFromTriplets(triplet_list.begin(), triplet_list.end());
    fclose(test);
    fclose(meta);
    puts("data load finished");
} // }}}
double cal_rmse(smat_t &testR, dmat_t& W, dmat_t &H) { // {{{
    int cols = testR.cols();
    double rmse = 0;
#pragma omp parallel for reduction(+:rmse) schedule(dynamic, 32)
    for(int c = 0; c < cols; c++) {
        for(smat_t::InnerIterator it(testR,c);it;++it) {
            int r = it.index();
            double err = W.col(r).transpose()*H.col(c)-it.value();
            rmse += err*err;
        }
    }
    return sqrt(rmse/testR.nonZeros());
} // }}}
void run_als(smat_t &R, smat_t &testR, int k, double lambda, int nr_threads, int maxiters=10) {//{{{
    int rows = R.rows();
    int cols = R.cols();
    omp_set_num_threads(nr_threads);
    dmat_t W = 0.5*(dmat_t::Random(k, rows)+dmat_t::Ones(k, rows));
    dmat_t H = 0.5*(dmat_t::Random(k, cols)+dmat_t::Ones(k, cols));
    std::vector<dmat_t> A_set(nr_threads);
    std::vector<vec_t> y_set(nr_threads);
    for(int i = 0; i < nr_threads; i++) {
        A_set[i].resize(k,k);
        y_set[i].resize(k);
    }
    dmat_t lambdaI = lambda*dmat_t::Identity(k,k);
```

```
    double total_time = 0;
    printf("start with RMSE %g nr_threads %d\n",
            cal_rmse(testR, W, H), nr_threads);
    for(int iter = 1; iter <= maxiters; iter++) {
        double starttime = omp_get_wtime();
#pragma omp parallel for shared(R,Rt,W,H,A_set,y_set) schedule(dynamic,32)
        for(int j = 0; j < cols; j++) {
            int thread_id = omp_get_thread_num();
            vec_t &y = y_set[thread_id];
            dmat_t &A = A_set[thread_id];
            for(int s = 0; s < k; s++) {
                for(int t = 0; t < k; t++) A(t,s) = 0;
                A(s,s) = lambda;
                y(s) = 0;
            }
            for(smat_t::InnerIterator it(R, j);it;++it) {
                int i = it.index();
                double val = it.value();
                for(int s = 0; s < k; s++) {
                    y(s) += W(s,i)*val;
                    for(int t = 0; t < k; t++)
                        A(t,s) += W(t,i)*W(s,i);
                }
            }
            H.col(j) = A.llt().solve(y);
        }
#pragma omp parallel for shared(R,Rt,W,H,A_set,y_set) schedule(dynamic,32)
        for(int i = 0; i < rows; i++) {
            int thread_id = omp_get_thread_num();
            vec_t &y = y_set[thread_id];
            dmat_t &A = A_set[thread_id];
            for(int s = 0; s < k; s++) {
                for(int t = 0; t < k; t++) A(t,s) = 0;
                A(s,s) = lambda;
                y(s) = 0;
            }
            for(smat_t::InnerIterator it(Rt, i);it;++it) {
                int j = it.index();
                double val = it.value();
                for(int s = 0; s < k; s++) {
                    y(s) += H(s,j)*val;
                    for(int t = 0; t < k; t++)
                        A(t,s) += H(t,j)*H(s,j);
                }
            }
            W.col(i) = A.llt().solve(y);
        }
        total_time += omp_get_wtime() - starttime;
```

```
        printf("iter %d walltime %lf rmse %lf |W| %lf |H| %lf\n",
                iter, total_time, cal_rmse(testR, W, H), W.norm(), H.norm());
    }
}// }}}
int main(int argc, char* argv[]){ // {{{
    if(argc != 5) exit_and_usage();
    int k = atoi(argv[1]);
    double lambda = atof(argv[2]);
    int nr_threads = atoi(argv[3]);
    char *data_dir = argv[4];
    int max_iters = 10;
    smat_t R, testR;
    load_data(data_dir, R, testR);
    run_als(R,testR,k,lambda,nr_threads,max_iters);
    return 0;
} // }}}
```