Babai Point Asynchronous Implementation Report

Shilei Lin
ID: 260887631
School of Computer Science
McGill University
shilei.lin@mail.mcgill.ca

October 05, 2020

1 Data Representations

The algorithm of finding the Babai point in my implementation is written in C++. The matrix and vector are initialized by the *Eigen*[1] Library. The reason I chose *Eigen* is that it is a widely used C++ library for doing some numerical computations. Especially, for our purposes, I use *Eigen* to randomly generate an n by n matrix A, then use *Eigen::HouseholderQR* to perform the QR decomposition on A, to obtain the upper triangular matrix R.

2 Initial point

The initial point x_0 and RHS vector y are generated by the following code in the class constructor

```
    std::random_device rd;
    std::mt19937 gen(rd());  //here you could also set a seed
    std::uniform_real_distribution<double> dis(-n, n);

    this->x0 = Eigen::VectorXd::Zero(n).unaryExpr([&](int dummy) { return round(dis(gen)); });
    this->y = R * x0 + 10 * VectorXd::Random(n); //VectorXd::Random(n) can generate a random nx1
    vector in range [-1,1].

```

3 Computer Specification

My own desktop's CPU is an Intel Core i7-8700k, which is an 6 cores/12 threads CPU. When starting the OpenMp parallel pool, I created 12 threads:

```
    int Max_threads = omp_get_max_threads();
    std::cout<<"Max_threads="<< MAX_JOB <<std::endl;

    int n = 1000, nswp = 6, n_jobs = 12;

```

Then, inside the solver function, before the for-loops, the parallel region starts like:

```
    #pragma omp parallel num_threads(n_proc) private(sum) shared(raw_x)

```

In the future, I can test the code on my GPU, a GTX Nvidia 1080, which has 2560 Cuda cores.

4 The residual of the Babai point and the running time

1. For matrix size as 1000 by 1000:
   The residual for regular Babai point is **120.38121**, and the running time is **0.075084** seconds. Then, for the parallel Babai point, the results are as following:

[1] `http://eigen.tuxfamily.org/index.php?title=Main_Page`

(a) For 0 ”sweeps”, the residual is 321.83642, and the running time is 0.000000 seconds
(b) For 1 ”sweeps”, the residual is 5277.48836, and the running time is 0.004534 seconds
(c) For 2 ”sweeps”, the residual is 1451.05695, and the running time is 0.008898 seconds
(d) For 3 ”sweeps”, the residual is 423.29749, and the running time is 0.013278 seconds
(e) For 4 ”sweeps”, the residual is 127.02070, and the running time is 0.017914 seconds
(f) For 5 ”sweeps”, the residual is 121.15369, and the running time is 0.023582 seconds
(g) For 6 ”sweeps”, the residual is 120.25204, and the running time is 0.026964 seconds
(h) For 7 ”sweeps”, the residual is 120.41889, and the running time is 0.031064 seconds
(i) For 8 ”sweeps”, the residual is 120.38121, and the running time is 0.035442 seconds
(j) For 9 ”sweeps”, the residual is 120.38121, and the running time is 0.041011 seconds
(k) For 10 ”sweeps”, the residual is 120.38121, and the running time is 0.044826 seconds
(l) For 11 ”sweeps”, the residual is 120.38121, and the running time is 0.049399 seconds
(m) For 12 ”sweeps”, the residual is 120.38121, and the running time is 0.053094 seconds

Clearly, after 8 sweeps, the result converged to the regular Babai point, and the running time is also faster than the regular version. Also, noticeably, that at the sixth sweep, the residual is even smaller than the regular version.(But I may need to investigate why this happened in the future).

2. For matrix size as 2000 by 2000: the residual for regular Babai point is **234.85885**, and the running time is **0.294** seconds. Then, for the parallel Babai point, the results are as following:

(a) For 0 ”sweeps”, the residual is 261.96518, and the running time is 0.000000 seconds
(b) For 1 ”sweeps”, the residual is 1986.01265, and the running time is 0.017821 seconds
(c) For 2 ”sweeps”, the residual is 765.52382, and the running time is 0.035294 seconds
(d) For 3 ”sweeps”, the residual is 270.05316, and the running time is 0.053627 seconds
(e) For 4 ”sweeps”, the residual is 246.01876, and the running time is 0.070473 seconds
(f) For 5 ”sweeps”, the residual is 241.38594, and the running time is 0.088520 seconds
(g) For 6 ”sweeps”, the residual is 234.90108, and the running time is 0.106270 seconds
(h) For 7 ”sweeps”, the residual is 235.72692, and the running time is 0.124928 seconds
(i) For 8 ”sweeps”, the residual is 234.85885, and the running time is 0.141407 seconds
(j) For 9 ”sweeps”, the residual is 234.85885, and the running time is 0.158683 seconds
(k) For 10 ”sweeps”, the residual is 234.85885, and the running time is 0.175248 seconds
(l) For 11 ”sweeps”, the residual is 234.85885, and the running time is 0.196180 seconds
(m) For 12 ”sweeps”, the residual is 234.85885, and the running time is 0.211432 seconds

Clearly, after 8 sweeps, the result converged to the regular Babai point, and the running time is also faster than the regular version.

5 Comments on the results

There are some remarks that I want to make.

The R upper triangular matrix in the parallel version is compressed and not stored as *Eigen* datatype. Since the half of the R matrix is filled with 0s, the actually number of entries that need to be stored is $n(n-1)/2$. Then I put all nonzero entries into a C++ standard $std :: vector < double >$, which actually speed up the program a lot.

Also, the visual studio 2019 only supports OpenMp 2.0 but the latest version is 4.0. So if the result is sound, then I can test the code in Linux with OpenMp 4.0 which has some new parallel functions built in. Furthermore, as I have mentioned in section 3, I have not yet run the code with my GPU.