Sparse Matrix Solver

Tian Yu (1004750655)

May 15, 2021

Contents

1 Source Code

All source code could be found on my Github Repo

2 Validation

I implemented the validation function the reverse way of the lsolve function. Given the original matrix L and output vector x, I multiplied them and compare the result with the original b vector.

However, this type of approach is not working for matrices with large dimensions, like $1k \times 1k$. This is because the floating point operation cannot resist accuracy and even type double cannot store numbers that are too large. This is further explained in section 5. Thus, I also implemented two python checker, autotest.py and autotest_omp.py to varify the logic of my code. I tested the correctness for 1000 iterations on random generated matrices of size 20×20 for both serial and multi-threaded code.

3 Serial Optimization

3.1 Implementations

```
1  int lsolve_improve_1(int n, int *Lp, int *Li, double *Lx, double *x)
2  {
3      int p, j;
4      if (!Lp || !Li || !x)
5          return (0);
6      /* check inputs */
7      for (j = 0; j < n; j++)
8      {
9          //check if x[j] is 0 to save time
10         if (x[j] == 0)
11         {
12             continue;
13         }
14         x[j] /= Lx[Lp[j]];
15         for (p = Lp[j] + 1; p < Lp[j + 1]; p++)
16         {
17             x[Li[p]] -= Lx[p] * x[j];
18         }
19     }
20     return (1);
21 }
```

Compared to the starter code, I find that the inner for loop will be useless if x[j] on line 14 is 0. That's why I added a checker from line 10-13 to save time. It turns out this small change results in the best performance improvement. In terms of the run time analysis, the run time of the algorithm is between $\Omega(n)$ and $\mathcal{O}(nz)$ where n is the number of rows in the original matrix and nz is the number of non-zeros in the matrix.

In order to better improve the run time, I also implemented the **lsolve_DFS_traversal** function (in main.c). Instead of looping over all the elements in the vector b, I used the DFS to search for all the "involved" element in b and only update those rows like line 15-18 in the code above starting from the original non-zero elements in vector b. (By "involve" I mean the elements in the original x vector that contributes to the result of non-zero elements in b.) This is also mentioned in the video at 36min. Here, the run time is $\mathcal{O}(Flop)$ instead. Where the "Flop" means the number of calculation required to get the final answer of x.

3.2 Runtime Comparison

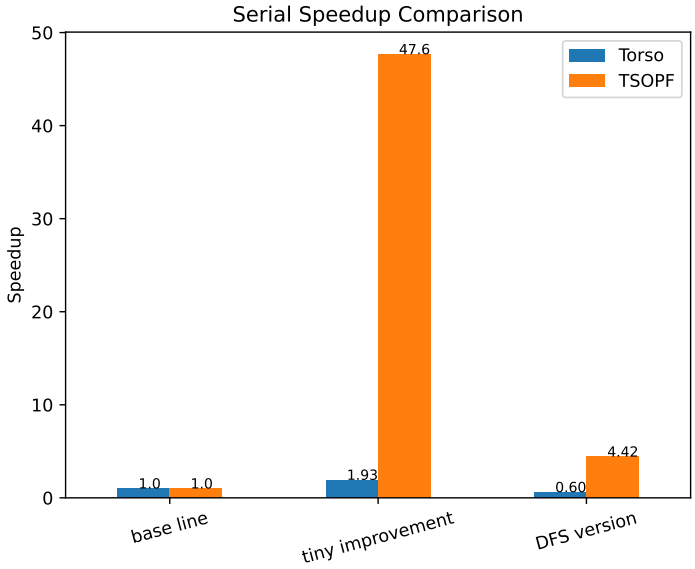

Figure 1: Serial Speedup Comparison

As shown in the graph, I find the "trivial improvement" achieves the greatest speedup in both datasets. I think this is because the "DFS" search implementation still requires more much more IO operations compared to the naive implementation.

On the other hand, both optimized algorithms experience a worse speedup in the Torso dataset. I think this is because Torso matrix is more "friendly" to the non-optimized algorithm. I used cachegrind to study the cache miss of functions on both matrices, which is shown below:

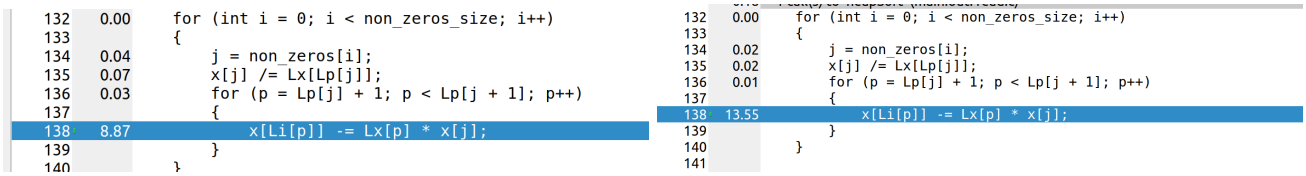

Figure 2: Cache miss, Left: Torso, Right: TSOPF

I find the "relative" total cache miss of the algorithm on Torso is almost half of that on TSOPF. (I choose relative cache miss because although the two matrices has different size, the proportion of cache miss for reading and on line 138 should remain the same "relatively" since they are all proportional to the number of non-zero elements in the matrix.) Thus even though the non-optimized code requires more IO operations as the inner loop cannot be avoided, the run time for the non-optimized code is not that bad thanks to the caching.

4 Parallel Optimization

4.1 Implementation

I also implemented two paralleled versions of the above algorithms. I parallelized the "tiny improved" code by doing multi-threading on the inner loop (line 15-18 in the page1's code).

On the other hand, the paralleled version of the second optimization involves two phases. The analysis phase and the numeric phase. The analysis phase is used to generate a dependency graph to inform the numeric phase which level of nodes could be calculated in parallel. And the numeric phase is the time for parallel computing the original vector

x. I got this idea from the **Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU**[1] paper. However, different from the idea of analysis phase in the paper, I decide not to generate the dependency graph of the entire matrix, but rather only partial of the graph related to the non-zero elements in the given vector b. Due to the limitation in the development time, my current correct implementation requires 1 iteration of customized DFS search to find the number of parent of each related node, and might also need another iteration to solve the problem of some elements being the child of others in the vector b and another iteration with BFS to generate the dependency graph by levels stored in a 1-D array with another array as it's level pointers. I think my implementation should work better in case of the given vector b. Because vector b is sparse, there is no need to iterate over the entire matrix.

4.2 Runtime Comparison

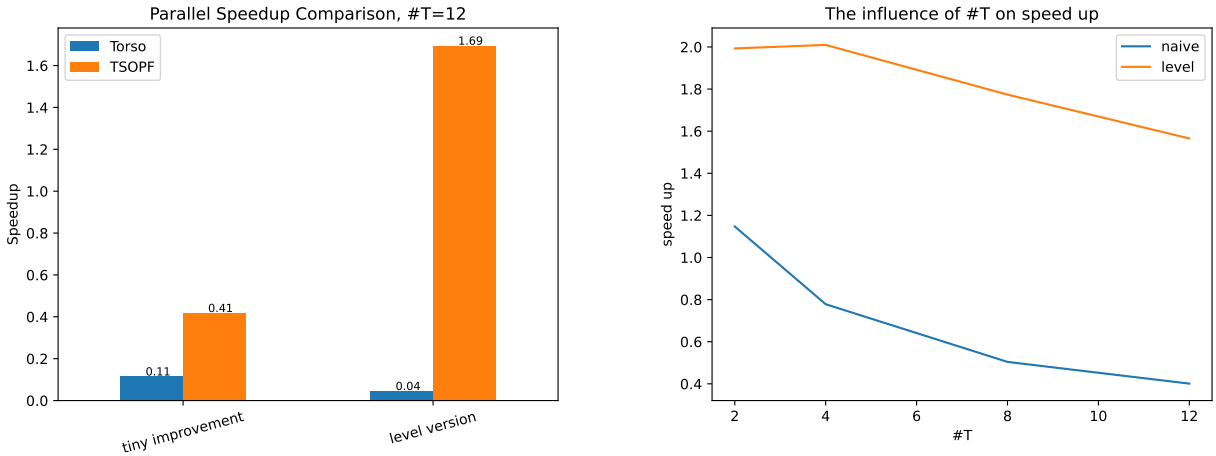

Figure 3: Parallel algorithm comparison

As shown on the left plot, I find the similar pattern as discussed in section 3.2, where the performance of improved algorithms on Torso matrix is worse than the one on TSOPF matrix. Besides, I also find that running the algorithms on multi-cores will not dramatically improve the speed up. In the case of naive approach, multi-threading may even decrease the performance. I think there are two causes The first reason is the need for "global synchronization". This is more a problem in the level version of the code, where the vector x should be synched on each update at line 9 between processes:

```
1  #pragma omp parallel default(shared)
2  #pragma omp for
3          for (int child = 0; child < cur_size; child++)
4          {
5              int j = arr[child];
6              //no need to add omp critical here as elements at the same level are independent
7              x[j] /= Lx[Lp[j]];
8              for (int p = Lp[j] + 1; p < Lp[j + 1]; p++)
9              {
10                 #pragma omp critical
11                 x[Li[p]] -= Lx[p] * x[j];
12             }
13         }
14         index = cur_upper;
15     }
```

Another reason is the "false sharing" and the cost of thread creation. This is the main problem causing the decrease in performance for the naive implementation. In terms of thread creation and destroy, each time, the inner loop will iterate on average 7 times. Thus the cost of thread create and destroy is even higher than the benefit of computation.

Besides, having multiple threads accessing the vector x continuously may result in false sharing. This is why the speed up curve for the tiny improvement decreases as the number of thread increases:

```
#pragma omp parallel default(shared) private(p)
#pragma omp for
        for (p = Lp[j] + 1; p < Lp[j + 1]; p++)
        {
            x[Li[p]] -= Lx[p] * x[j];
        }
```

5 Problems Improvements

The current algorithm still have the problem of getting "wrong" result when the size of matrix is pretty large.

One reason is the numerical errors will be significant if the number of floating point operation required increases. Below (left) is the comparison about the output of the numpy solve (labeled x) function and the starter code lsolve function (labeled xcout) on a randomly generated 100×100 matrix. You can see there is a slight difference in the last digit. However, this could result in an error summing up to thousands if I continue to use the original validation method.

Another reason is that double cannot store value that is too large. However, when solving the linear system of torso matrix for example, I can partition the elements required for calculation into more than 60 levels. This means the calculation will be accumulated and easily exceed the limitation that double could store. This result in inf initially and then result in nan on the further calculation. This is also shown on the right plot.

```
tian@tian:~/uoft/sparse_matrix_report$ python3 ./autotest.py
x: -6.693932204250502e+16 x_cout: -6.693932204250503e+16
x: 2.4953828846536666e+18 x_cout: 2.4953828846536663e+18
x: -2.1012308142589203e+17 x_cout: -2.10123081425892e+17
x: 1.594694598300504e+18 x_cout: 1.5946945983005046e+18
x: -1.2242054697879285e+17 x_cout: -1.2242054697879286e+17
```

```
x: -4.928688158274277e+304 x_cout: -4.928688158274266e+304
x: nan x_cout: -inf
x: nan x_cout: nan
x: nan x_cout: nan
x: nan x_cout: -inf
```

Figure 4: Cache miss, Left: floating point error, Right: number too large

References

[1] Maxim Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, 1, 2011.