A Report on Optimizing the Performance of the Sparse Triangular Solve

by Tara Saba

Table of Contents

Abstract

Optimization of sparse matrix kernels is one of the most important topics in today's world of computer science. In this report we report on the implementation of four sparse triangular solve algorithms. A naïve approach and its optimization are implemented as single core algorithms alongside parallel level-scheduling and self-scheduling approaches with two verification strategies. The project is implemented in C++ programming language and the performances of the algorithms on two matrices are reported. With simple optimizations being introduced, the single-core optimized and parallel algorithms performed significantly faster than the naïve approach.

Keywords: Sparse matrix kernels, Sparse triangular solve , Performance optimization

1. Implementation Details

In this section we will discuss the algorithms implemented to perform Sparse Triangular Solve. A total of four algorithms is implemented and discussed. After that, will discuss the two verification strategies used to verify the result of the algorithms.

1-1 Implemented Algorithms

Four algorithms were implemented in this project. The first two algorithms described in this section, namely the Naïve Single-threaded and the Optimized single-threaded Sparse Triangular Solve are single core implementations of the kernel and the last two algorithms, namely Parallel Level-scheduling and Parallel Self-scheduling are multicore algorithms for sparse triangular solve. All of the algorithms are implemented using the CSC format of the given matrix.

1-1-1 Naïve Single-threaded Sparse Triangular Solver

In this project the give naïve version of the sparse triangular solve approach is implemented. This algorithm (depicted in Figure1) is a good reference to evaluate the performance of other approaches.

1: **for** $i = 1, 2, \ldots, n$ **do**
2: $\quad x(i) := x(i)/d(i)$
3: $\quad$ **for** $j = ib(i), \ldots, ib(i+1) - 1$ **do**
4: $\quad\quad x(jb(j)) := x(jb(j)) - b(j) \times x(i)$
5: $\quad$ **end for**
6: **end for**

Figure 1 The Naive Single-threaded Triangular Solve [1]

1-1-2 The Optimized single-threaded Sparse Triangular Solver

When dealing with sparse right-hand side vectors, not all columns of the input matrix take part in the naïve algorithm [2]. Thus, by determining the columns that take part in the computations and only looping on them, the naive algorithm can be optimized with the cost of adding an extra preprocessing stage to find all of the relevant columns to the algorithm.

Finding the relevant columns requires following two implications:

1) The i^{th} index of the answer is zero if the i^{th} index of the right-hand side vector is zero.
2) If the j^{th} index of the so far computed result vector is not zero and there exists a nonzero element in the i^{th} row and j^{th} column of the input matrix then the i^{th} column of the result is nonzero.

To follow these two rules to optimize the algorithm, we model the problem to a graph traversal problem. A directed dependency graph for the input matrix is implemented in which vertices correspond to row/column numbers and the edge from vertex i to vertex j represents a nonzero element in the j^{th} row and i^{th} column of the matrix. The first implication marks all vertices corresponding to nonzero row numbers of the right-hand side vector and the second implication marks all vertices that are a neighbor to an already marked vertex [2]. In this project, we visit vertices using the iterative version of the BFS algorithm and find the relevant columns and then modify the naïve approach to iterate only on those columns to further improve the algorithm. The recursive version of DFS was first implemented but was much slower than the iterative BFS algorithm.

1-1-3 The Parallel Level-scheduling Sparse Triangular Solver

In this algorithm we group the relevant columns into levels where each level contains columns that the computations can be done in parallel for them since the computations corresponding to these columns do not depend on each other. Furthermore, the calculations for columns in lower levels must be completed before the calculations of higher-level columns. This is because the results of higher-level columns depend on the lower-level columns.

To implement this algorithm, we leverage the previously introduced dependency graph to find the dependencies among the columns. Column j depends on i if

there exists a path of edges from the vertex i to the vertex j. We place each column in a higher level than its dependencies. To implement this, we looped over relevant columns and updated the levels of the vertices that are connected to them via outgoing edges. Finally, the algorithm in Figure2 is implemented in which we loop over levels and do the calculations in each level, in parallel.

```
1: for m = 1, ..., nlev do
2:   for k = ilev(m), ..., ilev(m+1) − 1 do {in parallel}
3:     i = jlev(k)
4:     for j = ia(i), ..., ia(i+1) − 1 do
5:       x(i) := x(i) − a(j) × x(ja(j))
6:     end for
7:     x(i) := x(i)/d(i)
8:   end for
9: end for
```

Figure 2 The Parallel Level-scheduling Sparse Triangular Solve [1]

1-1-4 The Parallel Self-scheduling Sparse Triangular Solver

To mitigate a draw-back of the Parallel Level-scheduling algorithm, the Parallel Self-scheduling Sparse Triangular Solver is implemented. In the Parallel Level-scheduling approach, computations for all of the columns in all of the lower levels must be finished to start the computations for a column in a higher level. However, not all of the columns present in a lower level are dependencies for a higher-level column. Thus, in the Parallel Self-scheduling algorithm, we find all of the columns that are dependencies to each column and only wait for them to finish before starting the computations for that specific column. In this case, we can also parallelize the outer loop in the naïve approach but need to make a thread wait for the threads computing the dependencies to finish. This algorithm is depicted in Figure3.

```
1: for i = 1, 2, ..., n do {columns in parallel}
2:   while count(i) ≠ 0 do
3:     {do nothing}
4:   end while
5:   x(i) := x(i)/d(i)
6:   for j = ib(i), ..., ib(i+1) − 1 do
7:     CRITICAL SECTION ENTRY
8:     x(jb(j)) = x(jb(j)) − b(j) × x(i)
9:     count(jb(j)) := count(jb(j)) − 1
10:    CRITICAL SECTION EXIT
11:  end for
12: end for
```

Figure 3 The Parallel Self-scheduling Sparse Triangular Solve [1]

1-2 Verification Strategies

One main strategy and one auxiliary verification strategy is implemented in this project. The first strategy uses matrix multiplication to verify the result of the algorithms. The second strategy simply compares the result of the naïve approach to the optimized approaches.

1-2-1 Verification using matrix multiplication

This is the main strategy used to verify the algorithm. We implemented a naïve implementation of the matrix by vector multiplication and leveraged it to verify the results. The input matrix was multiplied by the result of each algorithm and the result was compared with the right-hand side vector. All of the implemented algorithms were verified by this approach and thus work correctly (They were also confidently verified using lower dimension sparse matrices). However, this strategy might not provide accurate results with large matrices with small elements since precision overflow might occur and affect the results (although this was anticipated and addressed to some degree in the code). For this reason, we also report the percentage of the indices of the result of multiplication that matches the right-hand side vertex to show that the algorithm is correct and accurate. In addition, we also check if the elements of the result of the multiplication corresponding to the nonzero elements of the right-hand side vector were the same.

1-2-2 Verification using the naïve approach

This approach is implemented simply because we are comparing the performances of the optimized algorithms to the naïve approach and it would have been informative to see if they produce the same results.

All of the implemented algorithms were completely verified by this approach as well (using smaller sparse matrices) however, there might be some slight differences due to precision overflow in this verification method as well.

1-3 Conclusion

In this section, we discussed the algorithms implemented in this project to optimize the sparse triangular solve kernel. The naïve and the optimization to the naïve approach were single core algorithms whereas the parallel level-scheduling and self-scheduling algorithms take advantage of parallelism. In the next section we discuss system specifications and the libraries used in the project before moving on to the results in the third section.

2. Implementation environment and dependencies

This project is implemented in C++ programming language. In this section, we discussed the most important libraries used to implement the project and the systems specifications.

2-1 Utilized C++ Libraries

To implement this project, we took advantage of two C++ libraries' capabilities namely the Boost Graph Library and OpenMP. As mentioned in the previous section, we needed to solve graph traverse problems in order to implement, all of the optimized versions of the naïve algorithm. For this reason, we used the Boost Graph Library to be able to implement the code easier and quicker.

We also used the OpenMP framework and its corresponding C++ library to implement parallelization. Since the last two algorithms introduced in Section 1 required the for loops to run in parallel, OpenMP was used to provide parallelization capabilities.

2-2 Systems specifications

We used a simple commercial laptop to implement and run the project with its specifications depicted in Figure 4. In addition, the system run MacOS Big Sur as an operating system.

Processor Name:	6-Core Intel Core i7
Processor Speed:	2.6 GHz
Number of Processors:	1
Total Number of Cores:	6
L2 Cache (per Core):	256 KB
L3 Cache:	9 MB
Hyper-Threading Technology:	Enabled
Memory:	16 GB

Figure 4 System Specifications

2-3 Conclusion

In this section we discussed the run environment of the project. We discussed two important libraries of the project and briefly mentioned the specifications of the system. In the next section, we will discuss the results of each algorithm.

3. Results and performance

In this section, we report on the performances of the optimization algorithms on the TSOP_RS_b678_c2 and Torso matrices. We discuss two types of running times namely the preprocessing time and the solve time.

3-1 Preprocessing time

The preprocessing time is the time the algorithm takes to provide information on columns to the original algorithm and making it optimized as a result. In the optimized version of the single-core algorithm, the time to find the relevant columns is considered the preprocessing time. Similarly, for the parallel level-scheduling and self-scheduling, this time refers to the time taken to form the levels and the dependencies among columns respectively.

Figure5 provides information on the preprocessing stage time of each algorithm on the two matrices. As we can see, the optimized single-threaded algorithm has the least preprocessing time since preprocessing steps in both parallel algorithms perform the single-threaded preprocessing (finding the relevant column) as well. The parallel self-scheduling algorithm has a higher preprocessing time than the level-scheduling algorithm because it needs to find all the dependencies among vertices. The preprocessing time of the parallel self-scheduling for the Torso1 matrix was in order of hours and much higher than the TSOP_RS_b678_c2 preprocessing time (probably because of its sparsity pattern) thus not depicted in the figure.

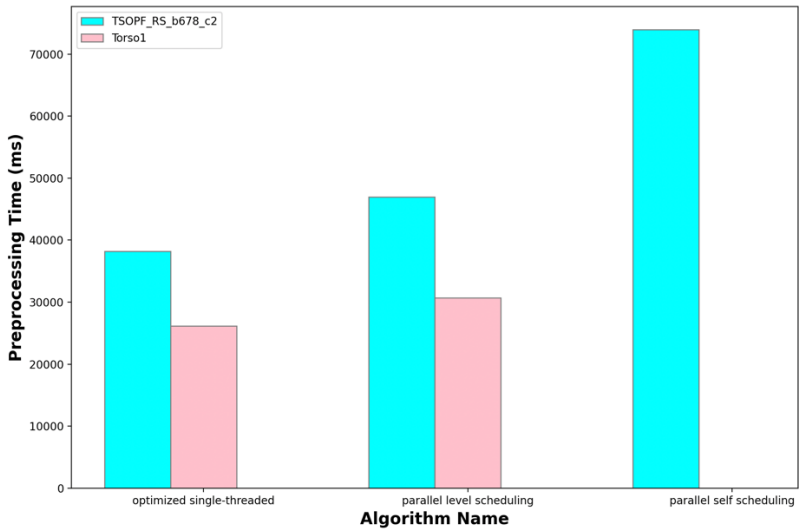


Figure 5 Preprocessing Time of Algorithms on the TSOPF and Torso1 Matrices

3-2 Solve time

The solve time is the time the algorithm takes to get the results of the triangular system after the preprocessing is over. The solve time of the parallel self-scheduling algorithm could not be provided because this algorithm is extremely resource intensive since it holds threads until threads computing on dependency columns are finished. So, it only improves the performance significantly if it has access to enough cores to be able to parallelize a large number of columns and release other threads. When I ran it on my system it was extremely slow because the system only has 6 cores.

As it can be viewed in figure6, the optimized single-core and parallel algorithms speeded up the operation significantly compared to the naïve approach on both matrices. However, the speedup was much greater with the TSOPF_RS_b678_c2 matrix. In addition, the parallel level scheduling algorithm showed slightly better performance than the optimized single-threaded algorithm.

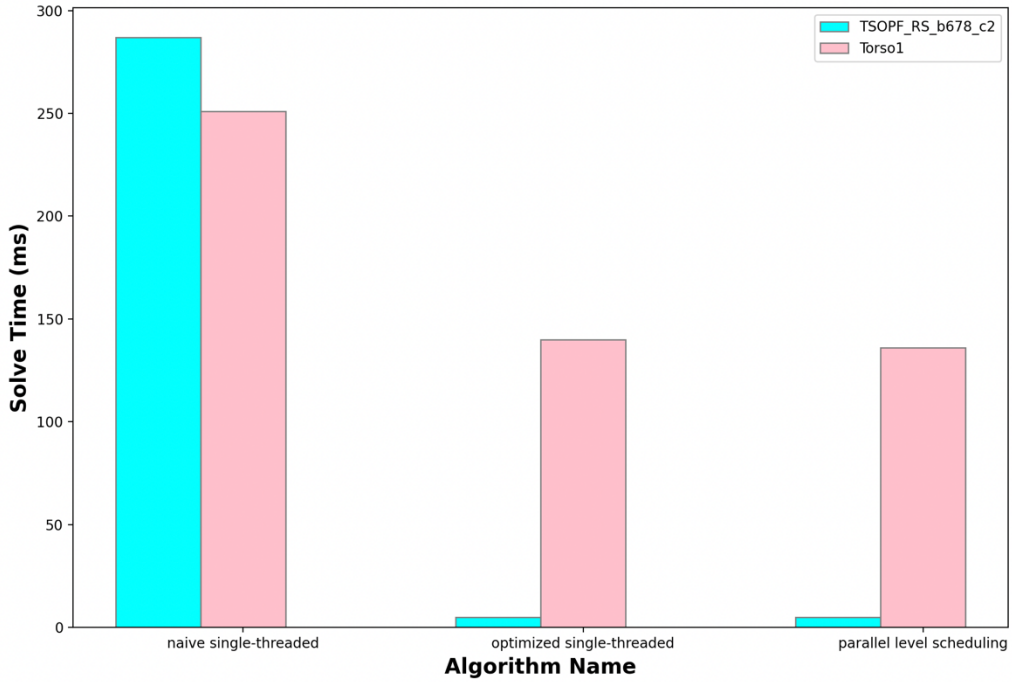
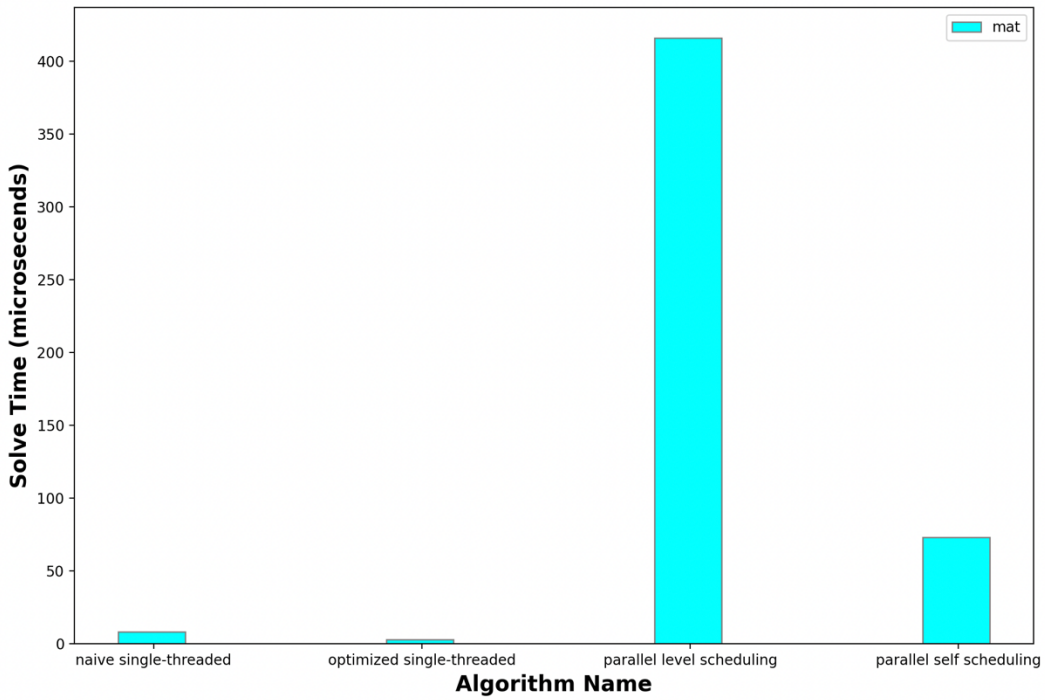


Figure 6 Solve Time of Algorithms on the TSOPF and Torso1 Matrices

To give a sense of the performance of the parallel self-scheduling algorithm, we ran the algorithm on a relatively small sparse matrix. In this matrix the overhead of parallelization was high with respect to the dimension and the characteristics of the matrix so the parallel algorithms ran slower than the single-threaded algorithms. The optimized single-threaded approach performed better than the naïve approach. Most importantly, the parallel self-scheduling algorithm performed much faster than the parallel level-scheduling algorithm.



References

[1] Li R. On parallel solution of sparse triangular linear systems in CUDA. arXiv preprint arXiv:1710.04985. 2017 Oct 13.

[2] Davis TA, Rajamanickam S, Sid-Lakhdar WM. A survey of direct methods for sparse linear systems. Acta Numerica. 2016 May; 25:383-566.