Sparse Matrix Solver Optimization

In the code optimization, the running time of **four sparse solver** implementations were compared.

#1:
The first solver consists of the naive implementation and the algorithm can be seen below:

```
int solve_sparse_naive(int n, const vector <double> &L_val, const vector <int> &L_rows, const vector
<int>  &L_col_p,
    vector <double> &x){
    cout << "This is a naive implementation." << endl;
    if (!L_rows.size() || !L_col_p.size() || !x.size())
        return 0;

    for (int j = 0 ; j < n ; j++){
        x[j] /= L_val[L_col_p[j]];

        for (int p = L_col_p[j]+1 ; p < L_col_p[j+1] ; p++)
            x[L_rows[p]] -= L_val[p] * x[j];
    }
    return 1;
}
```

#2:
The second optimization is derived using the following process:

Let us assume we have a generic matrix as our input and a generic vector as our output to see how the operations work. We will create a Lower Triangular matrix and assume all values are non-zero in the diagonal and on the diagonal.

$$A = \begin{Bmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ a_{20} & a_{21} & a_{22} & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} \end{Bmatrix} \quad x = \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{Bmatrix}$$

In the first iteration $j = 0$ and the following updates are done to the first column.

1. The diagonal value in the column is used to update the corresponding value in the same row in x,

$$x_0 = \frac{x_0}{a_{00}}$$

2. We iterate for all the non-zero values below the diagonal value and update the corresponding value in the same row in the x vector by taking the product of the value in

the column and multiplying it by the value in the x vector corresponding to the same column.

$$x_1 = x_1 - a_{10} * x_0$$
$$x_2 = x_2 - a_{20} * x_0$$
$$x_3 = x_3 - a_{30} * x_0$$

More generally we can see that when we iterate for all the non-zero in step 2 underneath the diagonal value all the updates are proportional to a fixed value. We can write this rule more generally as.

$$x_p = x_p - a_{p,j} * x_j$$

We can see that in the case of $x_j = 0$ we get the following

$$x_p = x_p - a_{p,j} * 0$$
$$x_p = x_p$$

The corresponding x_p values don't change, we can take advantage of this by skipping the entire for-loop if our x_j value is zero by adding a condition before the for loop to skip the for loop if the value of x_j is zero.

```
int solve_sparse_optimized(int n, const vector <double> &L_val, const vector <int> &L_rows, const
vector <int>  &L_col_p,
    vector <double> &x){
    cout << "This is an optimized implementation." << endl;
    if (!L_rows.size() || !L_col_p.size() || !x.size())
        return 0;

    for (int j = 0 ; j < n ; j++){
        if (x[j] == 0) {
            continue;
        }

        x[j] /= L_val[L_col_p[j]];

        for (int p = L_col_p[j]+1 ; p < L_col_p[j+1] ; p++)
            x[L_rows[p]] -= L_val[p] * x[j];
    }
    return 1;
}
```

#3:

The third solver implements the OpenMP library to enable parallelization.

```
int solve_sparse_parallel(int n, const vector <double> &L_val, const vector <int> &L_rows, const vector
<int>  &L_col_p,
    vector <double> &x){
    cout << "This is a parallelized implementation." << endl;
    if (!L_rows.size() || !L_col_p.size() || !x.size())
        return 0;
    #pragma omp parallel
    for (int j = 0 ; j < n ; j++){
        x[j] /= L_val[L_col_p[j]];

        #pragma omp parallel
        for (int p = L_col_p[j]+1 ; p < L_col_p[j+1] ; p++)
            x[L_rows[p]] -= L_val[p] * x[j];
    }

    return 1;
}
```

#4:

The last solver implements technique #2 and #3 and has parallelization and optimization enabled.

```
int solve_sparse_parallel_optimized(int n, const vector <double> &L_val, const vector <int> &L_rows,
const vector <int>  &L_col_p,
    vector <double> &x){
    cout << "This is a parallelized optimized implementation." << endl;
    if (!L_rows.size() || !L_col_p.size() || !x.size())
        return 0;
    #pragma omp parallel
    for (int j = 0 ; j < n ; j++){
        if (x[j] == 0) {
            continue;
        }
        x[j] /= L_val[L_col_p[j]];

        #pragma omp parallel
        for (int p = L_col_p[j]+1 ; p < L_col_p[j+1] ; p++)
            x[L_rows[p]] -= L_val[p] * x[j];
    }

    return 1;
}
```

The run time of each algorithm is recorded in the table below (Tested on Torso1.mtx):

Note: **Time for reading the file not included.**

<table>
<tr><th>Index</th><th>Technique</th><th>Results</th></tr>
<tr><td>1</td><td>Naive</td><td>
<table>
<tr><th>Trial</th><th>Time</th></tr>
<tr><td>1</td><td>0.1048430s</td></tr>
<tr><td>2</td><td>0.1106190s</td></tr>
<tr><td>3</td><td>0.1058980s</td></tr>
</table>
</td></tr>
<tr><td>2</td><td>Optimised</td><td>
<table>
<tr><th>Trial</th><th>Time</th></tr>
<tr><td>1</td><td>0.0639110s</td></tr>
<tr><td>2</td><td>0.0635580s</td></tr>
<tr><td>3</td><td>0.0622980s</td></tr>
</table>
</td></tr>
<tr><td>3</td><td>Parallized</td><td>
<table>
<tr><th>Trial</th><th>Time</th></tr>
<tr><td>1</td><td>0.1050860s</td></tr>
<tr><td>2</td><td>0.1090850s</td></tr>
<tr><td>3</td><td>0.1054000s</td></tr>
</table>
</td></tr>
<tr><td>4</td><td>Optimised and Parralelized</td><td>
<table>
<tr><th>Trial</th><th>Time</th></tr>
<tr><td>1</td><td>0.0621430s</td></tr>
<tr><td>2</td><td>0.0639820s</td></tr>
<tr><td>3</td><td>0.0638590s</td></tr>
</table>
</td></tr>
</table>

Reflection:

It is interesting to note that there was not a large amount of speedup between naive solution and parallelised solution. We used the default amount threads which has a one to one ratio to the number of cores in the PC. This tells us that the code could be further written in a way, where the data dependencies are removed and further parallel optimizations are possible. However, the speedup for the optimization solution was very pleasing since it almost cut the run-time by half.

Check:
During the initial programming phase I used a small worked out example for sanity check. The small example I have included in the repository and added to the repository. However, I also used the different implementations to compare the results using the naive implementation, the code for that function is shown below:

```
bool check(vector<double> A, vector<double> B) {
    if (A.size() != B.size()) {
        return false;
    }

    for(int i = 0; i < A.size(); i++) {
        if (abs(A[i] - B[i]) >= 0.01) {
            return false;
        }
    }

    return true;
}
```