Optimizing Sparse Lower Triangular Solve

Kei Imada

Optimizations. I developed and tested two potential optimizations to the given starter implementation of sparse lower triangular solve. The code for the optimizations are given in Figure 1. The first (Figure 1a) is a naive parallelization of the innermost loop. The second (Figure 1b) is an implementation of the level set partitioning described in [2] that finds blocks of parallelizable sections of the algorithm using a symbolic analysis of the input matrix.

```
template <typename T>
int parallel_lsolve(CCSMatrix<T> *matrix,
                    DenseVector<T> *input_vector) {
  if (!matrix || !input_vector) {
    return 1;
  }
  int n = matrix->num_col_get();
  int *Lp = matrix->column_pointer_get();
  int *Li = matrix->row_index_get();
  T *Lx = matrix->values_get();
  T *x = input_vector->values_get();
  int p, j;
  if (!Lp || !Li || !x) {
    return 1; /* check inputs */
  }
  for (j = 0; j < n; j++) { // for every col in L, row in x
    x[j] /= Lx[Lp[j]];
#pragma omp parallel default(shared) private(p) num_threads(8)
#pragma omp for
    for (p = Lp[j] + 1; p < Lp[j + 1]; p++) {
      x[Li[p]] -= Lx[p] * x[j];
    }
  }
  return 0;
}
```

(a) `parallel`

```
template <typename T>
int partitioned_parallel_lsolve(CCSMatrix<T> *matrix,
                                DenseVector<T> *input_vector,
                                Partition *partition) {
  if (!matrix || !input_vector || !partition) {
    return 1;
  }
  int *Lp = matrix->column_pointer_get();
  int *Li = matrix->row_index_get();
  T *Lx = matrix->values_get();
  T *x = input_vector->values_get();
  vector<vector<int>> partitioning = partition->partitioning_get();
  if (!Lp || !Li || !x) {
    return 1; /* check inputs */
  }
  for (unsigned int i = 0; i < partitioning.size(); i++) {
    vector<int> partition = partitioning[i];
#pragma omp parallel default(shared) num_threads(8)
#pragma omp for
    for (unsigned int part_idx = 0; part_idx < partition.size();
         part_idx++) {
      int j = partition[part_idx];
      x[j] /= Lx[Lp[j]];
      for (int p = Lp[j] + 1; p < Lp[j + 1]; p++) {
        double tmp = Lx[p] * x[j];
        int x_idx = Li[p];
#pragma omp atomic
        x[x_idx] -= tmp;
      }
    }
  }
  return 0;
}
```

(b) `level_partition`

Figure 1: Two optimizations

Results. I ran the original and the two optimized algorithms on an 8-core Linux system. The full specification of the machine is listed in Figure 2. The code is available on my GitHub. As seen from Figure 3, the two optimizations did not produce a speedup from the original, but rather, exhibited slowdowns of least one magnitude. For both optimizations, it is possible that the given sparse matrices were too small to produce a speedup that surpassed the overhead of OpenMP's thread bookkeeping. Furthermore, C++ constructs such as `vector` were used in the `level_partition` optimization in order to store the partitioning. The overhead of using C++ libraries instead of faster C arrays may have contributed to the slowdown of the two optimizations.

```
OS: Ubuntu 18.04.5 LTS x86_64
Host: XPS 13 9370
Kernel: 5.4.0-62-generic
Shell: zsh 5.4.2
WM: i3
Theme: Adwaita [GTK3]
Icons: Adwaita [GTK3]
Terminal: alacritty
CPU: Intel i7-8550U (8 cores) @ 4.000GHz
GPU: Intel UHD Graphics 620
Total Memory: 15729MiB (16GB)
```

Figure 2: System specification

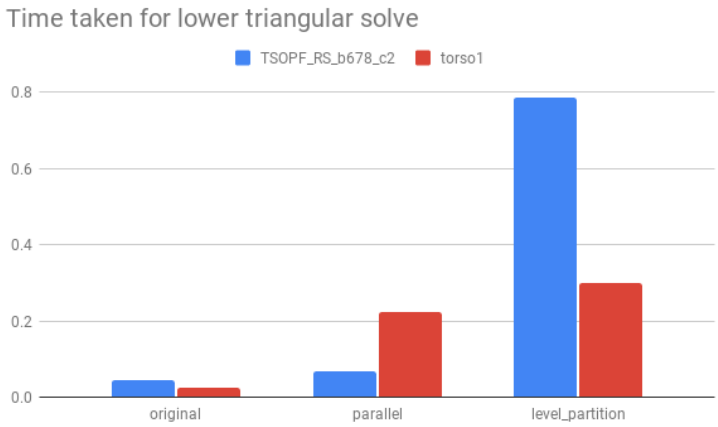

Figure 3: Time taken for sparse lower triangular solve

Development. I spent a nontrivial amount of time on code development. My unfamiliarity of available C++ libraries induced a longer development time. For example, I was inexperienced with C++ graph libraries such as those present in `boost` as well as `unordered_map`, and instead opted to use unoptimized implementations from my college courses. These libraries are only used during the symbolic analysis of the `level_partition` optimization and hence does not impact the triangular solve runtime. Furthermore, my unfamiliarity with the lower triangular solve algorithm and the one week deadline contributed to the overall difficulty of this task. Nevertheless, I was able to fully understand the mechanism of this algorithm through this task and therefore am more confident to produce further optimizations to sparse lower triangular solve using parallel and distributed computing.

Future work. There are many possible improvements to this project. Replacing the C++ constructs in the optimizations with faster C arrays will further optimize the algorithm with a tradeoff of code readability. Furthermore, implementing the h-level partitioning algorithm described in ParSy [1] may produce reliable speedups compared to the two optimizations listed here. The symbolic analysis for the `level_partition` optimization is currently slow since the dependency graph is not pruned into a tree. Reading [2] further to understand the pruning algorithm should allow for a faster symbolic analysis of the input matrices.

[1] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi. *ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism.* IEEE Press, 2018.

[2] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016.