A Hybrid Synchronization Mechanism for Parallel Sparse Triangular Solve

Prabhjot Sandhu, Clark Verbrugge, and Laurie Hendren

School of Computer Science,
McGill University, Montreal, Canada
prabhjot.sandhu@mail.mcgill.ca clump@cs.mcgill.ca hendren@cs.mcgill.ca

Abstract. Sparse triangular solve, SpTS, is an important and recurring component of many sparse linear solvers that are extensively used in many big-data analytics and machine learning algorithms. Despite its inherent sequential execution, a number of parallel algorithms like level-set and synchronization-free have been proposed. The coarse-grained synchronization mechanism of the level-set method uses a synchronization barrier between the generated level-sets, while the fine-grained synchronization approach of the sync-free algorithm makes use of atomic operations for each non-zero access. Both the synchronization mechanisms can prove to be expensive on CPUs for different sparsity structures of the matrices. We propose a novel and efficient synchronization approach which brings out the best of these two algorithms by avoiding the synchronization barrier while minimizing the use of atomic operations. Our web-based and parallel SpTS implementation with this hybrid synchronization mechanism, tested on around 2000 real-life sparse matrices, shows impressive performance speedups for a number of matrices over the classic level-set implementation.

Keywords: Sparse Matrix·Sparse Triangular Solve·SpTS·Performance·Level-set·Synchronization-free·WebAssembly.

1 Introduction

While large and sparse matrices have historically been known to arise frequently in several scientific and compute-intensive applications, many modern big-data analytics and machine learning applications [2,10,12] have become their popular target these days. Likewise, the sparse matrix computations involved in these applications have also become critically important for their performance. Sparse triangular solve (SpTS) is one such recurring computation that is a building block of a number of sparse linear solver algorithms for sparse direct [5] and preconditioned iterative [19] methods in addition to the least-squares problems [3] which are widely used in the machine learning fields [15,24].

In this paper, we focus on SpTS which computes the solution vector x for the equation $Lx = y$, where L is a lower triangular sparse matrix, and y is a dense vector. It is a forward substitution algorithm where the solution of x_i may depend on the solution of $x_0,...,x_{i-1}$. The upper triangular case is analogous, and uses a backward substitution algorithm instead. Unlike other popular sparse kernels like sparse matrix-vector multiplication (SpMV), it is not straightforward to run SpTS operation in parallel, and achieve scalable performance due to this inherent sequential nature of the algorithm.

However, it is still possible to parallelize SpTS by exploiting the very nature of the sparse matrix. One way is to construct a dependency graph representing the dependencies between the components of the solution vector x, and then allow the independent ones to run in parallel. The popular *level-set* method for CPUs employs this approach to create the sets of independent components, and uses a coarse-grained synchronization method to maintain the dependency between those sets. Another algorithm called *synchronization-free* or *sync-free* for short, primarily used for GPUs, avoids generating the dependency information, and uses a fine-grained synchronization method to maintain the dependency between the components themselves. The *level-set* method uses barrier synchronization, while the *sync-free* method uses atomic operations.

Despite their differences, the effectiveness of both of these approaches depends on the sparsity structure of the matrix, and the machine architecture. It is highly impractical to use the *sync-free* method for CPUs due to the heavy use of expensive atomic operations on the limited number of threads over the entirety of this algorithm. A number of sparsity structure patterns can degrade the SpTS performance for the *level-set* method : (1) *large number of sets*, a sparsity structure with a large number of sets may hurt the SpTS performance due to the increased number of synchronization barriers; (2) *small and varied number of components per set*, a structure pattern with a few and a varying number of components per set may waste the assigned CPU resources and incur an unnecessary maintenance overhead; (3) *uneven distribution of non-zeros among the rows*, a sparsity structure with a highly uneven number of non-zeros to be processed for the components within the same set may lead to load imbalance among the worker threads. In order to address these challenges, we propose a synchronization mechanism that is less coarse-grained than the *level-set* method and less fine-grained than the *synchronization-free* method at the same time.

Following are the main contributions of our work in this paper:

– We present a novel synchronization approach, a hybrid between the *level-set* and *sync-free* algorithms, to efficiently run SpTS in parallel on CPUs. Our cost-effective busy-waiting synchronization strategy is built upon the use of two different synchronization modes and the dynamic switching between them.
– We employ a row classification technique to minimize the use of expensive atomic operations in the WebAssembly [8] environment.
– We tackle the issues with costly synchronization barriers between the level-sets by eliminating them, and developing SpTS to be less sensitive to the number of worker threads employed in comparison to the *level-set* method.
– We implement web-based and parallel SpTS using our hybrid synchronization method on WebAssembly and JavaScript, and evaluate its performance in comparison to the *level-set* method over almost 2000 real-life sparse matrices, and demonstrate impressive performance speedups.

2 Motivation and Related Work

SpTS, for being a key component of popular sparse linear solvers, continues to attract the attention of high-performance computing (HPC) researchers. The level-set method, proposed by Anderson and Saad [1] and Saltz [20] in the early 1990s, is a classic and well-known technique to solve SpTS in a parallel environment on CPUs. This graph-based algorithm involves a preprocessing step to create the level sets, and a costly synchronization barrier after each level to satisfy the dependencies between the levels. As a result, a number of recent research contributions have been made to analyze and overcome these limitations [25,18] and also to improve the parallel performance of this sparse kernel on modern machines [11,6,16,7,23,13].

Wolf et al. [25] analyzed a few factors like barrier type that impact the performance of multithreaded SpTS, favouring the use of more active barriers (spin locks) instead of the passive barriers (blocking locks). In order to reduce the performance overhead of inter-level synchronization, Park et al. [18] presented a synchronization sparsification technique that performs point-to-point synchronization between the *super tasks*, however with an involved and expensive preprocessing stage to reduce the number of dependency edges in the task dependency graph. While sharing the same goal, our synchronization strategy avoids both the synchronization barriers and the intricate preprocessing on the task dependency graph.

On the other hand, for manycore platforms, such as GPUs, Liu et al. [11] exposed the parallelism in SpTS for CSC storage format by making use of atomic operations for synchronization rather than creating the level-sets [6,16]. It is called a synchronization-free algorithm which reduced the preprocessing cost and completely eliminated the conventional synchronization between the levels. Dufrechou and Ezzati [7] later showed the performance improvements for sync-free algorithm using CSR format over the CSC format. Following this, Su et al. [23] recently proposed CapelliniSpTRSV, a thread-level instead of warp-level sync-free parallelism technique to improve the performance of matrices with a large number of rows per level and a small number of non-zeros per row. Lu et al. [13] implemented a recursive block algorithm to solve SpTS in parallel on GPUs while improving upon the two-dimensional blocking technique previously proposed by Mayer [14] to divide the triangular matrix into multiple triangular sub-matrices and rectangular or square sub-matrices.

Next, Yilmaz et al. [26] presented adaptive level binning to balance the workloads among the threads while making use of some features from both level-set and sync-free methods. Our approach in this paper is to reap the benefits of both level-set and sync-free methods in a more simplified and distinct manner to build an efficient sparse triangular solve. Several web-based machine learning frameworks like TensorFlow.js [17] train and deploy models in web browsers. Existing studies [9,21,22] have thus analyzed the performance of their extensively used sparse computations like SpMV in a web context. We develop a web-based sparse triangular solve which is also one of the most critical sparse BLAS (Basic Linear Algebra Subprograms) routines. In addition to that, our work is a step towards building a web-based scientific computing framework that will provide optimized and parallel sparse BLAS routines, and has clear applicability as a high-performance backend for the leading ML frameworks.

```
for i = 0 to N−1 do
  x[i] = 0
  for j = row_ptr[i] to row_ptr[i+1] − 2 do
    x[i] += val[j] ∗ x[col[j]]
  end for
  x[i] = (y[i] − x[i])/val[row_ptr[i+1] − 1]
end for
```

Listing 1.1: A serial SpTS CSR algorithm to solve x in Lx = y

3 Preliminaries

In this section, we provide some background details for SpTS operation which includes its widely used, classic and state-of-the-art serial and parallel algorithms.

3.1 Sparse Matrix and Serial SpTS

Given a nonsingular lower triangular sparse matrix in the external format, it is stored in the compressed sparse row (CSR) format as shown in Figure 1. It is the most widely used internal sparse storage format which consists of three arrays `row_ptr`, `col` and `val`. While `col` and `val` arrays store the column index and the value of each non-zero entry in a row-major order, `row_ptr` array stores the starting index of each row, pointing at the other two arrays.

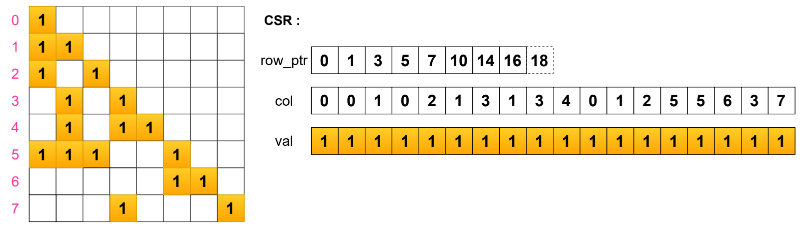

CSR :
row_ptr 0 1 3 5 7 10 14 16 18
col 0 0 1 0 2 1 3 1 3 4 0 1 2 5 5 6 3 7
val 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Fig. 1: An example of a lower triangular matrix in the CSR format.

In the serial SpTS CSR algorithm to solve `x` in the equation $Lx = y$, the sparse matrix rows are traversed sequentially, and each value of the solution vector `x` is solved accordingly. It is natural to solve it in this way because the solution of x_i may depend on the solution of x_0,...,x_{i-1}. As illustrated in Listing 1.1, for each x_i, the sum of the product of non-zero entries (excluding the diagonal entry) from the sparse matrix row `i` with the corresponding solutions of `x`, indicated via the column index of each non-zero entry, is calculated. Next, the solution of x_i is calculated by subtracting this sum from y_i, and then dividing it with the diagonal entry.

3.2 Parallel SpTS

Due to the sparse structure of a lower triangular sparse matrix in a SpTS operation, it is possible to find such matrix rows which are independent of each other, and can be solved in parallel. The level-set method and the synchronization-free algorithm are two popular techniques that exploit this property to run SpTS in parallel.

Level-set The preprocessing stage of this method makes sets of the matrix rows which can be solved independently and simultaneously as shown in Figure 2 for the example matrix from Figure 1. The cardinality of these sets describes the amount

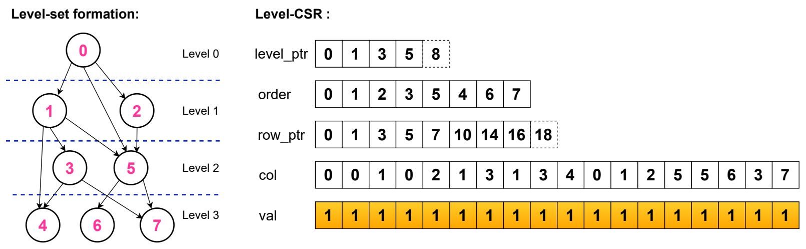


Fig. 2: An example of level-set formation for a lower triangular sparse matrix.

of parallelism available. A specific order is maintained between these sets, also called levels, to satisfy the dependency between them. This process basically converts the fine-grained dependencies between the rows into a coarse-grained dependency between the levels. In the parallel SpTS CSR level-set algorithm as shown in Listing 1.2, this coarse-grained dependency between the levels is satisfied by the use of a synchronization barrier after the completion of SpTS computation at each level.

```
for l = 0 to nlevels − 1 do
  for k = level_ptr[l] to level_ptr[l+1]
      − 1 in parallel do
    i = order[k]
    x[i] = 0
    for j = row_ptr[i] to row_ptr[i+1] −
        2 do
      x[i] += val[j] * x[col[j]]
    end for
    x[i] = (y[i] − x[i])/val[row_ptr[i
        +1] − 1]
  end for
  // barrier synchronization
end for
```

Listing 1.2: A simplified SpTS CSR level-set algorithm to solve x in Lx = y

```
for i = 0 to N−1 in parallel do
  x[i] = 0
  for j = row_ptr[i] to row_ptr[i+1] − 2
      do
    while atomic_read(flag[col[j]]) != 1
        do
      // busy−wait
    end while
    x[i] += val[j] * x[col[j]]
  end for
  x[i] = (y[i] − x[i])/val[row_ptr[i+1]
      − 1]
  atomic_write(flag[i], 1)
end for
```

Listing 1.3: A simplified SpTS CSR sync-free algorithm to solve x in Lx = y

Synchronization-free This algorithm eliminates the cost of barrier synchronizations between the levels by establishing a busy-waiting mechanism and making use of expensive atomic load/store operations as shown in Listing 1.3.

4 Our Approach

In this section, we first give an overview of our approach, and next provide the details on the internals of our hybrid synchronization strategy.

4.1 Overview

The level-set method employs a coarse-grained synchronization mechanism that works at each level, while the sync-free method applies a fine-grained synchronization scheme that works at each row. Both of these synchronization mechanisms have their own benefits depending on the structure of the matrix. Theoretically, the sparse matrices with a few levels and a significant and balanced workload among the given worker threads will prefer the level-set method. The sync-free method may prove useful otherwise. However, the situation may be different at different levels depending on the structure of the matrix. Hence, we intend to follow a middle path between these two algorithms to reap the benefits of both. The objective of our algorithm is to improve the performance of parallel SpTS by avoiding the synchronization barrier while minimizing the use of atomic operations as much as possible.

(1) No synchronization barrier with level-set formation Unlike the sync-free approach, we choose to keep the preprocessing step of the level-set method. We build the level-set formation in a simplified manner, and reorder the rows to bring together the rows from the same level-set to have spatial locality. We adopt this formation because it is a systematic way to guarantee that the worker threads at the same level can make progress independently and simultaneously. This becomes the basis of our simplified synchronization technique.

However, like the sync-free algorithm, we decide to employ no barrier synchronization after each level because the worker thread does no useful work while waiting at the barrier. Instead, we allow each worker thread to immediately start processing the next level after it is done with the current level, without waiting for other worker threads to reach the end of the same level. In order to maintain the accuracy of the algorithm, we employ a novel and effective busy-waiting synchronization mechanism described in detail in the following sections.

(2) Minimize the use of atomic operations In the sync-free system as shown in Listing 1.3, an atomic load operation is performed repeatedly for each non-zero until the value of vector x at the *required row* (represented by the column index of the non-zero) is completely solved and available for use. This availability is indicated after the complete computation of the value of vector x at each row by using an atomic store operation. Due to the heavy use of expensive atomic operations on the limited number of threads over the entirety of this algorithm, this system proves highly ineffective on the CPUs.

However, in order to avoid the synchronization barrier, we need another synchronization strategy to make sure that the value of solution vector x at a given row is completely solved before its use. As a result, it seems appealing to develop a hybrid synchronization mechanism that brings out the best of both the level-set method and the sync-free method for CPUs. A straightforward candidate solution is to employ the sync-free busy-waiting mechanism for the worker threads which have been advanced to the next levels, while other worker threads are still working at the current level. However, it continues to be an unproductive solution as it still requires an atomic store operation for each row to indicate the availability of the value of solution vector x at a given row.

Therefore, we employ a number of techniques to minimize the use of atomic operations, and build a more cost-effective busy-waiting synchronization strategy. In addition to that, we also keep a simplified synchronization mechanism where no atomic operations and busy-waiting are needed. We call our two synchronization techniques as `no-busy-wait` and `busy-wait`, as shown in Figure 3, and dynamically switch between the two during the course of SpTS computation as many times as required.

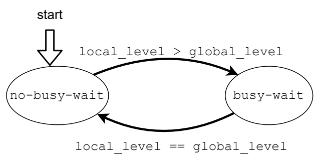
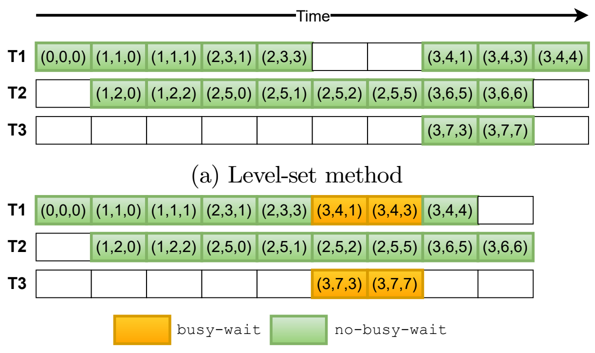


Fig. 3: Our synchronization modes and their dynamic switching conditions.



(a) Level-set method

(b) Our hybrid method

Fig. 4: An example with a timeline to show the parallel SpTS workflow for algorithms with different synchronization mechanisms.

4.2 no-busy-wait

This synchronization mode is set by default at the start of the SpTS computation. As the name implies, it doesn't involve any busy-waiting, and each thread can safely assume that the value of vector x at the *required row* is completely solved and available for use. Let's suppose `local_level` to be the current working level of a worker thread, and `global_level` to be the maximum working level achieved by all the worker threads. The value of `global_level` will always be less than or equal to `local_level` for any worker thread. A worker thread can work in this synchronization mode if its `local_level` is equal to `global_level`. Otherwise, the worker thread continues its work for the next level after switching its synchronization mode to `busy-wait`, the details of which will be presented in the following section.

Let's use the example matrix and its level-set formation from Figures 1 and 2 to inspect the SpTS workflow in a parallel environment while applying our synchronization strategy. We assume three worker threads T1, T2 and T3 in Figure 4 to show the SpTS computation timeline with each 3-tuple representing (`local_level`, `row`, `column`) for the processing of each non-zero.

While processing the non-zero, the *required row* index for the vector x is represented by the `column` value of the 3-tuple. For example, the tuple (3,4,3) for the worker thread T1 means that the current working level of thread T1 is 3, and it is processing the non-zero at 4th row and 3rd column of the matrix, and it requires the value of x[3] to proceed with the computation. All the worker threads start their work in the `no-busy-wait` mode as illustrated in Figure 4b. The worker threads T1 and T3 switch to `busy-wait` as their `local_level` reaches 3, represented by the tuples (3,4,1) and (3,7,3) respectively. But at the same time, T2 continues to work in the `no-busy-wait` mode, keeping the value of `global_level` to be 2. If it had been the conventional level-set method as shown in Figure 4a, the worker threads T1 and T3 must wait at the barrier until `global_level` becomes equal to their `local_level`, which happens after T2 also finishes its work at level 2.

4.3 busy-wait

A worker thread switches to this mode when its `local_level` becomes greater than `global_level` to allow itself to make feasible advancements and avoid the costly barrier. Unlike `no-busy-wait`, this mode doesn't provide any safety guarantees by default that the value of vector x at the *required row* is completely solved and available for use. As indicated earlier, the busy-waiting mechanism of the sync-free algorithm is a candidate solution, but it will require the worker thread to perform atomic store operations for each row even during the `no-busy-wait` mode. Therefore, we employed more cost-effective busy-waiting techniques to enable the worker thread to make progress in a safely manner.

We classify the required rows into four exhaustive categories : (1) *previous-level rows* (2) *intra-thread rows* (3) *advanced-worker inter-thread rows* (4) *inter-thread rows*. The worker thread identifies the *required row* to be one of these categories, and decides whether to proceed with the computation or wait. If the *required row* belongs to any one of them except (4), it is safe for the worker thread to perform the computation as illustrated by a flowchart in Figure 5. It is due to this classification that we are able to minimize the use of atomic operations. Listing 1.4 shows a portion of SpTS implementation in WebAssembly, providing details of the internals of our technique in the `busy-wait` synchronization mode.

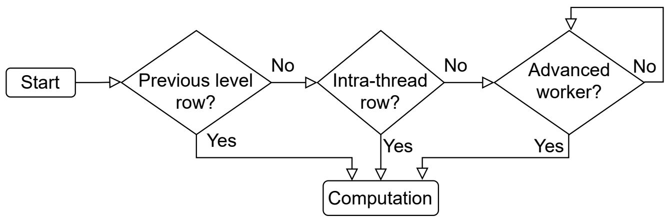


Fig. 5: An overview of the internals of the `busy-wait` synchronization mode for each non-zero in SpTS computation.

```
(f32.load (local.get $csr_val))
(i32.load (local.get $csr_col))
(local.set $required_row)
(i32.atomic.load (local.get $global_row_index))
(local.get $required_row)
(i32.le_s)
if
  (i32.load (i32.add (local.get $row_worker_index) (i32.shl (local.get $required_row)
      (i32.const 2))))
  (local.set $worker)
  (local.get $worker)
  (local.get $current_worker)
  (i32.ne)
  if
    (i32.load (i32.add (local.get $row_level_index) (i32.shl (local.get $required_row
        ) (i32.const 2))))
    (local.set $required_level)
    (loop $busy_wait_loop
      (local.get $required_level)
      (i32.atomic.load (i32.add (local.get $worker_level_index) (i32.shl (local.get
          $worker) (i32.const 2))))
      (i32.gt_s)
      (br_if $busy_wait_loop)
    )
  end
end
(f32.load (i32.add (local.get $x) (i32.shl (local.get $required_row) (i32.const 2))))
(f32.mul)
```

Listing 1.4: A portion of SpTS WebAssembly implementation in the `busy-wait` synchronization mode

(1) Previous level rows are the rows that belong to the preceding levels which have been completed by all the worker threads. It means the value of x at the required rows from this category must be available for use. We use `global_row_index` to represent the maximum row index of the level completed by all the worker threads. This value is atomically updated to the maximum row index of a particular level when all the worker threads finish their computation at that level. The worker thread atomically reads this value to identify the category of its required row. If `global_row_index` is greater than the required row, it is confirmed that the required row belongs to this category.

For example, the worker thread T1 from Figure 4b in the `busy-wait` mode at the tuple (3,4,1) identifies that the required row 1 (represented by the column value of this tuple) belongs to the previous level category, and hence proceeds with the computation.

(2) Intra-thread rows are the rows processed by the worker thread itself. It means the value of x at those required rows must already be completely solved. We statically store the mapping between the rows and the worker threads. The worker thread uses this information to identify if the required row belongs to this category. Since this mapping is static, non-atomic loads are sufficient to perform this check.

For example, in the `busy-wait` mode, the required row 3 from the tuple (3,4,3) of the worker thread T1 in Figure 4b was indeed processed by T1 itself. Therefore, it qualifies as an intra-thread row, and T1 goes on to carry out the computation.

(3) Advanced-worker inter-thread rows are the rows that are processed by other worker threads which are also ahead of `global_level`. In addition to that, these rows belong to those levels which are already completed by those worker threads. We statically have the mapping between the rows and the level sets, and also between the rows and the worker threads. The required level and the corresponding worker thread information is calculated using these mappings. Each worker thread atomically updates its recently completed level as it proceeds with its computations. Therefore, the given worker thread atomically reads the recently completed level of the corresponding worker thread. If it is greater than or equal to the required level, the given worker thread can safely assume that the value of x at the required row is available for use.

For example, the required row 3 belongs to the advanced-worker inter-thread category for worker thread T3 at the tuple (3,7,3) in Figure 4b. T3 determines this by checking that the required row 3 and its corresponding level 2 have been completely processed by the advanced worker thread T1.

(4) Inter-thread rows are the rows that are processed by other worker threads, and belong to the level which is not yet finished by those worker threads. Hence, the given worker thread needs to wait for the value to become available. In contrast to the sync-free algorithm, each worker thread atomically declares the completion of a level instead of the completion of a row as it proceeds with its computations. We have followed this style to reduce the cost of atomic store operations.

Since this mode involves busy-waiting, it is naturally more expensive than the `no-busy-wait` mode. In order to keep the benefits of not performing the atomic load operations when not needed, a worker thread checks before the computation on each non-zero if it is possible to switch back to its default synchronization mode. Therefore, the worker thread switches back to `no-busy-wait` mode if its `local_level` becomes equal to `global_level`.

5 Evaluation

In this section, we first describe our experimental layout. Next, we evaluate the SpTS performance using our WebAssembly implementations of both hybrid and level-set synchronization methods on our benchmark matrices.

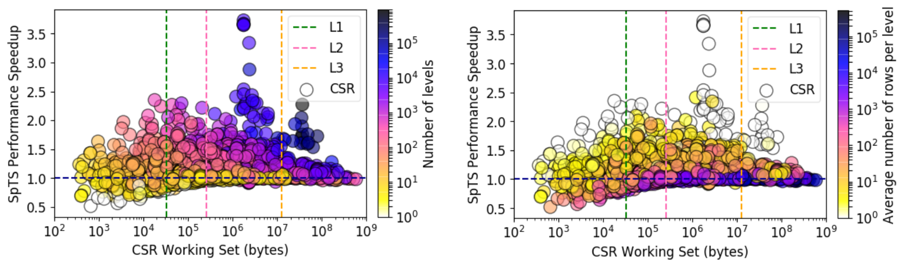
5.1 Experimental Setup

We conducted our experiments on an Intel Core i7-3930K with 6 3.20GHz cores, 12MB last-level cache and 16GB memory, running Ubuntu Linux 18.04.5. Our execution environment for WebAssembly is the Chrome 92 browser (Official build 92.0.4515.107 with V8 JavaScript engine 9.2.230.20). We ran headless Chrome with two flags, `--wasm-no-bounds-checks` and `--wasm-no-stack-checks` to avoid memory bounds checks and stack guards for performance testing. We used `Date.now()`, a JavaScript method to measure the execution time.

Our set of sparse matrix benchmarks consists of 1,957 real-life square sparse matrices from The SuiteSparse Matrix Collection [4]. We use Matrix Market external format as an input to our programs, and store the lower triangular portion of the sparse matrices in the internal format to solve Lx = y. We keep all the diagonal elements to be non-zeros to avoid the sparse matrix from being singular.

5.2 SpTS Performance Comparison

Figure 6 shows the single-precision SpTS performance speedup of using hybrid over level-set synchronization method for our benchmark matrices with *CSR Working Set* on the x-axis, which is calculated as $((nlevels + 1) + (N + 1) + 2 * nnz + 2 * N) * 4$, where *nlevels* is the number of generated levels for a N X N lower triangular sparse matrix with *nnz* number of non-zeros.



(a) Speedup vs number of levels (b) Speedup vs avg number of rows per level

Fig. 6: Performance speedup of our hybrid method over the level-set method for different sparse matrices depicted using different structure features

It is evident that the hybrid method shows quite promising performance speedups for a number of matrices especially with a large number of levels. Apart from that, we notice various different speedup results for different sparse matrices from the plot. It indicates that the choice of synchronization method can have a significant impact on the SpTS performance. Using our observation, we classify the speedup results into three categories : (1) *Below 1* (2) *Above 1* (3) *Close to 1*, and evaluate them on an individual basis.

Below 1 This category belongs to the set of matrices that shows better SpTS performance for the level-set synchronization method. Figure 6 clearly shows that the *CSR Working Set* size of the matrices from this category is usually small. In order to understand the properties of this set, we show a small subset of these matrices in Table 1, which largely represents the whole set of the *Below 1* matrices. First of all, we can observe from this table that some of these matrices have a large number of rows per level. It indicates that each worker thread likely receives a substantial amount of workload. Further examination uncovers that there is a nearly balanced workload among the worker threads at each level for these matrices. Since the cost of synchronization barriers becomes insignificant due to the nearly balanced and significant workload, these matrices perform better with the level-set synchronization method.

Next, the matrices with a very small number of rows per level like *psmigr_3* indicate that almost no parallelism is available for the worker threads. However, the large number of non-zeros per row likely reduces the cost of synchronization barriers. There-

Matrix	**N**	**nnz**	**nlevels**	**N/nlevels**	**Performance(GFLOPS)**		**Speedup**
					Level-set	**Hybrid**	
t2dal_e	4257	4257	1	4257	1.70	1.52	0.89x
iprob	3001	6001	2	1500.5	2.96	2.71	0.92x
bcsstm11	1473	1473	1	1473	1.37	1.07	0.78x
grid2	3296	9728	3	1098.6	3.36	3.07	0.91x
SmaGri	1059	1117	4	264.7	0.59	0.46	0.78x
fpga_dcop_09	1220	3857	6	203.3	1.49	1.34	0.89x
bcspwr08	1624	3837	14	116	0.89	0.74	0.83x
t3dl_a	20360	265113	633	32.2	1.32	1.22	0.92x
exdata_1	6001	1137751	1501	3.99	1.47	1.37	0.93x
psmigr_3	3140	278874	1638	1.91	0.89	0.82	0.92x

Table 1: Representative matrices from Below 1 category

fore, in both of these conditions, our approach suffers from the overhead of switching the synchronization modes via function calls and the atomic operations performed.

Above 1 This set of matrices shows better SpTS performance for the hybrid synchronization method. We list the details and the actual performance numbers of a small subset of these matrices in Table 2. First of all, we can observe from this table that these matrices have a large number of levels which means there are a large number of synchronization barriers involved for the level-set method.

Matrix	**N**	**nnz**	**nlevels**	**N/nlevels**	**Performance(GFLOPS)**		**Speedup**
					Level-set	**Hybrid**	
lung2	109460	273647	479	228.5	1.47	2.28	1.55x
dblp-2010	326186	1133886	1562	208.8	1.43	1.8	1.25x
delaunay_n17	131072	524248	910	144.0	1.36	1.82	1.34x
shallow_water1	81920	204800	1016	80.6	0.58	0.87	1.50x
twotone	120750	734744	1695	71.2	1.28	1.72	1.34x
e40r0100	17281	257727	512	33.7	1.49	2.06	1.38x
ted_A	10605	313099	1217	8.7	1.10	1.74	1.58x
ship_001	34920	1965708	4654	7.5	1.33	1.66	1.25x
smt	25710	1887646	4646	5.5	1.27	1.72	1.35x
t2em	921632	2756232	871133	1.05	0.012	0.028	2.33x

Table 2: Representative matrices from Above 1 category

Next, there are a small to moderate number of rows per level which indicates that a limited amount of workload is available for each worker thread. Our further investigation reveals that the distribution of rows among the levels is highly uneven. A small number of levels have a very large number of rows, while others have quite small. This imbalanced distribution of rows among the levels further limits the amount of workload per worker thread, leading them to be stuck at the synchronization barrier most of their lifetime for the level-set method. On the other hand, our approach benefits from allowing the worker threads to move to further levels to perform some feasible part of the SpTS computation.

Finally, there are a few matrices like *t2em* with a large number of levels and a very small number of rows per level. Although these matrices show great performance speedup for the hybrid method, they are potentially less interesting for the comparison due to their nearly nonexistent parallelism, and low absolute performance numbers.

Close to 1 Finally, this set of matrices shows similar SpTS performance for both hybrid and level-set synchronization methods. We show a small subset of these matrices along with their structure parameters in Table 3.

It is intriguing to notice that the matrices from this category have a varied number of levels, starting from as low as 1 and ranging up to a large number. We observe the presence of diagonal matrices of different sizes (with a number of levels equal to 1) in both *Below 1* and *Close to 1* categories. It shows that the overhead becomes insignificant for the large matrices with a small number of levels. However, for the matrices with a large number of levels, we investigated to find out that the workload among the worker threads at each level is a little imbalanced. The workload imbalance is such that the SpTS performance gain from some levels in our approach gets cancelled out by the overhead from other levels.

Matrix	**N**	**nnz**	**nlevels**	**N/nlevels**	**Performance(GFLOPS)**		**Speedup**
					Level-set	**Hybrid**	
mbeacxc	496	30309	214	2.3	0.76	0.76	1.00x
s3dkq4m2	90449	2259087	2369	38.2	1.39	1.40	1.01x
coPapersCiteseer	434102	16470822	8087	53.7	2.26	2.24	0.99x
TSOPF_RS_b39_c7	14098	238270	106	133	3.20	3.22	1.01x
kron_g500-logn18	262144	10844830	1820	144	1.21	1.19	0.98x
wikipedia-20051105	1634989	15512976	1273	1284.3	1.68	1.68	1.00x
hangGlider_5	16011	89187	6	2668.5	3.97	4.03	1.02x
t3dl_e	20360	20360	1	20360	1.87	1.83	0.98x
rajat29	643994	2294300	29	22206.6	3.97	4.03	1.01x
ins2	309412	1530448	8	38676.5	3.11	3.14	1.01x
parabolic_fem	525825	2100225	7	75117.8	4.27	4.27	1.00x

Table 3: Representative matrices from Close to 1 category

The analysis of these three different types of speedup results demonstrates the impact of the structure of the matrix on the choice of the synchronization method. A single synchronization method is therefore not appropriate for all the given input matrices. Our evaluations prove the potential of our hybrid method to support an adaptive synchronization technique for SpTS on CPUs based on the structure of the matrix.

6 Conclusion and Future Work

The limitations of the level-set method for different sparsity structures of the matrices, and the ineffectiveness of the sync-free algorithm on CPUs led us to develop our hybrid synchronization method. Our strategy specifically targeted the granularity of the existing synchronization techniques to overcome their performance bottlenecks. While keeping the level-set formation, we avoided the synchronization barriers and minimized the use of atomic operations. We tested our WebAssembly SpTS implementation with this hybrid synchronization approach on around 2000 matrices, and demonstrated impressive speedups for several matrices over the classic level-set implementation.

Our future directions include the exploration of more sparse storage formats and optimization techniques like SIMD in addition to the improvements over the present parallelization strategies. It involves using the upcoming synchronization constructs like atomic floating-point operations from the rapidly expanding WebAssembly instruction set. Besides, the non-trivial task to automatically figure out the best synchronization method for SpTS at runtime for a given sparse matrix has intrigued us to explore the pertinent matrix structure features for developing such techniques in the future.

7 Acknowledgments

This work was supported by the COHESA project, through NSERC Strategic Networks grant NETGP485577-15.

References

1. Anderson, E., Saad, Y.: Solving sparse triangular linear systems on parallel computers. Int. J. High Speed Comput. **1**(1), 73–95 (1989)
2. Baoyuan Liu, Min Wang, Foroosh, H., Tappen, M., Penksy, M.: Sparse convolutional neural networks. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 806–814 (2015)
3. Björck, A.: Numerical Methods for Least Squares Problems. Society for Industrial and Applied Mathematics (1996)
4. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS) **38**(1), 1 (2011)
5. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, Inc., USA (1986)
6. Dufrechou, E., Ezzatti, P.: A new gpu algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 920–929 (2018)
7. Dufrechou, E., Ezzatti, P.: Solving sparse triangular linear systems in modern gpus: A synchronization-free algorithm. In: 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 196–203 (2018)
8. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with webassembly. In: PLDI 2017. pp. 185–200. ACM
9. Herrera, D., Chen, H., Lavoie, E., Hendren, L.: Numerical computing on the web: Benchmarking for the future. In: Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages. pp. 88–100. DLS 2018, ACM, New York, NY, USA (2018)
10. Kepner, J., Gilbert, J.: Graph Algorithms in the Language of Linear Algebra. Society for Industrial and Applied Mathematics (2011)
11. Liu, W., Li, A., Hogg, J., Duff, I.S., Vinter, B.: A synchronization-free algorithm for parallel sparse triangular solves. In: Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833. pp. 617–630. Springer-Verlag, Berlin, Heidelberg (2016)

12. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.: Graphlab: A new framework for parallel machine learning. In: Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence. pp. 340–349. UAI'10, AUAI Press, Arlington, Virginia, USA (2010)
13. Lu, Z., Niu, Y., Liu, W.: Efficient block algorithms for parallel sparse triangular solve. In: 49th International Conference on Parallel Processing - ICPP. ICPP '20, Association for Computing Machinery, New York, NY, USA (2020)
14. Mayer, J.: Parallel algorithms for solving linear systems with sparse triangular matrices. Computing **86**(4), 291–312 (Nov 2009)
15. Natarajan, R., Sindhwani, V., Tatikonda, S.: Sparse least-squares methods in the parallel machine learning (pml) framework. In: 2009 IEEE International Conference on Data Mining Workshops. pp. 314–319 (2009)
16. Naumov, M.: Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011 (2011)
17. Nikhil Thorat, Daniel Smilkov, and Charles Nicholson: TensorFlow.js - A WebGL accelerated browser based JavaScript library for training and deploying ML models, `https://js.tensorflow.org`
18. Park, J., Smelyanskiy, M., Sundaram, N., Dubey, P.: Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In: International Supercomputing Conference. pp. 124–140. Springer (2014)
19. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics (2003)
20. Saltz, J.H.: Aggregation methods for solving sparse triangular systems on multiprocessors. SIAM J. Sci. Stat. Comput. **11**(1), 123–144 (Jan 1990)
21. Sandhu, P., Herrera, D., Hendren, L.: Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in javascript and webassembly. In: Proceedings of the 15th International Conference on Managed Languages & Runtimes. ManLang '18, Association for Computing Machinery, New York, NY, USA (2018)
22. Sandhu, P., Verbrugge, C., Hendren, L.: A fully structure-driven performance analysis of sparse matrix-vector multiplication. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering. pp. 108–119. ICPE '20, Association for Computing Machinery, New York, NY, USA (2020)
23. Su, J., Zhang, F., Liu, W., He, B., Wu, R., Du, X., Wang, R.: Capellinisptrsv: A thread-level synchronization-free sparse triangular solve on gpus. In: 49th International Conference on Parallel Processing - ICPP. ICPP '20, Association for Computing Machinery, New York, NY, USA (2020)
24. Sun, L., Ji, S., Ye, J.: A least squares formulation for a class of generalized eigenvalue problems in machine learning. In: Proceedings of the 26th Annual International Conference on Machine Learning. pp. 977–984. ICML '09, Association for Computing Machinery, New York, NY, USA (2009)
25. Wolf, M.M., Heroux, M.A., Boman, E.G.: Factors impacting performance of multithreaded sparse triangular solve. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) High Performance Computing for Computational Science – VECPAR 2010. pp. 32–44. Springer Berlin Heidelberg (2011)
26. Yılmaz, B., Sipahioğrlu, B., Ahmad, N., Unat, D.: Adaptive level binning: A new algorithm for solving sparse triangular systems. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. pp. 188–198. HPCAsia2020, Association for Computing Machinery, New York, NY, USA (2020)