Speculative Parallel Reverse Cuthill-McKee Reordering on Multi- and Many-core Architectures

Daniel Mlakar, Martin Winter, Mathias Parger, Markus Steinberger
Graz University of Technology, Austria
{daniel.mlakar, martin.winter, mathias.parger, steinberger}@icg.tugraz.at

***Abstract*—Bandwidth reduction of sparse matrices is used to reduce fill-in of linear solvers and to increase performance of other sparse matrix operations, *e.g.*, sparse matrix vector multiplication in iterative solvers. To compute a bandwidth reducing permutation, Reverse Cuthill-McKee (RCM) reordering is often applied, which is challenging to parallelize, as its core is inherently serial. As many-core architectures, like the GPU, offer subpar single-threading performance and are typically only connected to high-performance CPU cores via a slow memory bus, neither computing RCM on the GPU nor moving the data to the CPU are viable options. Nevertheless, reordering matrices, potentially multiple times in-between operations, might be essential for high throughput. Still, to the best of our knowledge, we are the first to propose an RCM implementation that can execute on multicore CPUs and many-core GPUs alike, moving the computation to the data rather than vice versa.**

Our algorithm parallelizes RCM into mostly independent batches of nodes. For every batch, a single CPU-thread/a GPU thread-block speculatively discovers child nodes and sorts them according to the RCM algorithm. Before writing their permutation, we re-evaluate the discovery and build new batches. To increase parallelism and reduce dependencies, we create a signaling chain along successive batches and introduce early signaling conditions. In combination with a parallel work queue, new batches are started in order and the resulting RCM permutation is identical to the ground-truth single-threaded algorithm.

We propose the first RCM implementation that runs on the GPU. It achieves several orders of magnitude speed-up over NVIDIA's single-threaded cuSolver RCM implementation and is significantly faster than previous parallel CPU approaches. Our results are especially significant for many-core architectures, as it is now possible to include RCM reordering into sequences of sparse matrix operations without major performance loss.

***Index Terms*—many-core, multicore, CPU, GPU, Reverse Cuthill-McKee, scheduling, work distribution**

I. Introduction

Sparse matrices are essential in various scientific computing domains: the finite element method heavily relies on sparse matrices [1]; large and complex systems of equations can often only be expressed as sparse systems [2]; highly constrained linear programming problems are subject to sparse optimizations [3]; electrical circuit design and simulation use sparse representations [4]; as 3D meshes grow very large, sparse matrices also find their way into geometry processing [5]. Due to the ever rising amount of information, sparse data structures are getting even more important in many disciplines: to capture social or communication networks [6]; for natural language processing [7]; and for sparse neural networks [8].

It is well known that the sparsity pattern of a matrix can affect performance of matrix operations [2], [9], [10]. While the matrix bandwidth is a good indicator for the fill-in, *e.g.*, in Cholesky solvers, it also dictates memory access patterns in sparse matrix operations, which, in turn, dictate caching behavior. On many-core architectures like the GPU, where algorithms are often memory bound, good access patterns are crucial. Thus, matrices are often reordered to reduce their bandwidth, *i.e.*, non-zero entries are placed closer to the diagonal. One of the most prominent reordering heuristics is the Reverse Cuthill-McKee (RCM) algorithm [11], [12].

RCM is predestined for serial execution, as the algorithm follows a Breadth-First-Search (BFS) of the graph view on the matrix, *i.e.*, rows and columns as nodes and non-zero entries as edges. In contrast to BFS, RCM also imposes that child nodes are visited in order of their degree. Thus, efficient parallel RCM is difficult, as the algorithm exhibits a complex pattern, known as *amorphous data-parallelism*, *i.e.*, parallelism and dependencies irregularly unfold during runtime and thus must be found and exploited dynamically. Previous parallelization attempts are scarce [13]–[15]. They follow the BFS pattern which they either execute speculatively or level-by-level. All three approaches exclusively target CPU architectures. No parallel solutions of RCM exist for modern supercomputing, where execution is often accelerated with many-core architectures like the graphics processing unit (GPU). Thus, the only way to reorder matrices on GPU-backed systems is to move them to the host, reorder and transfer them back to the GPU over a slow memory bus, delaying further execution.

To address these issues, we propose a novel parallelization of the RCM algorithm, which targets both multicore and many-core architectures. We make the following contributions:

- We present a batch-based parallelization of the RCM algorithm that can be distributed across large core counts and scales with the parallelism inherent to the matrix.
- We present a speculative node discovery approach, which balances speculative execution and ensures that compute-intensive sorting is never held back.
- We show that multiple handovers and chained signal propagation enable parallel progress. Additionally, data is kept local, effectively minimizing memory transactions.
- We include work aggregation in our signaling scheme to generate well balanced per-batch workloads.
- By porting our algorithm to the GPU, we present the first RCM implementation that targets many-core architectures. Our approach works completely in local scratchpad memory and does not require additional memory.

We evaluated our approach on a single 12 core CPU node and on a recent GPU architecture and observed speed-ups of up to $24.9\times$ and $13.0\times$ compared to the optimized single-core variant and $15.2\times$ and $18.7\times$ to previous work.

II. Related work

It is known that the distribution of non-zero entries in a sparse matrix can have a large impact on the performance of matrix operations [2], [9]. Recently this has also been shown for operations on the GPU [16]. A minimal bandwidth, *i.e.*, a minimum distance of matrix entries to the diagonal is associated with good performance. However, finding a minimum bandwidth reordering is an NP-hard problem [17]. Thus, various heuristics for reordering have been proposed, including minimum-degree, Cuthill-McKee, RCM, Sloan, and nested dissection [11], [18]–[21]. Most follow-up work on bandwidth reduction focused on improving the reordering result, adjusting previous work or combining different methods, all working on a graph-based view of the matrix [18], [22]–[24]. Alternatively, a spectral reordering can be used [25]. However, studies have shown that hybrid approaches using RCM or Sloan achieve the best results [26]–[29]. In practice RCM is still the go-to method, due to its good reordering and simplicity.

Unsurprisingly, efficient single-threaded implementations can be found in many wide-spread software packages, including Matlab [30] and the HSL software library [31]. HSL subroutines of RCM are implemented in Fortran and optimizations focus on performance enhancing factors such as determining supervariables. The literature on parallel reordering can be split into two categories: Parallelization of reordering algorithms that are by design parallel, such as nested dissection [19], [32] or hierarchical community-based ordering [33] and parallelizing inherently serial algorithms such as RCM and Sloan [13]–[15]. Previous RCM parallelizations either parallelize each level of the BFS structure or use a speculative BFS before parallelizing only across the level. We discuss the details of these approaches in the next section. Nevertheless, all aforementioned approaches have only been demonstrated on multicore architectures and not on many-core processors. We are the first to show an algorithm that works on both and achieves significant speed-ups compared to previous work.

III. Background

Before detailing our approach, we recap the RCM algorithm and outline the considerations behind previous parallelizations.

a) Sequential RCM: The core RCM algorithm, shown in Alg. 1, closely follows a BFS while imposing sorting criteria during the discovery of nodes. We consider matrices in the compressed sparse rows (CSR) format: an offset array pointing to the start of each row and an index array capturing the destination node of each adjacency. We use a local array c to temporarily store nodes for sorting. For each node in the queue, all children are visited (ln 8-12), processing those that have not been visited before (ln 9). The valence is computed for the visited children and they are sorted accordingly (ln 13). The sorted children are then added to the permutation array o and to the FIFO-queue. o in reverse order yields the RCM output. A key observation for high performance is that sorting needs to happen only among the children of each node, as the nodes are already added to the FIFO-queue in correct order.

b) Leveled RCM: The simplest parallelization of RCM follows the level structure of the BFS, as shown in Alg. 2: Each level is explored in parallel; every node that has not been discovered previously is added to a temporary array c_i.

Additionally, for each discovered node, the parent with the lowest position in the permutation is determined using atomic operations (line 10) to ensure that sorting, which now happens across the entire level, correctly considers the parent location of first discovery. Sorting and writing the output can be carried out in parallel. After one iteration is complete, the next can start. This strategy also lends itself to the execution

Algorithm 1: Sequential RCM

```
1  Function RCM (r, i, s)                          // row_offs, indices, start_n
2      o ← ∅                                       // the permutation array
3      m ← s                                       // marked nodes
4      Q ← s                                       // initial node
5      while Q ≠ ∅ do                              // while nodes in the queue
6          p ←(∩) Q                                // take next node from queue
7          c ← ∅                                   // init local child array
8          for n = i[r[p] ··· r[p + 1]] do         // iterate children
9              if n ∉ m then
10                 m ←(∪) n                        // mark n visited
11                 v ← r[n + 1] − r[n]             // compute valence
12                 c ←(∪) {v → n}                  // store node locally
13         sort (c)                                // sort children by valence
14         o ←(∪) c                                // add c to output
15         Q ←(∪) c                                // add c to the queue
16     return reverse (o)
```

Algorithm 2: Leveled RCM

```
1  Function LeveledRCM (r, i, s)                   // row_offs, indices, start_n
2      m ← ∞ , l_1 ← s , i ← 1
3      while l_i ≠ ∅ do
4          s_i ← {∞}                               // lowest source tracker
5          c_i ← ∅
6          foreach p ∈ l_i in parallel do
7              for n = i[r[p] ··· r[p + 1]] do
8                  m_n ← atomicMin (m[n], i + 1)
9                  if m_n ≥ i + 1 then             // is on level
10                     atomicMin (s_i[n], o[p])
11                     if m_n > i + 1 then         // is first time
12                         v ← r[n + 1] − r[n]
13                         c_i ←(∪) {v → n}
14         parallelSort (c_i, s_i)                 // sort with source
15         parallelWrite (o, c_i)
16         l_{i+1} ← c_i
17         i ← i + 1
18         return reverse (o)
```

Algorithm 3: Unordered RCM

```
1  Function RCMUnordered (r, i, s)
2      l ← BFSUnordered (r, i, s)          // get all level infos
3      produce (s, 0)                      // insert starting node
4      foreach l_i ∈ l in parallel do
5          m_i ← ∅
6          while consume (l_i − 1) do // while incoming nodes
7              p ← wait (l_i − 1)          // wait for next node
8              c ← ∅
9              for n = i[r[p] ··· r[p + 1]] do
10                 if level (n) = l_i + 1 and n ∉ m_i then
11                     m_i ∪← n
12                     v ← r[n + 1] − r[n]
13                     c ∪← {v → n}
14             sort (c)
15             produce (c, l_i + 1)        // prod. for next thread
16             o ∪← c
17     return reverse (o)
```

Algorithm 4: Batch RCM basic

```
1  Function BatchRCMBasic (b)              // batch input_n
2      {c, p_c} ← discover (b)             // discover children
3      sort (c, p_c)
4      wait (Discovered)                   // wait for discovered
5      c ← rediscover (c)                  // recheck discovered
6      signal (Discovered)
7      wait (Counted)                      // wait for output position
8      signal (Counted)
9      o ← trimAppend (c)                  // trim removed children
10     addNewBatches (o, c)                // add batches to queue
11 Function discover (b)
12     c ← ∅
13     p_c ← ∅
14     foreach p ∈ b do
15         for n = i[r[p] ··· r[p + 1]] do
16             m_n ← atomicMin (m[n], b_i)
17             if m_n > b_i then
18                 v ← r[n + 1] − r[n]
19                 c ∪← {v → n}
20         p_c ∪← |c|                      // store parent offsets
21     return c, p_c
22 Function rediscover (c)
23     foreach n ∈ c do
24         if m[n] < b_i then
25             c ← c\n
26     return c
```

on many-core architectures, however, parallelism is limited to the number of nodes per level. Variants of the algorithms avoid sorting the complete array, but rather first count the children assigned to each parent, compute a prefix sum over the counts, in parallel write the children to the output array and finally sort them for each parent separately [13].

c) Unordered RCM: Another parallelization of RCM relies on a parallel BFS and parallelizes across different levels of the BFS [13], as shown in Alg. 3: Using speculative execution and repeated relaxation of the already visited nodes, BFS can be parallelized, but the execution speed strongly depends on the structure of the graph. After the BFS, threads operate in a producer-consumer fashion, with one thread per level. Whenever a thread processed all children of one node, it writes them to the output array and forwards them to the thread on the next level. Each thread can directly write to the output, as the count per level and thus the offsets in the output array are known from BFS. While this strategy is well suited for multicore architectures, it does not offer sufficient parallelism for typical many-core processors.

d) Peripheral Node Finding: The chosen start node influences the reorder quality of the RCM algorithm. Typically, a *pseudo-peripheral node* is chosen, *i.e.*, a node that is far away from other nodes but the distance is not necessarily the diameter of the graph. In this work we focus on the core RCM algorithm. Still, to compare to other methods that include start node finding, we use a naive pseudo-peripheral node finding approach: we start with a random node and execute multiple rounds of BFS. For each BFS round we use the node with the lowest valence on the last level of the previous BFS as a new starting node. If two successive BFS lead to the same number of levels we stop and use the last node as start node for RCM.

IV. Batch-based RCM

Our main motivation is to provide a parallel RCM approach that can run efficiently on many-core architectures like the GPU. Previous parallelization are suboptimal. Leveled RCM (Alg. 2) can only draw parallelism from a single level and requires multiple synchronization points. Unordered RCM (Alg. 3) requires an unordered BFS, which has not been shown for GPU, and only supports one worker per level in the second phase, not properly utilizing a many-core system. We propose batch-based RCM, following three goals: (1) Extract parallelism where possible to provide sufficient load for many-core architectures. (2) Execute speculatively to not stall cores, but avoid excessive unnecessary work. (3) Avoid additional memory to run on systems with little memory, like the GPU.

Our batch-based solution is outlined in a basic version in Alg. 4 and its full version in Alg. 5 and Fig. 1. The execution is split across batches of input data, which dynamically become available as the execution progresses through the graph. All batches are processed independently when possible, but wait at signaling points to ensure correctness.

A. Speculative Discovery and Sorting

Given the effectiveness of the straightforward single-threaded RCM algorithm (Alg. 1), we follow its implementation in every batch, but pair it with speculative execution in Alg. 4. The three main steps of our RCM are *discovering* (ln 2) children of all nodes in a batch, *sorting* (ln 3) them and *writing* the result to the output array (ln 9). Dependencies arise between batches at two points: *Discovery* is only correct if all previous nodes in the RCM order already discovered their children and marked them accordingly. *Writing* the output

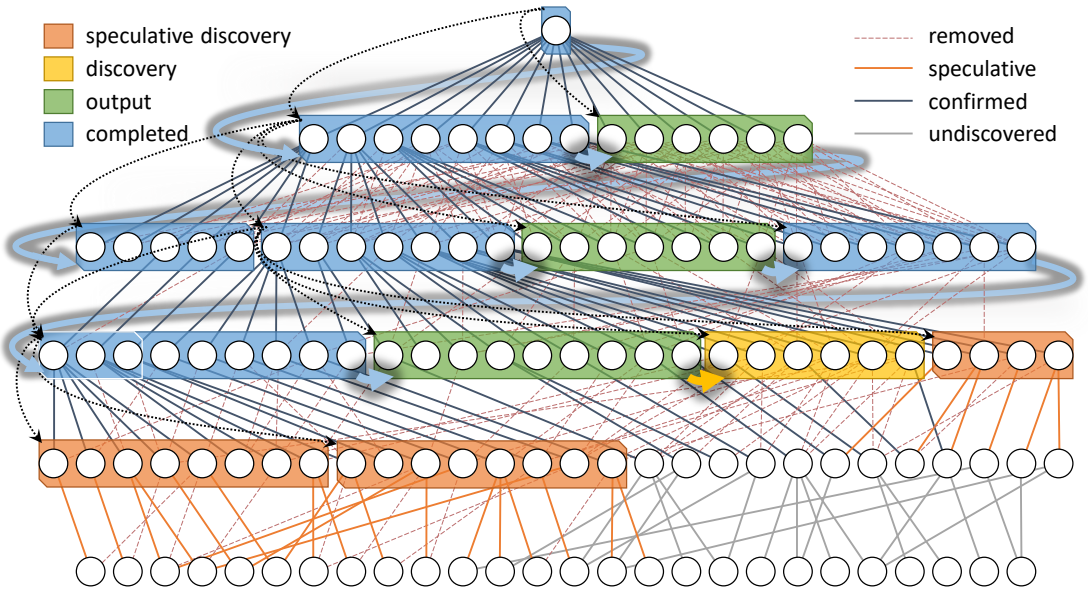


Fig. 1: Our batch-based RCM algorithm splits computation across batches that are processed in parallel (in this example all green, yellow and orange batches are concurrently active). Active batches may stem from multiple levels. Batches are in either of four states (speculative discovery, discovery, output or completed) and forward their states along a signal chain (highlighted arrows). Edges are discovered in a speculative manner (orange edges), which may require confirmation at a later point. Only those edges that contribute to the RCM order are kept (dark). To increase parallelism, we sort speculatively discovered batches.

is only possible when all previous batches know the exact number of outputs, *i.e.*, the number of owned child nodes.

a) Discovery: Waiting for all previous batches to complete *discovery* is overly restrictive, as only a single batch could run discovery at a time. To circumvent this issue, we use speculation: a batch that starts execution immediately performs discovery using atomicMin operations to mark nodes as discovered. The atomicMin ignores marks of previous batches and overwrites those of successors. For all discovered nodes (c), we immediately compute the valence and store it in an array in scratchpad memory for efficient cache access.

b) Sorting: In the next phase, we sort the speculatively discovered nodes. As sorting is typically one of the more costly steps, we also want to increase parallelism here rather than wait for speculative discovery to be confirmed. For efficient sorting, we keep information about which nodes belong to which parent (p_c) and sort them in successive sort operations.

c) Rediscovery: Before writing the sorted results to the permutation array, we have to remove those nodes from the temporary array that are discovered by a predecessor batch and thus we need to introduce a waiting point (ln 4). Afterwards, we *rediscover* nodes, *i.e.*, recheck all nodes in the array and remove those that have been discovered by a predecessor.

d) Output: To write the result, each batch has to wait for the output position, *i.e.*, the sum of nodes written by all predecessor batches. Finally, new batches for the output nodes are generated and added to a work queue. To ensure the correct execution order of batches and avoid algorithm stalls, the work queue follows the order of nodes in the output. After finishing a batch, a worker takes the next available batch from the queue.

e) Signal & Wait: The wait functionality is implemented using a simple signal per batch. We store batches according to their desired execution order in the queue and keep them available until the batch is completely processed. Therefore, a batch can signal its successor and store data alongside it.

B. Signaling and Speculation Avoidance

The previously outlined algorithm is not overly effective. We present an advanced version in Alg. 5, where we represent each batch's predecessor state either as Discovered, Counted or Completed and dynamically react to these states. A non-blocking `wait` allows us to switch to other jobs.

a) Early discovery: Even if all predecessor batches have completed discovery before a batch starts, our basic batch-based RCM version would run a rediscovery. To this end, we start by loading the signaled state before the first *discovery* attempt and thus avoid rediscovery later if possible. Right after *discovery*, a batch signals the next batch the discovery state, if itself has been signaled Discovered, as the entire chain of batches up to this point have completed discovery. This also happens if the batch requires rediscovery, *i.e.*, if it was only signaled during or after its initial *discovery*, because it still marked its nodes. This is a significant advancement from the basic approach, where signaling only happens after sorting and rediscovery. Before sorting, we perform *rediscovery* if the batch has been signaled during or after *discovery*. This lets us remove nodes before sorting and create a dense temporary array after sorting, which can be written without compaction.

b) Late discovery: Independently of whether the batch is signaled Discovered or not, we continue sorting and ensure

Algorithm 5: Batch RCM

```
1  Function BatchRCM (b)                                   // batch input_n
2      s_early ← signaled(b)                               // read early signal
3      {c, p_c} ← discover(b)                              // discover children
4      s_mid ← signaled(b)                                 // recheck signal
5      if s_early ≥ Discovered then
6          signal(Discovered)                              // all prev. discovered
7          f_this ← signalCount(s_mid, c, b)
8      else if s_mid ≥ Discovered then
9          s_early ← s_mid
10         signal(Discovered)                              // all prev. discovered
11         c ← rediscover(c)                               // recheck discovered
12         f_this ← signalCount(s_mid, c, b)
13     sort(c, p_c)
14     wait(Discovered)                                    // wait for discovered
15     if s_mid < Discovered then
16         signal(Discovered)
17         c ← rediscover(c)
18         s_mid ← signaled(b)
19         f_this ← signalCount(s_mid, c, b)
20     wait(Counted)                                       // wait for output position
21     if s_mid < Counted then
22         s_mid ← signaled(b)
23         f_this ← signalCount(s_mid, c, b)
24     if s_early < Discovered then
25         o ← trimAppend(c)                               // trim removed children
26     else
27         o ⟵∪ c
28     wait(Completed)                                     // ensure overhang chains
29     if Overhang(f_this) then
30         signal(Completed)
31     addNewBatches(f_this)                               // add batches to queue
32 Function signalCount(s, c, b)
33     if s ≥ Counted then                                 // need prev. output position
           /* combine output with previous batch?                          */
34         f_prev ← Overhang(b);
35         f_this ← countBatches(c, f_prev)
36         if Overhang(f_this) then                        // combine /w next?
37             signal(Counted)
38         else
39             signal(Completed)                           // no need to wait
40         return f_this
```

that the batch has been signaled Discovered afterwards, before continuing with the rediscovery (in case it is still needed). If we rediscover after sorting, we only mark the removed nodes in the array and compact during output (ln 25), potentially saving many memory operations.

The second signal, Counted, can be sent earlier than in our basic version as well. As soon as a batch is signaled Counted and it has run a successful *discovery*, it can already signal the next batch, as the exact number of outputs is already known at this point. Thus, the output positions can often be forwarded before sorting to create additional execution independence.

C. Batch Generation and Combination

Batches should ideally comprise a constant number of nodes to provide sufficient workload and parallelism for our many-core version. Furthermore, the number of discovered child nodes should not exceed the temporary memory requirement for c. If a graph is highly connected, we want to reduce the batch size to reduce the number of potential temporary elements. To build batches of child nodes, we rely on the information from discovery: as we need the valency for sorting, we know the exact number of temporary elements required when combining nodes to batches in the final step (ln 31). We already compute the batch membership while writing the outputs (ln 24-27 - not shown here for simplicity), introducing batch boundaries when the number of output nodes reaches the batch size or the sum of all valencies to this point overflows temporary memory bounds. If a single node would overflow temporary memory bounds, we create a single node batch. This requires an extension of scratchpad memory and potentially reduce cache efficiency. Splitting a node among threads would be more complicated as there is no simple way to distribute nodes before sorting. In our experiments on the CPU, the increased scratchpad memory requirements did not lead to a measurable performance hit.

a) Early Generation: The previous outline works but synchronizes batch generation from one batch to the next, as the number of batches is only known when scanning through the ordered output. However, we can already estimate the number of required batches after a successful discovery, as we already know the exact number of output nodes as well as the sum of their valences. Only their order and thus the possible combination is unknown. As we already accept scratchpad memory overload in some cases, we assume that we can optimally pack the nodes into batches. When signaling Counted to the next batch, we include the number of output batches that will be generated and thus the batch offsets in the queue. We ensure that we balance any surplus on temporary elements among the generated batches: while the sum of valences of remaining nodes divided by the to-be-generated batches is above the valence sum of the current batch, we add further nodes. An alternative is to overestimate the number of generated batches and add empty batches if they cannot be filled, which we use for our GPU implementation.

b) Batch Combination and Load Balancing: In parallel algorithms it is important to distribute the amount of available work approximately equal among the available workers to achieve high throughput. This is particularly challenging when work is generated dynamically during execution as in RCM, where the number of outputs per batch can vary drastically. Especially when reaching the end of reordering or if the front of discovery narrows, the number of child nodes per batch might fall significantly below the number of input nodes and leave a large part of the generated batch unoccupied. As starting batches and managing their data induces an overhead, very small batches can significantly reduce performance. To tackle this issue, we support *overhangs* to the next batch, *i.e.*, output nodes can be forwarded, such that the next batch adds the overhang to its first generated batch, thereby balancing the number of nodes per batch.

To this end, we again rely on the node count and the

sum of valences after a successful discovery. If the number of output nodes and the amount of temporary elements is below half of what a batch can hold, we forward the elements to the next batch. If a batch forwards nodes, it only sets the Counted state and informs the next batch how many nodes and temporary elements are being forwarded. The next batch includes them when creating its first batch. As batches only reference ranges in the output array, Completed is signaled after writing the output. Therefore, a not yet written overhang is never referenced. Note that chains of overhangs can be generated, and thus, Completed is only signaled after also being received. Also, nodes on different levels (of the corresponding BFS) can be combined into a single batch. Actually, our approach does not contain the notion of levels.

D. Multi-batch Execution and Termination

To further reduce stalls we avoid explicit waits for signals. Instead of stalling the execution when waiting for Discovered, Counted or Completed, we check the current signal and may draw a new element from the work queue. We switch back to the previous batch when reaching a wait point with the new batch. While the number of concurrent batches can be chosen arbitrarily, the required additional temporary memory increases with every additional batch. Later batches cannot overtake earlier ones for Discovered, however it is possible to finish a batch while a previous is still waiting for a Completed signal, as `signalCount` may send Completed early on. Thus, even within one worker, batches can finish out of order.

Interestingly, the batch-based RCM implementation can terminate while batches are still in the queue. If the number of written output nodes is equal to the number of nodes, the permutation is done. Nodes that have been visited for the first time may still reside in the queue, as the fact that they do not have un-visited children was not known during batch generation. Thus, we include a flag in our batch queue to allow for early termination. Whenever a worker tries to dequeue a new batch, this flag is checked first and if it is set, the worker simply terminates execution as if all batches were finished. Note that workers will still finish batches that they dequeued before the early termination flag was set and will only exit when they try to get a new one from the queue. Nevertheless, these batches do not generate any output, as all their child nodes were already visited by earlier batches. As we draw batches in order from the queue, early termination cannot break the signal chain.

V. Many-core Implementation

Our algorithm works well for multicore processors with single threads as workers as well as for many-core processors, such as the GPU, where we use a cooperative thread array (thread-block) as worker. Translating the batch-based algorithm to the GPU is straightforward as long as computation can stay in scratchpad memory (shared memory in CUDA terms).

We use a ring buffer as a work queue, data and signals are set using explicit non-locally cached writes. We use busy waiting with sleep operations (supported on Volta+) to back off. Processing of a batch happens exclusively in scratchpad memory. General batch information, *i.e.*, loading batch start and end pointers, querying the signal state, as well as signaling the next batch, are handled by the first thread of the block.

A. Parallel Batch Processing

To perform discovery in parallel, we first query the number of child nodes for each parent (one thread per parent), store the offsets to the children in scratchpad memory and compute the maximum child count across the entire thread-block. Based on this count, we adjust the number of threads per parent for discovery. We use the last power of two smaller than the maximum child count for each parent. In this way, we achieve coalesced memory access for batches with large child counts and reduce the number of stalled threads for batches with nodes that have few children. As threads query the children, they use atomicMin for marking and then add the found children to a scratchpad array, using atomicAdd to reserve a unique spot in the array. Rediscovery is straightforward, as we simply recheck the marks of all currently stored nodes.

For sorting, we use CUB radix sort with a local parent id in conjunction with the child valence as sort key. This also generates the correct order across children of different parents, which was destroyed by the atomicAdd. As the number of elements to sort varies, we switch between different sort implementations with one to eight elements per thread.

Before writing nodes to the output, we perform a prefix sum to determine the correct output locations as nodes may have been removed through rediscovery. We use a prefix sum over the valences to determine the temporary memory requirements when combining nodes to batches. We use a single prefix sum for both computations, using subsets of bits for the offset and the sum. Depending on the number of threads, the maximum number of temporary elements and the maximum valence, we either use 32 or 64 bit values. To generate new batches, a single warp moves across the prefix sum result and determines batch boundaries, *i.e.*, where an overflow of the batch size or temporary memory requirements is reached.

B. Memory Limits

Overflowing scratchpad memory is a major issue on the GPU, as the scratchpad memory allocation cannot be adjusted dynamically and we want to avoid spilling to global memory. First, we avoid running out of pre-allocated batches by adding a safety margin. We count the number of child nodes that actually reach beyond the available temporary memory and clamp their valence sum contribution to exactly the temporary memory requirements, as they will receive their own batch anyway. From the resulting sum, we compute the ideal number of batches and multiply it by two: In the worst case, a batch is half filled and the next node just does not fit and creates another half filled batch. In these cases, we simply add empty batches to the queue—in the worst case 50% of batches are empty. In practice we saw performance fluctuations of 1-3% when using a per-matrix tuned multiplication factor, indicating that empty batches are discarded efficiently.

The last issue are single nodes exceeding scratchpad memory. Often, a number of children has already been discovered by predecessor batches and we can operate in scratchpad memory, although the number of children would overflow the available memory. To this end, we run a first discovery step, count the number of nodes that should be handled by the batch and create a *valence histogram*. If we can directly work on the nodes, we follow the standard path. Otherwise, we select as many histogram bins as possible (starting from the smallest) to fill up scratchpad memory, chunking the execution. This is possible, because we guarantee that large batches only contain a single parent node (compare Sec. IV-C).

We limit the histogram to min and max valence, found during discovery, to make more efficient use of our 128 histogram bins. Still, nodes of different valence can end up in the same bin and a single bin may contain more nodes than can be handled at once. Therefore, we follow these strategies: Valence distributions are often skewed, *i.e.*, many nodes of a small valence range and a few larger or smaller valences. Thus, we compute the mean valence and linearly remap valences such that the mean is in the middle of the histogram. While this solved more than 95% of all issues in our test set, it might still fail. In this case, we create histograms hierarchically, computing a new histogram for a single bin. At the lowest level, if reached, bins only contain a single valence and we can directly copy all nodes in a bin from the input matrix to the permutation array without going through scratchpad memory.

VI. Evaluation

We ran our experiments on an AMD Ryzen 3900X (12 cores, 64MB cache), 32GB RAM and an NVIDIA TITAN V (12GB VRAM) with Linux, gcc 10.2 and CUDA 11.0.3. Our test-set was randomly selected from the SuiteSparse Matrix collection [34] to include symmetric matrices from various application fields with largely different number of non-zero entries and sparsity patterns. Our framework is publicly available at https://github.com/GPUPeople/ParallelBatchRCM.

A. Approaches

Unfortunately, there is little RCM code publicly available and we could not find a single GPU-based implementation. Rodrigues et al. provided us with their *Reorderlib* [15], which offers multiple variants to perform RCM, including a leveled RCM and an unordered RCM version. We use the latter in this evaluation as it performed significantly better. According to the authors, the original implementation of the unordered RCM [13] is not available. We also include HSL [31] as an alternative baseline. NVIDIA provides RCM via *cuSolver* [35], which is, however, completely CPU-based and single threaded. We also compare the MATLAB's RCM implementation.

We use two baselines: a serial CPU-only RCM (*CPU-RCM*, Alg. 1) and a leveled RCM on the GPU, which uses CUB for sorting (*GPU-RCM*, Alg. 2). Furthermore, we show results of a simplified version (*CPU-BATCH-BASIC*, Alg. 4). Lastly, we provide data for our fully optimized parallel versions on the CPU (*CPU-BATCH*) and the GPU (*GPU-BATCH*).

B. Core RCM Performance

Data about our test-set as well as performance of the core RCM implementations is shown in Tab. I. Our single-threaded *CPU-RCM* outperforms HSL significantly. We attribute this to three facts: First, we may benefit from more efficient sorting algorithms, now present in the C++ STL. Second, modern C++ compilers may be able to optimize better for current hardware. Third, using scratchpad memory to keep temporary data may increase cache hit rates. Nevertheless, we use HSL as a baseline to allow for better comparison to previous work.

The data indicate that our versions provide the fastest RCM across the tested matrices. As expected, there is a clear difference between smaller and larger matrices, see Fig. 2: For small matrices with a limited BFS front width, *CPU-RCM* clearly performs best, as parallelization overhead does not amortize. For medium sized matrices, *CPU-BATCH* and *GPU-BATCH* start to outperform *CPU-RCM* by $2\times$ - $4\times$. *GPU-RCM* as well as *Reorderlib* are already far off. For large matrices, which provide a higher parallelism, our parallel implementations achieve speedups of $4\times$ or more compared to *CPU-RCM*, which in turn is about $5.8\times$ times faster than HSL on average over our test-set. Interestingly, *Reorderlib* always falls short of *CPU-RCM*. *GPU-BATCH* and *CPU-BATCH* are quite similar in performance, indicating that *GPU-BATCH* eliminates the need to move matrices from the GPU back to the CPU for reordering. At the same time, *GPU-RCM*, especially for matrices with limited parallelism, is a lot slower than *GPU-BATCH*. Note that, although the main difference from *CPU-BATCH-BASIC* to *CPU-BATCH* is the improved signaling mechanism, this results in an average speed-up of $1.14\times$ and up to $1.53\times$ for large matrices.

One very interesting observation is that we achieve a superlinear speedup for *mycielskian18* compared to *CPU-RCM*. Deeper analysis reveals that this performance advantage of *CPU-BATCH* and *GPU-BATCH* comes from our early stopping detection, while batches are still in the queue. Due to the special structure of some matrices, many times more nodes are put into the queue than actually need to be processed as shown in Fig. 3. The difference between *Generated* and *Dequeued* batches is due to early termination (Sec. IV-D), as batches are left in the queue when the permutation is already complete. While in most cases more than 99% of the generated batches are also dequeued, there are matrices where early termination can reduce the number of executed batches significantly, *e.g.*, to 16% on *gupta3* or even to $< 1\%$ on *mycielskian18*. The reduction in *Executed* compared to *Dequeued* is due to empty batches, see Sec. V-B, which results in approximately 36% of all dequeued batches being discarded.

While we could not directly compare against the linear algebra-based RCM version [14], we look at the overall runtimes for *nlpkkt240*, which they also include in their test-set. *CPU-RCM* running on a single core on our machine requires $4.6\,\mathrm{s}$, *CPU-BATCH* $0.9\,\mathrm{s}$ with 24 threads. The linear algebra-based RCM [14] needs $3.2\,\mathrm{s}$ on 54 cores and $1.2\,\mathrm{s}$ on 4056 cores, clearly showing the advantage of our implementation.

Name	n_r/n_c	NNZ	max valence	avg BFS front	init. BW	reord. BW	HSL	*Reorder-lib*	tc	*CPU-RCM*	*CPU-B.-BASIC*	tc	*CPU-BATCH*	tc	*GPU-RCM*	*GPU-BATCH*
bcspwr10	5.3k	22k	14	186	5189	285	1.28	1.98	1	**0.26**	0.33	1	0.33	1	3.81	1.09
bodyy4	17.5k	122k	9	170	16818	248	1.49	2.24	1	**0.29**	0.78	1	0.76	1	10.74	2.89
benzene	8.2k	243k	37	1303	2898	1905	2.11	2.17	1	**0.30**	0.56	5	0.64	3	4.55	0.43
ncvxqp3	75.0k	500k	15	1054	69996	14154	11.34	11.44	1	2.38	2.36	10	**2.33**	8	7.56	0.91
ecology1	1.0M	5.0M	5	667	1000	1000	154.95	190.84	1	**26.81**	31.13	1	40.61	1	541.21	57.21
gupta3	16.8k	9.3M	14672	7939	16744	15584	59.00	21.73	3	5.64	**1.18**	6	1.67	2	33.10	1.16
SiO2	155.3k	11.3M	2749	14340	55068	20209	104.41	75.64	8	16.30	12.09	7	**11.10**	8	22.99	9.71
CurlCurl_3	1.2M	13.5M	54	12502	26759	20045	179.05	271.25	2	44.74	40.79	9	**31.41**	13	78.98	17.94
nd12k	36.0k	14.2M	13	4596	34517	6341	100.52	26.73	8	12.47	9.14	6	**8.18**	7	22.90	15.49
Si41Ge41H72	185.6k	15.0M	519	18438	31518	26518	144.77	72.66	6	22.82	16.69	7	**15.30**	7	28.04	16.92
great-britain_osm	7.7M	16.3M	662	2334	7693184	4677	1274.45	—	—	291.08	326.02	1	**270.17**	3	3875.03	223.12
human_gene2	14.3k	18.1M	8	6414	14257	12037	150.54	56.28	2	11.65	9.29	6	**8.69**	6	29.49	20.63
Ga41As41H72	268.1k	18.5M	7229	23496	40195	33379	189.44	97.18	5	30.06	21.93	10	**19.36**	12	34.00	20.63
bundle_adj	513.4k	20.2M	702	5094	510044	20738	87.54	144.39	2	29.76	**22.41**	8	27.17	8	341.25	16.49
nd24k	72.0k	28.7M	12588	8156	68114	11291	200.89	46.14	12	23.77	16.41	7	**15.59**	8	36.16	31.24
coPapersDBLP	540.5k	30.5M	520	159025	539587	254848	392.93	—	—	65.34	27.32	12	**26.42**	12	47.15	31.60
Emilia_923	923.1k	41.0M	3299	12280	17279	16883	194.62	213.01	23	47.06	45.44	10	**30.71**	13	89.60	49.25
delaunay_n23	8.4M	50.3M	57	10403	8382693	16777	1557.97	—	—	271.13	153.71	15	**132.41**	10	828.79	79.03
hugebubbles-00020	21.2M	63.6M	28	3211	21188550	4575	9377.19	—	—	1598.78	1241.05	12	**905.41**	6	8490.28	248.43
audikw_1	943.7k	77.7M	3	16586	925946	34400	377.90	244.46	14	118.25	58.99	14	**49.58**	12	139.62	85.55
nlpkkt120	3.5M	96.8M	345	51894	1814521	86876	1411.13	837.78	13	383.20	203.19	17	**132.63**	18	200.00	114.05
Flan_1565	1.6M	117.4M	28	11159	20702	20849	510.34	339.62	23	168.81	89.83	13	**68.62**	14	223.86	134.16
nlpkkt160	8.3M	229.5M	81	92232	4249761	154236	3675.97	1912.27	24	1166.98	436.58	22	**286.23**	18	442.00	268.57
mycielskian18	196.6k	300.9M	28	98300	196590	196589	2770.78	—	—	213.77	8.73	20	**8.58**	17	468.59	14.02
nlpkkt200	16.2M	448.2M	98303	144089	8240201	240796	7335.28	3402.59	24	2547.49	784.54	24	**540.97**	24	814.90	520.01
nlpkkt240	28.0M	774.5M	28	207467	14169841	346556	13218.79	5644.68	23	4574.78	1283.31	24	**938.80**	24	1534.99	900.77

TABLE I: Number of rows/columns (n_r/n_c), number of non-zeros (NNZ), maximum node valence, the average width of the BFS front and initial as well as reordered bandwidth for each matrix. Best timing of the core RCM implementation (from $1-24$ threads) for each approach in ms as well as the thread-count (tc) that achieved that timing. Best CPU time bold. Best GPU time always *GPU-BATCH*. *MATLAB* and *cuSolver* excluded, as they perform start node finding automatically.

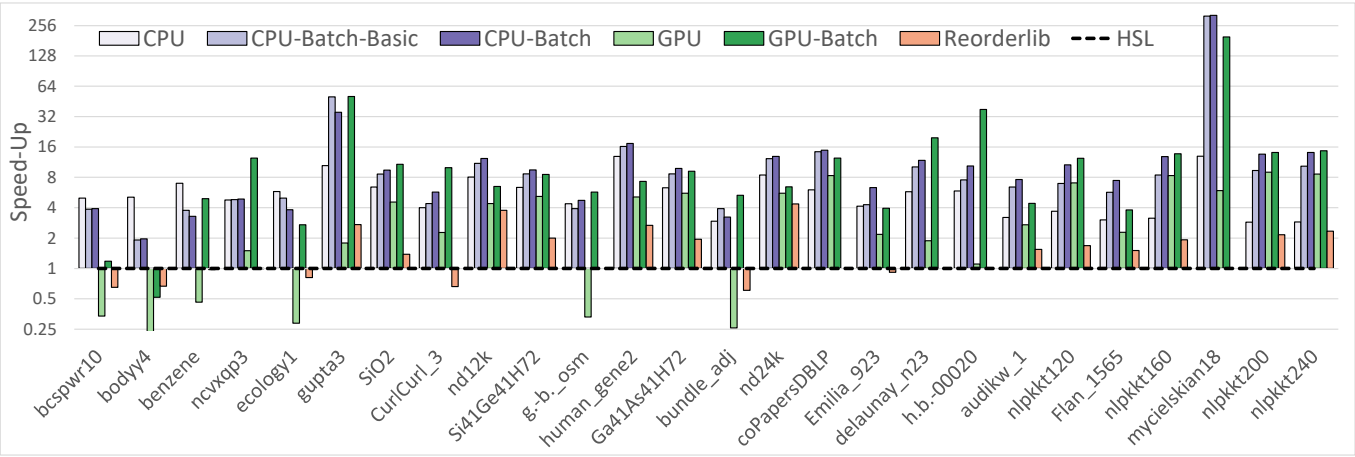


Fig. 2: Speed-up of each approach compared to HSL in log-scale.

C. Overall Performance

While our work focuses on the core RCM, we rely on a simple peripheral node finding strategy, as described in Sec. III-0d, for comparisons of our CPU versions with Matlab and cuSolver, as they do not separate the core RCM implementation from initial node finding. For our GPU versions, we use our complete RCM implementation for node finding, but disable sorting, which natively leads to a parallel BFS. However, we did not apply any optimization, thus all peripheral node finding results should only be considered a reference point and focus is put on the RCM performance.

As can be seen in Fig. 4, even with inefficient peripheral node finding, our approaches achieve the best performance, outperforming cuSolver by multiple orders of magnitude and still outperforming Matlab with a significant margin; even *CPU-RCM* is faster than Matlab. Also note that the core RCM is always significantly faster than our naive peripheral node finding, showing that our optimized versions clearly work well. Especially when looking at *GPU-BATCH*, the core RCM step only makes up a fraction of the overall runtime.

Note that transferring a matrix from the GPU to the CPU for reordering and back would incur additional overhead. To

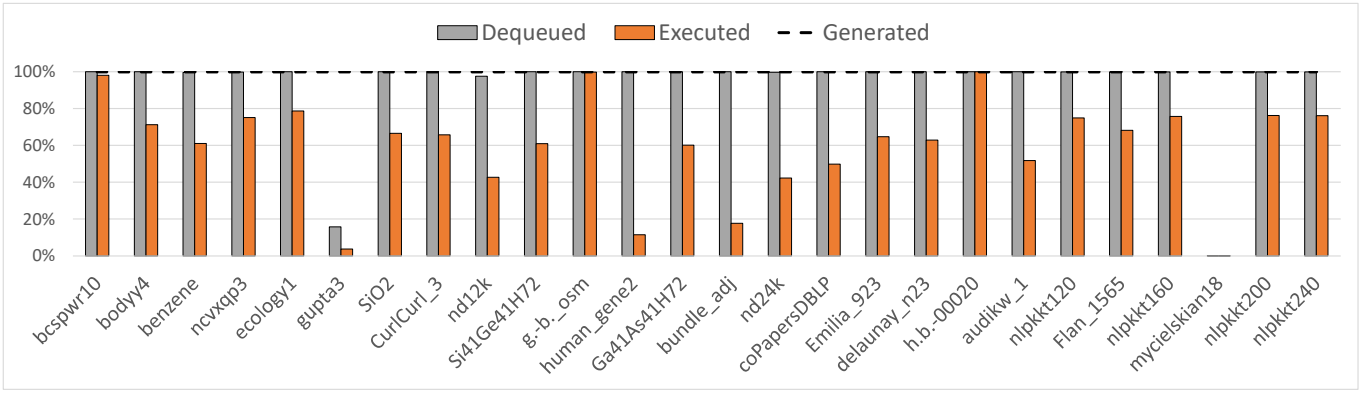


Fig. 3: Early termination enables us to discard all batches in the queue as soon as the permutation is complete. Furthermore, batches, which where generated and enqueued to allow early progression for the next batch but where not filled due to over-estimation of the number of output nodes, do not have to be executed and can be discarded. Numbers given for *GPU-BATCH*.

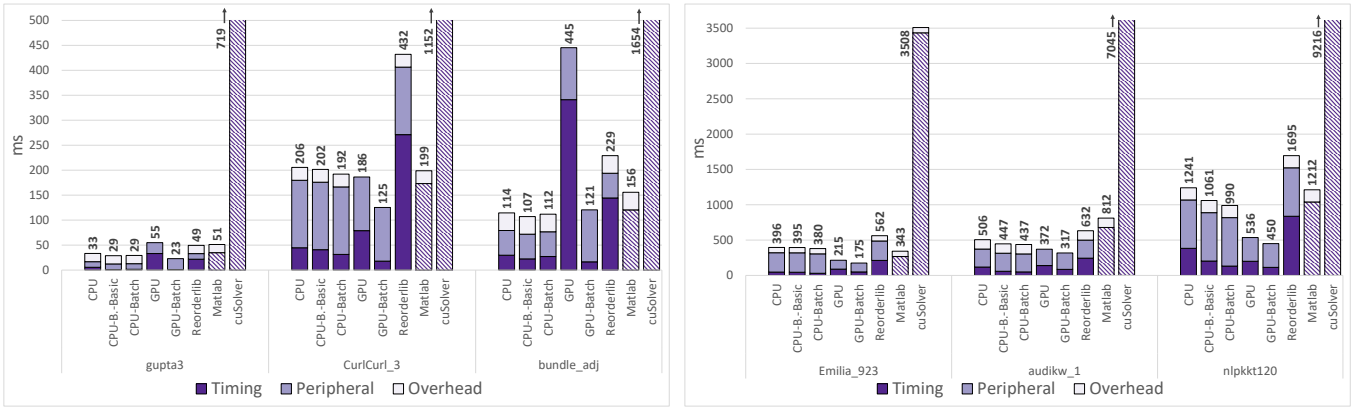


Fig. 4: The core RCM timing of each approach, see also Tab. I, in dark violet (*cuSolver* & *MATLAB* also include pseudo-peripheral node finding), pseudo-peripheral node finding in lilac and potential overhead of GPU/CPU interaction in light violet. For small matrices, *CPU-RCM*, *CPU-BATCH* and *GPU-BATCH* are very close in performance, closely followed by *Reorderlib* and *MATLAB*, while *cuSolver* is orders of magnitudes slower. *GPU-RCM* cannot exploit the available parallelism well. For large matrices, *GPU-RCM* and *GPU-BATCH* clearly highlight their advantages. *CPU-BATCH* outperforms *CPU-RCM* in all cases, with *CPU-BATCH-BASIC* being in between. *MATLAB* comes next, followed by *Reorderlib* and a distant last is *cuSolver*.

judge whether it makes sense for any matrix to be transferred for reordering, we added the transfer overhead for the CPU approaches. Only for the smallest matrices, the overhead of transfer amortizes (and only when using our *CPU-RCM*), whereas the gains are negligible.

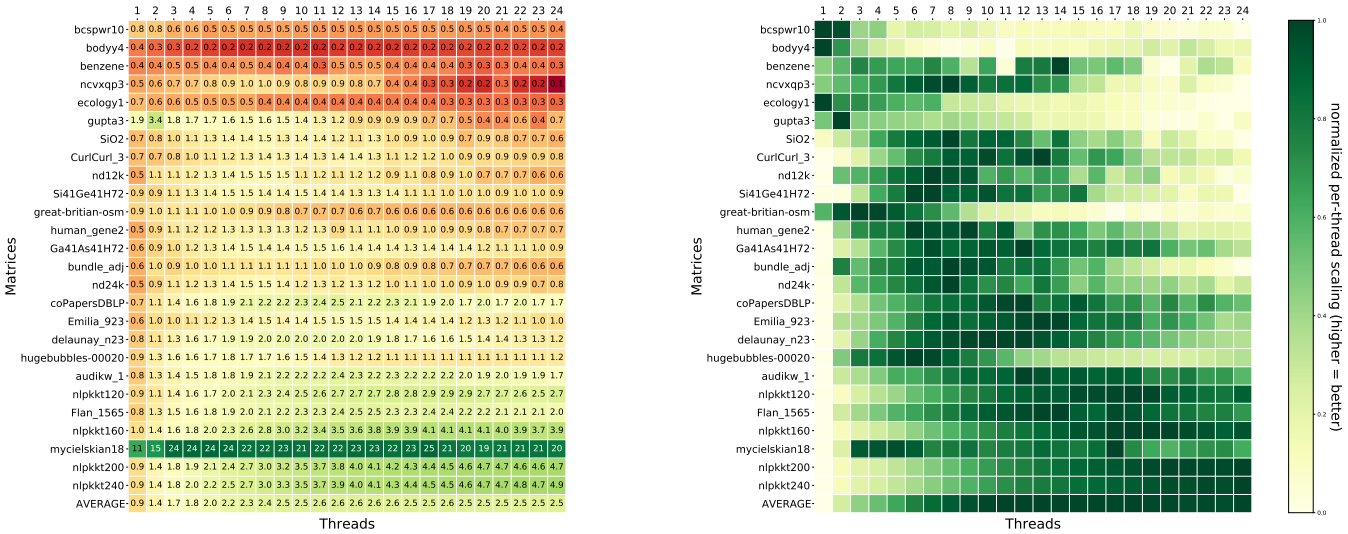
D. Performance Scaling

To analyze the scaling, we look at *CPU-BATCH* with increasing thread counts across all matrices in Fig. 5a and Fig. 5b, where we plot the absolute speed-up against the *CPU-RCM* and a normalized speed-up per matrix respectively.

Overhead is clearly a factor in running a parallel algorithm, as the first column in Fig. 5a shows: For the smallest matrices ($\leq 5M$ NNZ), performance drops by up to 60% by simply managing batches and using atomic operations for discovery. For larger matrices, adding a few threads amortizes the overhead, leading to speed-ups of about $1.5\times$ for small matrices ($5M <$ NNZ $\leq 30M$), $2.1\times$ for medium-sized ($30M <$ NNZ $\leq 100M$) matrices, and up to $4.9\times$ for large matrices (NNZ $> 100M$). Again, *mycielskian18* is a clear outlier due to the effectivity of early stopping in this matrix. For *great-britian-osm* and *hugebubbles-00020*, the parallel approaches do not scale well. This is likely due to a narrower average BFS front through the matrix (see Tab. I), *i.e.*, less available parallelism.

As shown in Fig. 5b, increasing the thread count may also reduce performance, leading to a somewhat diagonal pattern in the plot. Performing heavy parallel speculative discovery may lead to wasted sorting efforts and increased memory congestion. While the plot also indicates that our approach scales well for larger matrices, it may be reasonable to limit parallelism and potentially stall threads if many rediscoveries fail, *i.e.*, if speculation is generating parallelism that is not backed by the matrix structure.

Fig. 6 shows the percentage of the total number of cycles



(a) Heatmap depicting speed-up of *CPU-BATCH* over *CPU-RCM* for different thread counts on our test-set.

(b) Heatmap depicting a per-matrix, per-thread scaling, normalized by minimum and maximum speed-up per matrix.

Fig. 5: Scaling heatmaps for our *CPU-BATCH* version compared to *CPU-RCM*. Matrices are sorted by increasing NNZ elements from top to bottom. The absolute heatmap on the left clearly shows that the parallel version profits from larger input sizes, while the overhead of parallelization can not be amortized for very small inputs. The relative heatmap on the right shows, that *CPU-BATCH* can clearly take advantage of an increased thread count, if enough parallel workload is available.

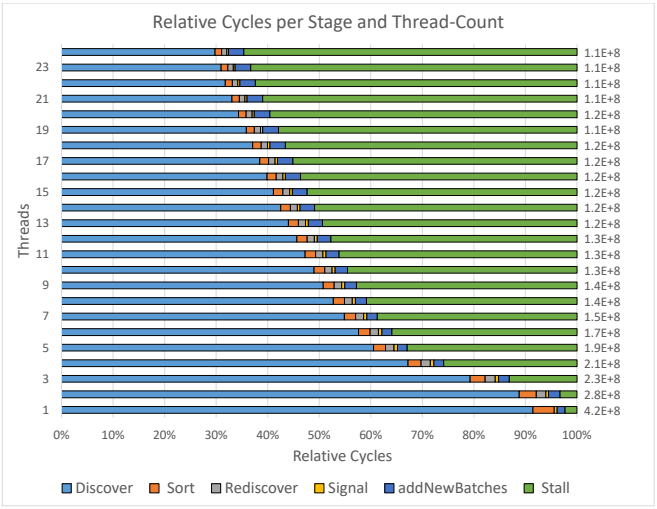


Fig. 6: Relative number of cycles spent in different stages of *CPU-BATCH*, averaged over the whole test-set for different thread counts. Average total cycles per thread over the complete test-set given on the right for each thread count.

spent in each stage of *CPU-BATCH* averaged over the full test-set for different thread counts. For small thread counts, the majority of cycles, *e.g.*, 88.2% with two threads, is spent on *Discover* due to the atomics used to mark the nodes. While the relative share of compute cycles for *Discover* and *Sort* reduces with increasing thread counts, their absolute numbers summed over all threads increases: *Discover* with atomics comes with interference between threads; *Sort* becomes more costly as the amount of data to be sorted increases due to speculation. In comparison to *Discover*, *Rediscover* always requires a very small share of the cycles (1.3% on average over all thread counts). While the former needs to perform atomic operations to mark nodes as discovered by the batch, the latter only needs to check which child nodes were discovered by a predecessor and mark them in local memory for compaction when writing the output. The time spent on *Signal* is virtually negligible in all cases. *addNewBatches* includes both writing the output nodes and adding them to the queue. The relative increase in cycles with increasing thread counts can be attributed to inter thread communication when adding queue elements. Clearly, using a larger number of threads increases waiting times for our test set: *Stall* captures queue fetch times including waiting for new elements to be added to the queue. With 12 threads, already almost half the cycles (48%) are spent idling and even 65% with 24 threads. Again, note that this is an average over all matrices, including small matrices and matrices with a narrow front, *i.e.*, matrices with little inherent parallelism.

E. Limitations

While our approaches performs very well throughout the test-set, there are some exceptions as discussed above. Clearly, for very small matrices an efficient serial implementation like our *CPU-RCM* will perform best. However, there are also larger matrices for which *CPU-BATCH* and *GPU-BATCH* do not scale well. Unfortunately, that means that the NNZ of a matrix alone are not a good predictor of the available parallelism. The average BFS front gives a better indication about the available parallelism and about how well our parallel versions perform. While it is costly to compute in general, it

could be determined alongside a peripheral node and be readily available before starting the core RCM algorithm. Other than the available parallelism, which is related to the BFS front, we could not find any properties of the matrix structure that may determine the run time of our algorithm.

VII. Conclusion

In this work we proposed a new parallelization of RCM reordering, which scales well with the available parallelism in the data and the number of available threads. Our approach dynamically adapts to the varying parallelism at runtime and redistributes work along a signal chain in order to balance the load between threads. Confirmation messages along the chain enable parallel progress as soon as requirements are fulfilled. Pressure on the memory bus is reduced by holding data locally until it is confirmed to be ready to be written to memory.

Our work executes on multicore and many-core processors alike, thus, for the first time, enables us to reorder matrices directly on the device they arise from, eliminating data transfer overheads and potential idle times. We present the first ever GPU parallelization of RCM, making it possible to integrate reordering mechanisms into operation sequences on the GPU.

Similar strategies as we use for RCM are viable for pseudo-peripheral node finding. Directly applying our RCM approach as BFS replacement already achieved good performance. Still, an in-depth treatment of pseudo-peripheral node finding is needed to not bottleneck our RCM implementation.

While our approach is currently limited to a single multicore or many-core device, its intrinsic properties lend themselves to multi-device and multi-node extensions, transmitting signals across devices/nodes. We hope that this work will be the first step of RCM into modern supercomputing environments.

References

[1] K. H. Huebner, D. L. Dewhirst, D. E. Smith, and T. G. Byrom, *The finite element method for engineers*, 2001.

[2] T. Davis, *Direct Methods for Sparse Linear Systems*, ser. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, 2006.

[3] H. M. Markowitz, "The elimination form of the inverse and its application to linear programming," *Manage. Sci.*, vol. 3, no. 3, pp. 255–269, Apr. 1957.

[4] T. A. Davis and E. P. Natarajan, *Sparse Matrix Methods for Circuit Simulation Problems*. Springer Berlin Heidelberg, 2012.

[5] R. Zayer, M. Steinberger, and H.-P. Seidel, "A GPU-adapted structure for unstructured grids," *Computer Graphics Forum*, vol. 36, no. 2, pp. 495–507, 2017.

[6] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–9.

[7] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Advances in neural information processing systems*, 2014, pp. 2177–2185.

[8] W. Gerstner and W. M. Kistler, *Spiking neuron models: Single neurons, populations, plasticity*, 2002.

[9] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, New York, NY, USA, 2009, pp. 18:1–18:11.

[10] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, "Speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20, New York, NY, USA, 2020, p. 362–375.

[11] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69, New York, NY, USA, 1969, pp. 157–172.

[12] J. A. George, "Computer implementation of the finite element method," Ph.D. dissertation, Stanford, CA, USA, 1971.

[13] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali, "Parallelization of reordering algorithms for bandwidth and wavefront reduction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, Piscataway, NJ, USA, 2014, pp. 921–932.

[14] A. Azad, M. Jacquelin, A. Buluç, and E. G. Ng, "The reverse cuthill-mckee algorithm in distributed-memory," in *2017 IEEE IPDPS*, 2017, pp. 22–31.

[15] T. N. Rodrigues, M. C. S. Boeres, and L. Catabriga, "A non-speculative parallelization of reverse cuthill-mckee algorithm for sparse matrices reordering," in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sep. 2017, pp. 527–536.

[16] D. Mlakar, M. Winter, P. Stadlbauer, H.-P. Seidel, M. Steinberger, and R. Zayer, "Subdivision-specialized linear algebra kernels for static and dynamic mesh connectivity on the GPU," *Computer Graphics Forum*, vol. 39, no. 2, pp. 335–349, 2020.

[17] C. H. Papadimitriou, "The np-completeness of the bandwidth minimization problem," *Computing*, vol. 16, no. 3, pp. 263–270, 1976.

[18] A. George, M. T. Heath, J. Liu, and E. Ng, "Solution of sparse positive definite systems on a shared-memory multiprocessor," *International journal of parallel programming*, vol. 15, no. 4, pp. 309–325, 1986.

[19] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 71–95, 1998.

[20] R. J. Lipton, D. J. Rose, and R. E. Tarjan, "Generalized nested dissection," *SIAM journal on numerical analysis*, vol. 16, no. 2, pp. 346–358, 1979.

[21] S. Sloan, "An algorithm for profile and wavefront reduction of sparse matrices," *International Journal for Numerical Methods in Engineering*, vol. 23, no. 2, pp. 239–251, 1986.

[22] N. E. Gibbs, W. G. Poole, Jr, and P. K. Stockmeyer, "An algorithm for reducing the bandwidth and profile of a sparse matrix," *SIAM Journal on Numerical Analysis*, vol. 13, no. 2, pp. 236–250, 1976.

[23] J. G. Lewis, "Implementation of the gibbs-poole-stockmeyer and gibbs-king algorithms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 8, no. 2, pp. 180–189, 1982.

[24] W.-H. Liu and A. H. Sherman, "Comparative analysis of the cuthill–mckee and the reverse cuthill–mckee ordering algorithms for sparse matrices," *SIAM Journal on Numerical Analysis*, vol. 13, no. 2, pp. 198–213, 1976.

[25] S. T. Barnard, A. Pothen, and H. Simon, "A spectral algorithm for envelope reduction of sparse matrices," *Numerical linear algebra with applications*, vol. 2, no. 4, pp. 317–334, 1995.

[26] W. W. Hager, "Minimizing the profile of a symmetric matrix," *SIAM Journal on Scientific Computing*, vol. 23, no. 5, pp. 1799–1816, 2002.

[27] Y. Hu and J. A. Scott, "A multilevel algorithm for wavefront reduction," *SIAM Journal on Scientific Computing*, vol. 23, no. 4, pp. 1352–1375, 2001.

[28] G. Kumfert and A. Pothen, "Two improved algorithms for envelope and wavefront reduction," *BIT Numerical Mathematics*, vol. 37, no. 3, pp. 559–590, 1997.

[29] M. Manguoglu, M. Koyutürk, A. H. Sameh, and A. Grama, "Weighted matrix ordering and parallel banded preconditioners for iterative linear system solvers," *SIAM Journal on Scientific Computing*, vol. 32, no. 3, pp. 1201–1216, 2010.

[30] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[31] HSL, "A collection of fortran codes for large scale scientific computation. http://www.hsl.rl.ac.uk," 2013.

[32] C. Chevalier and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering," *Parallel computing*, vol. 34, no. 6-8, pp. 318–331, 2008.

[33] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *2016 IEEE IPDPS*, May 2016, pp. 22–31.

[34] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[35] NVIDIA, "cuSolver :: CUDA Toolkit Documentation," https://docs.nvidia.com/cuda/cusolver/index.html, 2020.