Shared-memory implementation of the Karp–Sipser kernelization process

Johannes Langguth
Simula Research Laboratory and
University of Bergen
Bergen, Norway
langguth@simula.no

Ioannis Panagiotas
Sorbonne Université
CNRS, LIP6
F-75005 Paris, France
Ioannis.Panagiotas@lip6.fr

Bora Uçar
CNRS and LIP (UMR5668:
CNRS, Université de Lyon -
Inria - ENS Lyon), France
bora.ucar@ens-lyon.fr

***Abstract*—We investigate the parallelization of the Karp–Sipser kernelization technique, which constitutes the central part of the well-known Karp–Sipser heuristic for the maximum cardinality matching problem. The technique reduces a given problem instance to a smaller but equivalent one, by repeated applications of two operations: vertex removal, and merging two vertices. The operation of merging two vertices poses the principal challenge in parallelizing the technique. We describe an algorithm that minimizes the need for synchronization and present an efficient shared-memory parallel implementation of the kernelization technique for bipartite graphs. Using extensive experiments on a variety of multicore CPUs, we show that our implementation scales well up to 32 cores on one socket.**

I. Introduction

Data reduction is a well-known algorithmic technique when dealing with large problem instances. The goal is to reduce the initial problem instance to a smaller yet equivalent one, called kernel, for which the solution can be found more efficiently. This can lead to significant improvements in run time for a variety of different problems [1].

Our work focuses on data reductions techniques for sparse graphs applications. We examine two data reduction rules originally proposed by Karp and Sipser [9] for the maximum cardinality matching problem. The first rule removes from the graph degree-1 vertices and their neighbor, whereas the second rule merges the neighbors of a degree-2 vertex and removes it from the graph. Rules similar to those of Karp and Sipser can also be applied to additional problems such as vertex cover or maximum independent set [14]. A recent experimental study [12] demonstrates that finding a maximum cardinality matching on the kernel leads to significant speed-ups on several real-life graphs (versus working directly on the original graph). Another recent study [11] discusses efficient implementations of the two reduction rules in the sequential setting, proposes subquadratic algorithms for sparse bipartite graphs, and experimentally demonstrates the importance of both rules.

Our aim is to design and implement an efficient parallel algorithm for the two reduction rules on shared memory systems, focusing on bipartite graphs. Based on existing work [11], we propose an algorithm with reduced synchronization and present an implementation using atomic operations and standard OpenMP directives. On a large set of test instances we show that the proposed implementation scales well up to 32 cores on one socket. As the applications of the reduction rules necessitate updating degrees by visiting vertices in a highly irregular manner using locks or atomic operations, an implementation scaling across multiple sockets on standard architectures seems unlikely to be possible.

Our main contributions are the following:

- To the best of our knowledge, this is the first work that proposes the parallelization of both rules for matching kernelization purposes.
- We propose a specialized data structure for merging vertices efficiently.
- We achieve 10x speed-ups compared to the fastest sequential code on Kronecker graphs and 6.6x speed-ups on other large instances using recent multicore CPUs.

The rest of the paper is organized as follows. Section II provides the background and summarizes related work. In Section III we propose efficient parallel algorithms for the two rules along with a specialized data structure. Section IV presents the experimental results. Section V concludes the paper.

II. Background and related work

Let $G = (V_A \cup V_B, E)$ be a simple bipartite graph, with V_A and V_B being the two disjoint vertex parts and E the set of edges. Two vertices u, v are neighbors iff the edge $\{u, v\}$ exists in E. The degree of a vertex v, denoted as d_v, is equal to the number of v's neighbors. A *degree-k* vertex has exactly k neighbors. A *matching* in G is a set of disjoint edges, such that no two edges share a common vertex. The matching with the most edges is referred to as the *maximum cardinality* matching.

The Karp–Sipser [9] kernelization applies the following two reductions on a graph:

- **Rule-1**: If a degree-1 vertex u with neighbor v exists, add edge $\{u, v\}$ to the matching and remove u, v from the graph. One can show that $\{u, v\}$ belongs to a maximum cardinality matching in the graph.
- **Rule-2**: If a degree-2 vertex u with neighbors v and w exists, remove u and its edges from the graph and merge v and w to a new vertex vw whose set of neighbors is the union of those of v and w (excluding u). Either v or w (not both) can be matched in the reduced graph, so that u's pair can be decided afterwards.

When neither rule applies, the graph has been reduced to a kernel. The original paper [9] describes a heuristic which adds a random edge to the matching after the initial kernelization has been performed, removes the matched vertices along with all their edges, and then applies the reduction rules again. The process is repeated, until no edges remain in the graph. We consider the initial kernelization aspect of Karp–Sipser. Once the kernel has been found, one can use an exact algorithm for the matching problem [10], [7], [20] on it, and then convert the solution to one for the original graph.

Rule-1 is easy to implement and has linear run time complexity. Merging two vertices is the most time consuming part of the Karp–Sipser kernelization process. This is so as when two vertices are merged, any duplicate edges must be removed from the graph; and the degrees of their common neighbors should be updated. We make use of a technique called *components*, which was first introduced for developing fast heuristics for the maximum cardinality matching problem [15] and has been recently used for implementing the Karp–Sipser kernelization process in sequential computing environments [11]. A *component* is a connected subgraph that consists of a set of *boundary vertices* and *paths* connecting those boundary vertices. The boundary vertices have arbitrary degree greater than one, and belong to the same vertex part. A path begins and ends with a boundary vertex, has an even number of edges, and every non-boundary vertex in it has degree exactly 2. We also require the paths to be of maximum even length, i.e. they cannot be contained in a longer such path. When dealing with a maximal degree-2 path of odd length, we treat one of its degree-2 vertices as a boundary vertex such that the path's length becomes even. In all other cases, a boundary vertex has degree higher than 2. An example can be seen in Figure 1.

The implementation of Karp–Sipser using components works along the following lines. First, Rule-1 is applied for as long as possible. The algorithm then *grows* degree-2 paths, locates boundary vertices, and identifies the associated components. Components are stored with a union-find structure, where two vertices belong to the same component only if they are part of the same tree. Once all components are known, each component is *shrunk* by merging all of its boundary vertices into a single vertex. This new vertex is called the *head* of the component and assumes the name of one of the boundary vertices. The key idea is that if the merge operations were to be done one-by-one, at the end all boundary vertices in a component would be merged into the same vertex. All these merge operations can thus be performed all at once which is less costly than doing them one by one. The above process repeats until the kernel is identified.

Azad et al. [2] and Patwary et al. [18] examine Karp–Sipser as a parallel matching heuristic. As

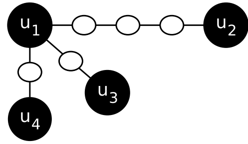


Fig. 1: A sample component, in which the boundary vertices $u_1, \ldots, u_4$ are connected by disjoint degree-2 paths. When the component is shrunk, the boundary vertices will be merged to the component *head* represented by u_1.

both studies consider only Rule-1, they are not directly comparable with our work, since they avoid the more expensive Rule-2 applications. Hespe et al. [6] discuss parallel algorithms for applying kernelization rules to the maximum independent set problem. One of these rules involves merging the neighbors of a degree-2 vertex akin to Rule-2. They use a graph partitioner to separate the vertices into disjoint blocks. The reductions are then applied in parallel on the blocks under the restriction that all involved vertices must belong to the same block; some reductions might not be processed. They leave as future work the parallelization of a method by Chang et al. [3] which uses maximal degree-2 paths similar to components above. Reductions are also applied to the core decomposition problem where at each step the vertex of minimum degree is removed from the graph. The procedure can be perceived as a generalization of Rule-1, with the difference that the neighbors of the removed vertex remain in the graph. There is therefore no equivalent for Rule-2 in core decomposition and parallel algorithms [8] are not comparable with our work. Truss decomposition [21] removes at each step the edge $\{u, v\}$ that participates in the least number of triangles in the graph. After the deletion, any adjacent edge of either u and v must be updated in case it participates in a triangle with $\{u, v\}$ which is done by computing the vertex intersection between the neighbors of u and v. Neighborhood intersection is also part of Rule-2, however, Truss decomposition algorithms lack the concept of merging vertices as well as the removal of duplicate edges.

III. Parallel reduction algorithm

The default version of Karp–Sipser where Rule-2 reductions are applied one by one as soon as they are found can create performance bottlenecks in the parallel setting. That is due to the fact that in order to merge two vertices one needs to enforce mutual exclusion on those vertices so that they are not merged with other vertices. To circumvent this problem and have an efficient parallel solution, we adapt the technique of *components* discussed in Section II. Through the use of this technique, we can produce batches of merge operations that are independent of each other and thus can be performed in parallel efficiently. In addition, merge operations affect the degrees of the vertices in the neighborhood of the merged vertices, and we need to apply atomic operations to properly maintain them. The component strategy also leads to an overall decrease in the number of atomics by performing the merge operations in batches.

Our algorithm works in rounds separated from each other by applications of the two rules. The overall procedure for a round is summarized in Algorithm 1. As it can be seen, a round consists of three steps, and all threads must operate within the same step. The first step applies Rule-1 reductions in parallel. The second step grows the components in parallel, which are shrunk in parallel in the third step. The next three subsections discuss efficient parallel implementations of the three steps.

We keep two buckets B_1 and B_2 which hold the degree-1 and degree-2 vertices, respectively. These buckets are accompanied by two integers n_1 and n_2 which hold the number of vertices in the respective buckets. The variables B_1, B_2, n_1 and n_2 are shared by all threads. In case we need to add an element v to B_1, we set B_1[*atomic_fetch_add*(n_1,1)]=v, where *atomic_fetch_add* performs an addition to n_1 as an atomic set of operations without other threads interfering. The same applies for B_2.

A. First step: Degree-1 reduction

We adopt the approach of Azad et al. [2] for applying Rule-1 in parallel and modify it to avoid recursion, as the recursive calls can exceed the stack size in large instances. The proposed approach to handle degree-1 vertices is given in Algorithm 2. In this algorithm, degree-1 vertices are processed one by one. When a new degree-1 vertex v is discovered, the thread that discovers v stores it in

Algorithm 1 Procedure for a round

```
Input: B_1 of size n_1, shared bucket of degree-1 vertices
Input: B_2 of size n_2, shared bucket of degree-2 vertices
Output: Rules 1 and 2 are applied on vertices, respectively, in B_1 and in B_2
 1: for i = 1 : n_1 (in parallel) do          // 1st Step starts
 2:     u_i ← B_1[i]
 3:     DEG-1-REDUCE(u_i)
 4: for i = 1 : n_2 (in parallel) do          // 2nd Step starts
 5:     u_i ← B_2[i]
 6:     DEG-2-GROW(u_i)
 7: n_c ← num. components                     // 3rd Step starts
 8: for i = 1 : n_c (in parallel) do
 9:     Let u be the head of the i-th component
10:     COMP-SHRINK(u)
```

a private array pB_1 to process it later. To ensure correct parallelization, we use a global array of flags LOCK1, and set LOCK1$[y]$ equal to TRUE whenever y is matched by an application of Rule-1.

When DEG-1-REDUCE, shown in Algorithm 2, is called on x, we first insert x to pB_1. As long as pB_1 is not empty, we pop an element v from it. At first, we perform a test to ensure that v is still a degree-1 vertex in Line 5. If v is indeed a degree-1 vertex, we find its sole remaining neighbor y, and match v and y if LOCK1$[y]$ is FALSE. To check LOCK1$[y]$, we use the *atomic_test_and_set* operation that returns the current value of LOCK1$[y]$ before setting it to TRUE. If v is matched with y, we iterate over y's neighbors and reduce their degree by one using the *atomic_fetch_sub* command which performs the subtraction as an atomic set of operations. If any of them end up being a degree-2 vertex (i.e., old degree equal to 3), we add them to the shared bucket B_2 of degree-2 vertices. Likewise, if any of them ends up being a degree-1 vertex (i.e., old degree equal to 2), we add them to pB_1 to process them later.

The test in Line 5 can be done without atomics. Since v is in pB_1, v's degree was one at some point during Step-1 of Algorithm 1. If v's degree became zero, and it went unnoticed in Line 5, then v's sole neighbor was matched with some vertex at a call of DEG-1-REDUCE. In this case, the test in Line 7 which tries to see if the neighbor of v is available for matching by checking that its LOCK1 value is FALSE will fail and v will remain unmatched.

Algorithm 2 DEG-1-REDUCE(x)

```
Input: x, a degree-1 vertex
Input: B_2, bucket of degree-2 vertices
 1: pB_1 : private degree-1 bucket
 2: pB_1.push(x)
 3: while pB_1 ≠ ∅ do
 4:     v ← pB_1.pop()
 5:     if deg[v] = 1 then
 6:         y ← NEIGHBOR[v]
 7:         b ← atomic_test_and_set(LOCK1[y])
 8:         if b = FALSE then
 9:             match v with y
10:             LOCK1[v] ← TRUE
11:             for z : NEIGHBORS[y] do
12:                 oldegz ← atomic_fetch_sub(deg[z], 1)
13:                 if oldegz = 2 then
14:                     pB_1.push(z)
15:                 else if oldegz = 3 then
16:                     add z to B_2 with atomics
```

B. Second step: Growing components.

The procedure for growing components is summarized in Algorithm 3. We start from a degree-2 vertex x, identify the maximal degree-2 path that x belongs to, and link the two boundary vertices in the path together. Line 3 ensures that each maximal path will be processed at most twice by considering only those degree-2 vertices that are neighbors with a vertex of degree at least three. Algorithm 3 ignores cycles composed exclusively of degree-2 vertices, which can be deleted afterwards as such cycles do not appear in the kernel.

Algorithm 3 DEG-2-GROW(x)

```
Input: x, a degree-2 vertex
 1: if deg[x] = 2 then
 2:     y_1, y_2 ← NEIGHBORS[x] s.t deg[y_1] ≤ deg[y_2]
 3:     if deg[y_2] ≥ 3 then
 4:         while deg[y_1] = 2 and LOCK2[y_1] = −1 do
 5:             LOCK2[y_1].vertex = y_2
 6:             LOCK2[y_1].len = length(PATH(y_1,y_2))
 7:             y_1 ← NEIGHBOR [y_1]
 8:         if deg[y_1] = 2 then
 9:             y_1 ←LOCK2[y_1].vertex
10:         if PATH(y_1, y_2) even then
11:             UFUNION(y_1, y_2)
12:         else
13:             if y_2 ≥ y_1 then
14:                 UFUNION(y_1, x)
```

If degree-2 vertex x has a neighbor y_2 of degree at least three, we start growing a maximal degree-2

path from x's other neighbor y_1. While doing so, we use an atomics-free locking mechanism on the visited vertices along the path by using array LOCK2 during Line 5. That is, for a vertex along the degree-2 path, we set its LOCK2.vertex value equal to y_2 and also store its distance from y_2 in LOCK2.len. If we discover a vertex with LOCK2.vertex $\neq -1$, then we know that another thread has traversed or is simultaneously traversing the same path. We can hence terminate the path traversal in the current thread as both endpoints of the maximal path can be found thanks to LOCK2.

LOCK2 can be maintained without atomics. Even if two threads update the LOCK2 value of the same vertex concurrently, each thread will encounter a locked (or of degree $k \geq 3$) vertex next and stop its traversal. Once the endpoints of the degree-2 path are known (see Line 9), we can link the two boundary vertices as follows:

- If the path has even length, we link together using union-find the two boundary vertices with degree $k \geq 3$ defined by the path.
- Otherwise, we must create a path of even length by excluding an endpoint of the odd path. We thus must link a degree-2 vertex with a vertex of degree $k \geq 3$. We break a tie in Line 13 to decide which endpoint to exclude and call a link operation from one of the two threads that traverse this path.

To find the parity of the path's length we use the LOCK2.len values. We use an efficient parallel implementation of Rem's algorithm [4] by Patwary et al. [19] for the union-find structure. We modify their code to improve efficiency in the subsequent steps by setting the parents of boundary vertices to the root between Steps 2 and 3.

C. Third step: Shrinking components

Algorithm 4 shows the procedure to shrink a component whose head is vertex u_1. In this algorithm, we merge u_1 with the rest of the boundary vertices in its component and remove duplicate edges. The degree of the head of a component is only updated by the thread that is responsible for the component. The degrees of vertices which do not participate in any components at the given round however can be modified by different threads. We first start by populating a merge array with u_1's neighbors in Lines 4–11. If u_1 has edges to many boundary vertices in the same component, we keep only one of them and delete the rest (Line 11).

Algorithm 4 COMP-SHRINK(u_1)

Input: u_1, head-vertex of the component
Input: B_1, B_2, bucket of degree-1,2 vertices respectively
Output: u_1 merged with the boundary vertices in its component

```
1: u_1, ..., u_k : vertices in the component of u_1
2: MERGEARRAY ← [0, ..., 0]
3: NONCOMPONENTS ← ∅
4: for y : NEIGHBORS[u_1] do
5:    z ← UFHEAD[y]
6:    if MERGEARRAY[z]= 0 then
7:       MERGEARRAY[z]← 1
8:       replace y in u_1's adj.list with UFHEAD[y])
9:    else
10:      deg[u_1] ← deg[u_1] − 1
11:      delete edge from y's adj.list
12: for i = 2 : k do
13:    for y : NEIGHBORS[u_i] do
14:       z = UFHEAD[y]
15:       if MERGEARRAY[z]= 0 then
16:          MERGEARRAY[z]= 1
17:          deg[u_1] ← deg[u_1] + 1
18:          add an edge to z in u_1's adj. list
19:       else
20:          if z is a non-component vertex then
21:             Add z to NONCOMPONENTS if needed
22:             MERGEARRAY[z]++
23: add u_1 to B_1 or B_2 with atomics if needed
24: for z : NONCOMPONENTS do
25:    d_z ← MERGEARRAY[z] −1
26:    atomic_fetch_sub( deg[z],d_z)
27:    add z to B_1 or B_2 with atomics if needed
```

We then iterate over the neighbors of the rest of the boundary vertices of the component, and adjust u_1's adjacency list if needed (i.e., link u_1 with a vertex it was not linked to before at Line 18). At Lines 10 and 17, we update the u_1's degree accordingly. Since no other thread will alter u_1's degree, these updates can be done without atomics. At the end, if u_1's degree is either one or two, we add u_1 to the appropriate bucket B_1 or B_2.

We now discuss how to correctly update non-component vertices that are neighbors of the boundary vertices. Instead of decreasing their degree whenever a duplicate edge involving them is found,

we call *atomic_fetch_sub* once. We use NONCOMPONENTS to store a private set of non-component vertices whose degree must be updated. This set is extended when we encounter a non-component vertex with at least two neighbors in $u_1, \dots, u_k$ as shown in Line 21. In order to identify such vertices efficiently, we set their corresponding positions in MERGEARRAY to their number of neighbors in the component, which is increased when needed at Line 22. Once the shrinking is done, we pass over the vertices in NONCOMPONENTS to update their degrees, and add them to B_1 or B_2 if needed.

In Algorithm 4, MERGEARRAY is initialized implicitly. Each thread keeps a value r that plays the role of 1, while any value smaller than r is considered 0. The number of duplicate edges towards a non-component vertex can be given simply by subtracting r from its corresponding value in MERGEARRAY. Once Algorithm 4 has finished, the thread sets r to the maximum value found in MERGEARRAY increased by one. In this way, the same MERGEARRAY can be reused for different components without reinitialization. The way to modify the adjacency lists of non-component vertices is postponed until Section III-D3.

D. The data structure

Applying Rule-1 is straightforward and requires only a stack to keep track of the degree-1 vertices. On the other hand, the implementation of Rule-2 is complicated by the fact that whenever a merge operation occurs we have to adjust the adjacency lists of the vertices. Methods used in handling dynamic graphs or matrices (e.g., [5], [22]) are applicable to our case, but are much more general than the task at hand. We need to transfer edges from one vertex to another while avoiding duplicates. We propose a thread-safe data structure for this purpose. The main idea is to create an implicit link between two merged vertices u, v. Then, we can find the edges of uv by iterating first over u's edges, and then visit neighbors of v not in u's adjacency list.

The proposed data structure extends the well-known sparse matrix format *Compressed Sparse Row* (CSR). In this format, an array `ids` stores the ids of neighbors, and a pointer array `begins` stores the beginning of the list of neighbors of a vertex in `ids` so that the neighbors of the ith vertex are stored in between the locations `begins`$[i]$ and `begins`$[i+1]-1$ of the `ids` array. Our data structure keeps three additional arrays:

- `ends`$[u]$: the position where the edges of u end in the `ids` array;
- `next`$[u]$: Next vertex merged with u;
- `last`$[u]$: Last vertex merged with u.

These three arrays are allocated at the beginning, where `ends`$[u]$=`begins`$[u+1]-1$, `next`$[u]$=-1, and `last`$[u]$=u, for any vertex u. Now we will see how to perform some basic operations.

1) Merging vertices: During Algorithm 4, we merge a component head u_1 with a boundary vertex u_j, where $j > 1$, by setting `next[last`$[u_1]$`]`=u_j and `last`$[u_1]$ = `last`$[u_j]$. The `next` array thus forms a chain-like structure of the merged vertices in each component. In case u_j was merged before with other vertices, u_1 is also merged with these vertices. This operation hence implements Line 18 of Algorithm 4 and updates u_1's adjacency list without explicitly modifying it. To find u_1's neighbors, we iterate the `ids` array from `begins`$[u_j]$ to `ends`$[u_j]$ for all u_j in u_1's `next` chain.

2) Edge deletion: Assume we want to delete an edge of u stored at position `begins`$[u]+k$. To achieve this, we first replace `ids[begins`$[u]+k$`]` with `ids[ends`$[u]$`]` and then reduce `ends`$[u]$ by one so that u's edges remain consecutive.

3) Edge relabelling: Here we show how the data structure updates the adjacency lists of vertices. Lines 11 and 18 of Algorithm 4 show that we can maintain the adjacency list of a component head by renaming or deleting entries in the `ids` array while iterating so that no duplicates edges remain. Algorithm 4 however does not update the adjacency list of a non-component vertex. The entries of such a vertex in `ids` can thus become obsolete, that is store boundary vertices which got merged with their corresponding component head.

To deal with the above problem, we opt for a lazy strategy where we update the entries of a vertex in `ids` only when accessed; for example during Line 6 of Algorithm 2 or Line 4 of Algorithm 4. We make use of a secondary union-find data structure

called OLDUF to distinguish it from the union-find (UF) structure used in Algorithms 3 and 4. OLDUF and UF start as equal during Step-1 of Algorithm 1, but can differentiate in Step-2 as Algorithm 3 updates only UF. After Step-3, we set OLDUF equal to UF again. In other words, at a given round, OLDUFHEAD[u] gives the vertex in the current graph that u is merged to, whereas an UFHEAD[u] value different than OLDUFHEAD[u] signals the vertex that OLDUFHEAD[u] will be merged with at the end of the round.

The OLDUF structure is important once Step-2 begins and vertices become linked with each other in UF. If during Step-2 or Step-3, two vertices x, y have the same UFHEAD value, but their OLDUFHEAD values differ, then until the given round x, y were in distinct vertices which will be merged together at the given round. If a vertex has in its entries in `ids` two vertices with the same OLDUFHEAD value, one of the two can be removed, since implicitly it was deleted during the degree alterations in Algorithm 4.

Algorithm 5 builds upon the idea above to remove any duplicate edges from a given vertex and provides the implementation of the NEIGHBORS function. As seen, its overall structure is similar to that of Algorithm 4. We iterate over the edges in u's adjacency list and remove duplicates as soon as they are found, while all unique neighbors of u are stored in NEIGHBORS$_u$ and returned at the end.

4) Complexity of neighborhood iteration: Here, we discuss the complexity of Algorithm 5. We first assume that OLDUFHEAD in Line 7 runs in $O(1)$ time to derive the run time complexity bounds and then discuss why that is a valid assumption.

Recall that while visiting u's neighbors, we start from u, examine its edges, and move on to the vertices merged with u. Thanks to the swapping deletion technique from the previous subsection, we can ensure that for any v in u's `next` chain, all edges between `begins`[v] and `ends`[v] are either valid neighboring edges of u or duplicate edges than can be removed. The complexity of the above procedure is therefore $O(d_u + N_u)$, where N_u is the number of vertices that u has been merged with. The N_u term can be avoided by discarding

Algorithm 5 NEIGHBORS(u)

Input: u, a vertex of a degree at least one
Output: NEIGHBORS$_u$, the set of neighbors of u

```
1: NEIGHBORS_u ← ∅
2: DUPLICATES ← [0, ..., 0]
3: curr ← u
4: while curr ≠ −1 do
5:     for p = begins[curr] : ends[curr] do
6:         x ← ids[p]
7:         px ← OLDUFHEAD(x)
8:         if px is discarded then
9:             delete this edge from ids
10:        else
11:            if DUPLICATES[px]=1 then
12:                delete this edge from ids
13:            else
14:                ids[p] ← px
15:                add px to NEIGHBORS_u
16:                DUPLICATES[px]← 1
17:    curr ← next[curr]
18: return NEIGHBORS_u
```

all vertices whose `ids` entries were all deleted as duplicates. When we encounter such a vertex v, we update v's predecessor so that its `next` value becomes `next[`v`]`. Thus each accessed vertex either has at least one valid neighbor or is deleted from the chain with $O(1)$ amortized cost. Overall, iterating over u's edges requires $O(2 \cdot d_u)$ time.

Now we discuss the complexity of the OLDUFHEAD operation. Recall that no unions occur in OLDUF. Instead, after Step-3, for each component head u, we iterate over the vertices in u's `next` array chain and set their OLDUFPARENT value to u. While doing so, we remove from `next` any vertex with `begins` larger than `ends` after updating their OLDUFPARENT value as discussed above.

Assume now that while iterating the `ids` entries of u we see vertex v'. If `begins`[v'] $\leq$ `ends`[v'], then OLDUFPARENT[v'] points correctly to the component head of v' and OLDUFHEAD[v'] requires $O(1)$ time. Otherwise, the OLDUFPARENT of v' does not necessarily point to its component head, and we might need to apply several OLDUFPARENT operations to reach the component's head. While this can have a complexity linear in the number of rounds, for most practical purposes the behavior should remain constant by making vertices point directly to their root in OLDUF.

IV. Experiments

A. Experimental setup

In order to obtain experimental results that do not depend on the features of a single hardware platform, we expriment on a set of six different multicore CPUs shown in Table I. Five of these are dual socket NUMA systems, and AMD Epyc 7302P has a single socket.

All codes were compiled with GNU `g++` 10.2.0 using options `-std=c++11, -O3`, and `-march=native`. In all cases we use `numactl` to explicitly select cores which are as close to each other as possible, thus filling a NUMA domain before using a second one. We verified that this gives better results than scattering processes among the NUMA domains. We consider the effect of simultaneous multithreading, or hyperthreading in Intel processor, (SMT) separately. Thus, we separate between thread and core count.

The test set consists of 153 bipartite graphs obtained from matrices available in the SuiteSparse [13] collection. We selected all matrices with more than one million nonzeros, but removed instances that have no degree-1 and degree-2 vertices, as well as those matrices which are too large to be run on all platforms. The smallest and the largest test graphs have 1.54 million and 1.02 billion edges, respectively.

B. Results

We benchmark the run time of our implementation on all platforms with all core counts that are powers of 2 for a single socket using all available SMT settings. AMD and Intel processors allow up to two concurrent threads, Cavium ThunderX2 allows four, and Kunpeng only one. Also, we test the maximum number of threads on two sockets where available. This results in 12,699 individual results. Due to space constraints, we only report the arithmetic mean for each configuration in Figure 2. This means that the results mostly reflect performance on large instances, as the kernelization for smaller instances takes negligible time.

The `Skylake Xeon Gold` showed a relatively good single core performance, and a consistently strong benefit from SMT. The `Icelake Xeon Platinum` starts out similar but scales better, reaching the best performance in the benchmark. Interestingly, even on a single socket, using 32 cores is detrimental to performance here.

Despite its low number of cores, the `AMD Epyc 7302P` shows very good performance. It is about as fast as the `Xeon Gold`. Since no system scales beyond 16 cores or to a second socket, no other system has an advantage here. Only the `Icelake Xeon Platinum` showed noticeably better performance. For the `AMD Epyc 7601`, we actually observe a slowdown at 16 cores. Its module-based design which results in slower communication between cores not placed on the same 8-core module is a likely explanation for this effect. However, the AMD systems show the best single thread performance, which may be due to their larger L3 caches.

The `Cavium ThunderX2` has a single core performance that is noticeably worse than the Epycs and Xeons, but it does scale well to multiple cores. It supports 4-way SMT, which improved upon 2-way SMT for smaller core counts. Similar to other architectures, the best performance was reached at 32 threads using 8 cores with SMT4. Similarly, the `HiSilicon Kunpeng 920-6428`, which features a total of 128 cores but no SMT, reached optimum performance at 32 cores/threads. Its single core performance is the weakest among all CPUs, which means it showed the best scaling since the difference between its maximum performance and that of other architectures is much smaller.

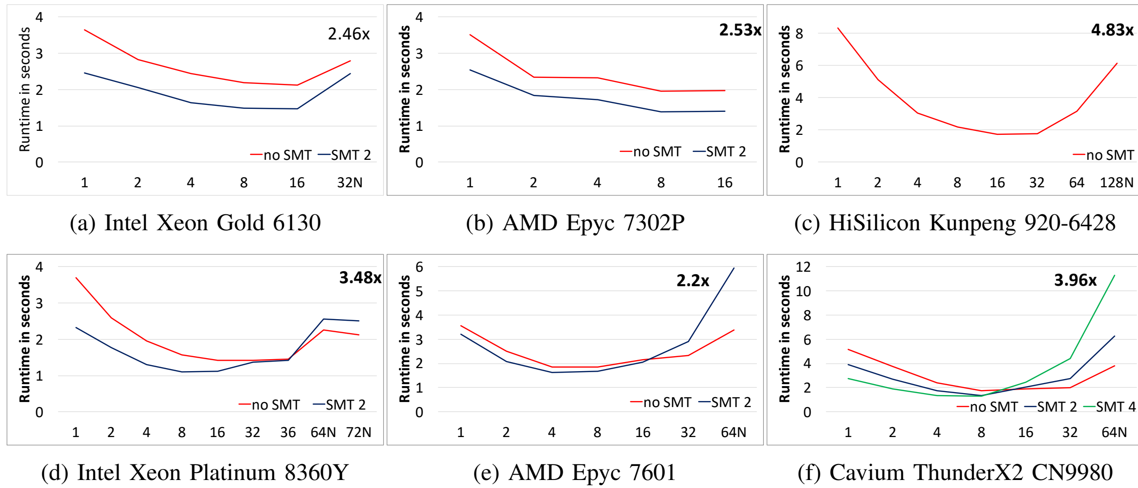
The results show a very consistent picture across all architectures. The performance always scales with the number of cores, up to 8 or 16 cores, and in that range SMT always improves performance. On the other hand, thread counts larger than 32 or the addition of a second socket never gave the best performance due to the use of atomic operations.

In order to test the stability of our results we performed 100 repetitions for each of the 10 different core counts on an `AMD Epyc 7302P` for the `uk-2002` instance. In each case, the run time never diverges more than 10% from the average, and the standard deviation never exceeds 0.093.

Finally, we also take a closer look at five of the largest instances, as well as `kron_g500-logn21`

TABLE I: Hardware used in our experiments. STREAM Triad [16] was run with the stated compilation flags.

Chip Model	AMD Epyc 7302P	Intel Xeon Gold 6130	AMD Epyc 7601	Cavium ThunderX2 CN9980	HiSilicon Kunpeng 920-6428	Intel Xeon Platinum 3rd Gen.
Instruction set	x86-64	x86-64	x86-64	ARMv8.1	ARMv8.2	x86-64
Microarchitecture	Zen 2	Skylake (server)	Zen	Vulcan	TaiShan v110	Ice Lake (server)
Cores/Threads	16/32	2 × 32/64	2 × 32/64	2 × 32/128	2 × 64/64	2 × 36/72
Core frequency (GHz)	3.0 to 3.3	1.9 to 3.6	2.7 to 3.2	2.0 to 2.5	2.8	1.8 to 3.6
L1I/L1D/L2 (KiB per core)	32/32/512	32/32/1024	64/32/512	32/32/256	64/64/512	32/48/1280
L3 cache (MiB per socket)	128 (8x16)	22 (16x1.375)	64 (8x8)	32 (32x1)	64 (64x1)	54
Memory Channels	8	2 × 6	2 × 8	2 × 8	2 × 8	2 × 8
Total Memory (GB)	256	384	2048	1024	1024	512
Max. bandwidth (GB/s)	204.8	238.4	340.8	317.9	381.4	409.6
STREAM Triad (GB/s)	90.6	147.1	161.4	173.3	193.5	293.1



(a) Intel Xeon Gold 6130 (b) AMD Epyc 7302P (c) HiSilicon Kunpeng 920-6428

(d) Intel Xeon Platinum 8360Y (e) AMD Epyc 7601 (f) Cavium ThunderX2 CN9980

Fig. 2: Average run times in seconds over the entire test set by number of cores for all architectures and all available SMT configurations. Dual socket configurations are marked as NUMA. The maximum speedup is given in the upper right corner. NUMA configurations are marked with N.

in Table II. We compare our results to the best sequential code [11] and list the speedup obtained by switching to our parallel code. Similar to many other graph algorithms, instances such as the European road network show little scaling. The *kmer* graphs on the other hand show significant improvement, and the scaling factor increases with the increasing instance size. Finally, the code shows excellent scaling for the Kronecker graph. Such graphs are known to allow relatively good scaling, and they constitute the test instances for Graph500 benchmark [17]. Thus, the behavior of our implementation resembles that of other graph algorithms.

V. Conclusion

We have investigated a shared-memory parallel implementation of the two data reduction rules for the maximum cardinality matching problem proposed by Karp and Sipser [9]. Our implementation is based on the component based approach [15] which allows the algorithm to operate in phases, thereby reducing the need for locking. Our experiments show that the resulting algorithm scales well to 16 cores and 32 threads on one socket, outperforming the best sequential algorithm by up to 10x. In the future, we plan to adapt the components approach to nonbipartite graphs and investigate its parallelization, on distributed memory systems. In

TABLE II: Scaling performance of the largest instances on the Ice Lake Xeon processor with up to 32 cores, with and without hyperthreading. All numbers except speedup are run times in seconds. Seq refers to the fastest sequential code [11], which also forms the basis for our speedup analysis.

Cores/SMT:	Seq.	1/1	2/1	4/1	8/1	16/1	32/1	1/2	2/2	4/2	8/2	16/2	32/2	Speedup
kmer_V2a	41.3	49.8	30.9	19.6	13.7	11.3	10.7	30.0	19.5	12.8	9.8	8.4	8.9	4.9
kmer_U1a	53.6	61.7	36.6	23.0	16.1	13.3	11.7	34.9	21.8	14.1	10.6	9.0	9.1	5.9
kmer_V1r	256.4	319.4	169.1	102.8	68.8	52.5	48.7	179.1	102.3	64.2	48.4	39.3	39.0	6.6
europe_osm	14.5	17.6	14.5	11.7	9.6	9.0	9.1	10.9	10.9	8.4	7.3	6.9	7.5	2.1
webbase-2001	23.1	31.4	21.9	17.3	15.2	15.7	16.3	23.7	16.9	13.8	13.6	12.4	12.9	1.9
kron_g500-logn21.mtx	2.5	3.0	1.7	0.9	0.5	0.3	0.3	2.7	1.6	0.9	0.5	0.4	0.3	10.0

the future, we plan to adapt the components approach to nonbipartite graphs and investigate its parallelization, both on shared and distributed memory systems, which will also address multiple sockets.

Acknowledgements. This work is supported in part by PHC Aurora Programme implemented in France by Ministry of Europe and Foreign Affairs (MEAE) and the Ministry of Higher Education, Research and Innovation (MESRI), and in Norway by the Research Council of Norway (RCN) under contract # 309662. Additional support from RCN was received under contract #303404. This work makes use of hardware funded by RCN under contract #270053.

References

[1] F. Abu-Khzam, S. Lamm, M. Mnich, A. Noe, C. Schulz, and D. Strash, "Recent advances in practical data reduction," *arXiv preprint arXiv:2012.12594*, 2020.

[2] A. Azad, M. Halappanavar, S. Rajamanickam, E. G. Boman, A. Khan, and A. Pothen, "Multithreaded algorithms for maximum matching in bipartite graphs," in *Proc. 26th IPDPS*. IEEE CPS, May 2012, pp. 860–872.

[3] L. Chang, W. Li, and W. Zhang, "Computing a near-maximum independent set in linear time by reducing-peeling," in *Proc. 2017 ACM International Conference on Management of Data*, 2017, pp. 1181–1196.

[4] E. W. Dijkstra, *A discipline of programming*. Prentice-hall Englewood Cliffs, 1976.

[5] T. Gilray, J. King, M. Kirby, and M. Might, "Dynamic-CSR: A format for dynamic sparse-matrix updates," in *Springer-Verlag*, vol. 9697, 2016, pp. 61–80.

[6] D. Hespe, C. Schulz, and D. Strash, "Scalable kernelization for maximum independent sets," *ACM J. Exp. Algorithmics*, vol. 24, Sep. 2019.

[7] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 225–231, 1973.

[8] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *2017 IPDPSW*. IEEE, 2017, pp. 1482–1491.

[9] R. M. Karp and M. Sipser, "Maximum matching in sparse random graphs," in *FOCS'81*, Nashville, TN, USA, 1981, pp. 364–375.

[10] K. Kaya, J. Langguth, F. Manne, and B. Uçar, "Push-relabel based algorithms for the maximum transversal problem," *Computers & Operations Research*, vol. 40, no. 5, pp. 1266–1275, 2013.

[11] K. Kaya, J. Langguth, I. Panagiotas, and B. Uçar, "Karp-Sipser based kernels for bipartite graph matching," in *Proc. ALENEX20*. SIAM, Jan. 2020, pp. 1–12.

[12] T. Koana, V. Korenwein, A. Nichterlein, R. Niedermeier, and P. Zschoche, "Data reduction for maximum matching on real-world graphs: Theory and experiments," *ACM J. Exp. Algorithmics*, vol. 26, Apr. 2021.

[13] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The suitesparse matrix collection website interface," *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019.

[14] S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck, "Finding Near-Optimal Independent Sets at Scale," in *Proc. ALENEX16*, 2016.

[15] J. Langguth, F. Manne, and P. Sanders, "Heuristic initialization for bipartite matching problems," *J. Exp. Algorithmics*, vol. 15, pp. 1.1–1.22, 2010.

[16] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE TCCA Newsletter*, pp. 19–25, Dec. 1995.

[17] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[18] M. M. A. Patwary, R. H. Bisseling, and F. Manne, "Parallel greedy graph matching using an edge partitioning approach," in *Proc. 4th Int. Workshop on High-level parallel programming and applications*, 2010, pp. 45–54.

[19] M. M. A. Patwary, P. Refsnes, and F. Manne, "Multicore spanning forest algorithms using the disjoint-set data structure," in *26th IPDPS*. IEEE, 2012, pp. 827–835.

[20] A. Pothen and C.-J. Fan, "Computing the block triangular form of a sparse matrix," *ACM T. Math. Software*, vol. 16, pp. 303–324, 1990.

[21] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *Proc. HPEC*. IEEE, 2017, pp. 1–6.

[22] B. Wheatman and H. Xu, "A parallel packed memory array to store dynamic graphs," in *Proc. ALENEX21*. SIAM, 2021, pp. 31–45.