Faster Multi-Object Segmentation using Parallel Quadratic Pseudo-Boolean Optimization

Niels Jeppesen, Patrick M. Jensen, Anders N. Christensen, Anders B. Dahl, and Vedrana A. Dahl
Department of Applied Mathematics and Computer Science
Technical University of Denmark, Kgs. Lyngby, Denmark
{niejep,patmjen,anym,abda,vand}@dtu.dk

Abstract

We introduce a parallel version of the Quadratic Pseudo-Boolean Optimization (QPBO) algorithm for solving binary optimization tasks, such as image segmentation. The original QPBO implementation by Kolmogorov and Rother relies on the Boykov-Kolmogorov (BK) maxflow/mincut algorithm and performs well for many image analysis tasks. However, the serial nature of their QPBO algorithm results in poor utilization of modern hardware. By redesigning the QPBO algorithm to work with parallel maxflow/mincut algorithms, we significantly reduce solve time of large optimization tasks. We compare our parallel QPBO implementation to other state-of-the-art solvers and benchmark them on two large segmentation tasks and a substantial set of small segmentation tasks. The results show that our parallel QPBO algorithm is over 20 times faster than the serial QPBO algorithm on the large tasks and over three times faster for the majority of the small tasks. Although we focus on image segmentation, our algorithm is generic and can be used for any QPBO problem. Our implementation and experimental results are available at DOI: 10.5281/zenodo.5201620

1. Introduction

Computational parallelism is essential to the performance and thereby the usefulness of many image segmentation algorithms. The best example is perhaps deep learning, which owes much of its success to highly efficient parallel implementations of the matrix operations used during both training and inference. However, not all algorithms used in computer vision rely on easily parallelizable matrix operations.

Graph cut algorithms are popular for solving binary optimization problems in image analysis, due to their speed and guarantee of optimality. Thus, they provide efficient solutions to a variety of computer vision problems – on their own [8, 9, 11, 16, 23, 26, 33, 35], or in combination with other methods [18, 29, 30]. While some popular graph cut algorithms have been parallelized [11, 36, 41, 43], other algorithms have remained serial, which severely limits their ability to utilize modern hardware. One example is the Quadratic Pseudo-Boolean Optimization (QPBO) algorithm [7, 19, 33, 39], which allows non-submodular energy terms, making it particularly useful for instance segmentation. Instance segmentation without training data is common in microscopy and material science, where manually labeling large volumetric datasets can be highly impractical. Often, the input needed for segmentation with QPBO can be obtained much easier.

In this paper, we introduce the first parallel QPBO (P-QPBO) algorithm. Our goal is to provide an efficient and scalable algorithm that can take advantage of modern multicore processors. With our P-QPBO algorithm, results can be obtained over an order of magnitude faster than with previous serial methods, and the scale of the tasks can be increased significantly. It enables us to segment a volume with hundreds of interacting 3D objects in minutes based on limited user input and no training data. Although we only demonstrate the advantage of P-QPBO for image segmentation, P-QPBO can be used for any QPBO problem.

This work focuses on our parallel algorithm and its time and memory efficiency on image segmentation tasks. Thus, the formulation of suitable energy functions for specific computer vision tasks is outside the scope of this paper.

1.1. Related work

Several algorithms have been developed to solve QPBO problems [19, 32, 33]. In computer vision, the QPBO algorithm [7, 19] implemented by Kolmogorov and Rother [33, 39], utilizing the serial Boykov-Kolmogorov maxflow/mincut (BK) algorithm [9] for solving the optimization problem, is arguably the most popular. Generally, maxflow/mincut algorithms can be separated into three groups [42]: push-relabel algorithms [3, 10, 14, 15], augmenting path algorithms [9, 17], and pseudoflow algorithms [16, 20, 21], which are a hybrid of the two previous categories. BK is an augmenting path algorithm. It is the

most popular maxflow/mincut algorithm in computer vision, due to its performance and ability to handle dynamic maxflow/mincut scenarios, by reusing computations from previous solutions when changes are made to the graph [31]. In the last decade, pseudoflow algorithms like Excesses Incremental Breadth-First Search (EIBFS) [16] have outperformed BK in most static cases, as well as some dynamic cases. However, the overhead associated with graph changes for dynamic problems is still higher for EIBFS than for BK.

Push-relabel algorithms have traditionally been the target of most parallelization efforts [2, 5, 11, 13, 22, 43], as operations mainly act locally, making them well-suited for parallel execution. However, synchronization overhead means that many threads are needed to achieve good performance [6, 40]. More recent works [36, 40, 41, 44, 45] have focused on parallelizing augmenting path algorithms. Here, a graph is partitioned into multiple sub-graphs and a serial algorithm is applied to each sub-graph in parallel. Information is then propagated between sub-graphs, or they are merged. This process is repeated until a global solution is found. Parallel pseudoflow algorithms have not been attempted yet.

Finally, as grid-based graphs are relatively common in computer vision, algorithms specialized for this structure, such as Grid-Cut [24, 25], have also been developed. Grid structured graphs have also been the target of GPU-based implementations [38, 43], as they rely on the highly regular structure of grid graphs to fully utilize the GPU. While grid-based algorithms achieve significant performance improvements, they are only usable on a limited (but certainly important) subset of binary optimization problems. In this paper, we are concerned with a general-purpose parallel QPBO algorithm and will therefore not discuss the grid-based algorithms further.

1.2. Contribution

We introduce a fast parallel algorithm for solving QPBO problems. It is based on the efficient two-stage approach of the QPBO algorithm as presented in [33] and the bottom-up merging approach from [36]. Our algorithm is fully compatible with the original QPBO algorithm and we prove that it is guaranteed to find equivalent solutions.

We show that our parallel algorithm reduces the solve time significantly on a large multi-object 3D segmentation task compared to current state-of-the-art approaches. We also benchmark our algorithm for segmentation using a large set of 2D images and show significant performance improvements, even for smaller segmentation tasks.

Our implementation, benchmark code, results, notebooks, and proof are available at 10.5281/zenodo.5201620.

2. QPBO

We briefly summarize the original QPBO algorithm here. Both the QPBO algorithm and general-purpose maxflow/mincut algorithms can be used to minimize energy functions of the form

$$E(\mathbf{x}) = \sum_{p \in \mathcal{V}} \theta_p(x_p) + \sum_{p,q \in \mathcal{V}} \theta_{pq}(x_p, x_q) \,. \tag{1}$$

Here, $\mathcal{V}$ is a set of nodes, $x_p \in \{0, 1\}$ are the node labels, θ_p are unary energy terms, and θ_{pq} are pairwise energy terms. If E is submodular, meaning that all pairwise energies satisfy

$$\theta_{pq}(0,0) + \theta_{pq}(1,1) \leq \theta_{pq}(0,1) + \theta_{pq}(1,0) \,, \tag{2}$$

then maxflow/mincut-based algorithms (including QPBO) are guaranteed to find the global optimal solution to the minimization problem [9, 33]. However, if the energy function contains non-submodular terms, we cannot directly model it as a maxflow/mincut problem [12, 34]. To overcome this limitation, the QPBO algorithm uses an extended graph approach, in which every node is represented by two graph nodes: a node $p \in \mathcal{V}$ and a flipped node $\overline{p} \in \overline{\mathcal{V}}$. Every energy term is then represented by two graph edges (see Table 1). This allows the non-submodular terms to be represented as graph edges between nodes p and q, and the flipped nodes $\overline{p}$ and $\overline{q}$. Computing the maximum flow/minimum cut of this extended graph corresponds to minimizing the energy function (1) [33] given that all terms are non-negative. One can convert a QPBO function to an equivalent non-negative one using the linear time algorithm described in [33].

Table 1. Conversion from energy terms to graph edges [33], where s the source node, t is the sink node, and $\overline{p}, \overline{q} \in \overline{\mathcal{V}}$ are the flipped versions of the nodes $p, q \in \mathcal{V}$.

Energy term	Corresponding edges	Edge capacity
$\theta_p(0)$	$(p \to t), (s \to \overline{p})$	$\frac{1}{2}\theta_p(0)$
$\theta_p(1)$	$(s \to p), (\overline{p} \to t)$	$\frac{1}{2}\theta_p(1)$
$\theta_{pq}(0,1)$	$(p \to q), (\overline{q} \to \overline{p})$	$\frac{1}{2}\theta_{pq}(0,1)$
$\theta_{pq}(1,0)$	$(q \to p), (\overline{p} \to \overline{q})$	$\frac{1}{2}\theta_{pq}(1,0)$
$\theta_{pq}(0,0)$	$(p \to \overline{q}), (q \to \overline{p})$	$\frac{1}{2}\theta_{pq}(0,0)$
$\theta_{pq}(1,1)$	$(\overline{q} \to p), (\overline{p} \to q)$	$\frac{1}{2}\theta_{pq}(1,1)$

However, the support for non-submodular terms means that the guarantee of finding the global optimal solution is replaced by one of finding a partial optimal solution [33]. This means that we may get a solution with unlabeled nodes, which may lead to bad segmentation results [23]. However, when non-submodular terms are used only for exclusion, [26] has shown that unlabeled nodes are rare and hardly affect the resulting segmentation.

The original QPBO algorithm uses an efficient two-stage approach [33]. In Stage 1, only submodular terms are considered and the problem is modelled and solved as a regular

maxflow problem, *i.e.*, without the flipped nodes. In Stage 2, the flipped graph is created by copying the residual graph from Stage 1 and reversing the edges. Finally, the edges for the non-submodular terms are added and the solution is updated. This two-stage approach reduces the solve time significantly, but relies on maxflow solvers that can handle dynamic graphs efficiently, *e.g.*, the BK algorithm [31].

3. Parallel QPBO

We now describe our new Parallel QPBO (P-QPBO) algorithm, which combines the two-stage QPBO approach described in Section 2 with the bottom-up merging parallelization approach by Liu and Sun [36]. Judging from previous work [36, 40, 41, 44, 45], bottom-up merging provides good performance on non-distributed multi-core systems. Like Liu and Sun's algorithm, ours has two phases. In Phase A, the QPBO problem is split into disjoint sub-problems, which are solved independently in parallel. In Phase B, these partial solutions are merged and re-solved, also in parallel, to get the complete solution. Note that Phase A/B strictly refers to the splitting and merging of sub-problems. Stage 1/2 refers to whether the sub-graph associated with a sub-problem has had the flipped graph added or not.

In contrast to Liu and Sun's algorithm, which strictly works as a maxflow/mincut solver, our algorithm also considers each sub-problem as a QPBO problem. Specifically, each sub-problem is kept in Stage 1 (where we do not need the flipped graph) as long as it contains only submodular terms. Thus, in the case of few non-submodular terms, most sub-problems will remain in Stage 1 for most of Phases A and B. This significantly reduces the solve time. We will now describe the two phases, including the specific conditions, which will trigger a conversion to Stage 2 for a sub-problem. Figure 1 shows a visual summary.

Phase A: Partitioning of the QPBO problem is done by splitting the underlying graph $\mathcal{G} = \langle \mathcal{V} \cup \overline{\mathcal{V}}, \mathcal{E} \rangle$. We split the node set $\mathcal{V}$ into N disjoint sets $\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_N$. This gives a partition of the graph nodes into blocks $\mathcal{W}_1, \mathcal{W}_2, ..., \mathcal{W}_N$ where $\mathcal{W}_i = \{p, \overline{p} \mid p \in \mathcal{V}_i\}$. Then, for each pair of blocks $\mathcal{W}_i, \mathcal{W}_j$ connected by one or more edges, we identify inter-block edges and store these in separate lists. From now on, we refer to these edge lists as *block interfaces*. After building the block interfaces we remove inter-block edges from $\mathcal{G}$.

We now have a series of sub-graphs $\mathcal{G}_i = \langle \{\mathcal{W}_i, s, t\}, \mathcal{E}_i \rangle$ where $\mathcal{E}_i = \{(p \rightarrow q) \in \mathcal{E} \mid p, q \in \mathcal{W}_i\}$. Because these sub-graphs are disconnected, except through the source and sink nodes, we can compute their individual maxflow solutions in parallel (see Figure 1a). For each sub-graph, we adapt the two-stage approach from the serial QPBO algorithm. First, we only consider submodular terms and do not add the flipped graph. Then, if (and only if) a sub-graph contains non-submodular terms, we transition the sub-graph to Stage 2. During this transition, the flipped graph is constructed by copying the residual graph from Stage 1, the remaining non-submodular edges are added, and the maxflow solution is updated (see Figure 1b). When all sub-graphs have been solved, we move to Phase B.

Phase B: In this phase we merge the sub-graphs to re-create the original extended graph, $\mathcal{G}$. Merging two sub-graphs is done by re-adding the inter-block edges, which were removed in Phase A. If all sub-graphs and inter-block edges correspond to submodular terms, we keep both sub-graphs in Stage 1 (see Figure 1c). If some of the inter-block edges correspond to non-submodular terms, both sub-graphs are transformed to Stage 2 (see Figure 1d) before the inter-block edges are added (see Figure 1e). Furthermore, if the two sub-graphs are in different stages, the sub-graph in Stage 1 is transformed to Stage 2 before inter-block edges are re-added and the sub-graphs are merged (see Figure 1f). After merging, the solution of the combined graph is updated.

To further reduce the solve time, we want merges to happen in parallel, for which we use the strategy from [36]. Updating the maxflow solution is a serial process, so only one thread can work on a sub-graph at a time. For synchronization, each sub-graph can be locked (meaning it is being worked on) or unlocked (meaning it is free for merging).

To decide which sub-graphs to merge, each thread scans through the list of block interfaces created in Phase A, until it finds one that connects two unlocked sub-graphs. The thread then locks the sub-graphs and merges them. Then, it re-computes the maxflow solution for the merged sub-graph and the sub-graph is unlocked. Note that after sub-graphs have been merged, there may be several block interfaces connecting the previously merged sub-graphs. Therefore, when a thread finds a pair of sub-graphs to merge, it continues to scan the list of block interfaces to find all block interfaces connecting the pair. The block interfaces are then removed from the global list and the merge proceeds. A global synchronization object is used to ensure that only one thread can scan the list of block interfaces at a time.

At the end of Phase B, the number of remaining merges will be less than the number of running threads (unless only one thread is used). Therefore, if a thread scans the whole list of block interfaces without encountering a pair of unlocked sub-graphs, it terminates. As a result, the degree of parallelism is gradually reduced near the end of Phase B. However, for most problems, the time required for the last merge will be small compared to the total solve time. In total, the number of merges performed will be one less than the number of blocks. The process for each thread is summarized in Algorithm 1.

3.1. Correctness

Our P-QPBO algorithm will always give a solution equivalent to that of the serial QPBO algorithm.

Energy: The energy of the solution is given by the unique

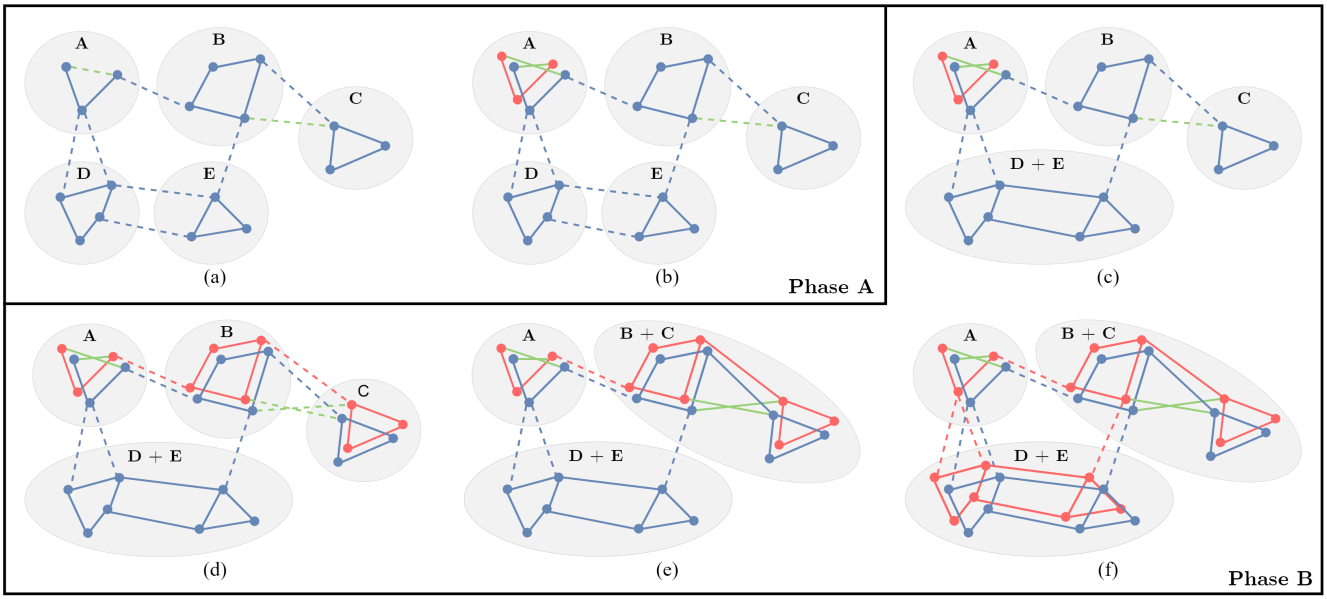

Figure 1. Illustration of merging strategy. Blue dots and lines represent nodes and edges in the non-flipped graph, while red dots and lines are nodes and edges in the flipped graph. Green lines represent edges between a node $p \in \mathcal{V}$ and flipped node $\overline{q} \in \overline{\mathcal{V}}$, corresponding to non-submodular terms. Dashed lines represent inter-block edges, which are re-added when the sub-graphs are merged. The exception is green dashed lines between two blue nodes. These represent non-submodular energy terms, which will be translated to edges, once the flipped graphs are added. Source/sink nodes and edges are not shown. (a) The graph is split into sub-graphs and the Stage 1 solution for each sub-graph is computed in parallel. (b) Sub-graph A contains internal non-submodular terms, so it is transformed to Stage 2 and the solution is updated. (c) Sub-graphs D and E are merged. Inter-block edges are re-added and the sub-graph solution is updated. As all intra- and inter-block terms are submodular the sub-graph remains in Stage 1. (d) A term between B and C is non-submodular, so the sub-graphs are transformed to Stage 2 to prepare for merge. (e) Sub-graphs B and C are merged. (f) Sub-graph D + E is transferred to Stage 2 to allow merges with the remaining sub-graphs. All sub-graphs are now in Stage 2, and merging can proceed as normal bottom-up merging.

value of the minimum cut for the extended graph. Since the final graph is identical for the serial and parallel algorithms, and they both compute a minimum cut, the solutions must have the same energy.

Labeling: There may be several minimum cuts which have the same cost/energy but label a different number of nodes [33]. However, given a residual graph, the algorithm from [4, 33] will choose the minimum cut that labels the maximum number of nodes. It can be shown (proof in supplementary material) that since both QPBO and P-QPBO compute a minimum cut of the same graph, they must label the same nodes after running this extra algorithm. Since this extra step is an insignificant part of the overall runtime, we do not include it in our runtime experiments.

3.2. Efficient graph partitioning and merging

The partitioning of the graph nodes into blocks is important for the performance of the P-QPBO algorithm. While our method allows for any partitioning of nodes, ideally, we want as much work as possible to be done in Phase A (computing the partial solutions) and as little as possible to be done in Phase B (merging sub-graphs and updating solutions). A good way to achieve this is to separate the nodes into blocks that are densely packed (many intra-block terms) and sparsely related (few inter-block terms). This speeds up the merging by reducing the number of changes made to the graph. Of course, the ideal partitioning very much depends on the energy function.

For image segmentation, we can use the spatial position of the nodes/pixels when partitioning them into blocks. Cutting the image into evenly sized rectangular blocks, as done by [36, 41] should result in many intra-block terms, compared to inter-block terms, as long as the blocks are not very small. When solving instance segmentation tasks using Sparse Layered Graphs (SLG) [26], an intuitive way to partition the nodes is to create a block per label/object. This works well when the interaction between the objects is low compared to the size of the objects (which is usually the case), and we have at least as many objects as the number of parallel threads available on the system. We use this natural way of partitioning the nodes for all our experiments, as most of our images contain many objects.

Algorithm 1: Phase B of the parallel QPBO algorithm for each thread.

```
while true do
    Lock synchronization object
    Let S = ∅
    foreach block interface s do
        Let G_i and G_j be sub-graphs connected by s
        if both G_i and G_j are unlocked then
            S = {s_i | s_i connects G_i and G_j}
            break
    Remove entries of S from list of block interfaces
    if S is empty then
        Unlock synchronization object
        return
    Lock sub-graphs G_i and G_j connected by S
    Unlock synchronization object
    /* Ensure sub-graphs are in stage 2 if needed. */
    if S contains non-submodular terms or G_i in stage 2
      or G_j in stage 2 then
        if G_i in stage 1 then Transform G_i to stage 2
        if G_j in stage 1 then Transform G_j to stage 2
    Unite sub-graphs G_i and G_j to sub-graph G_ij
    /* Re-insert boundary edges */
    foreach inter-block edge e in S do
        Reinsert e in graph
        if G_ij in stage 2 then
            Reinsert flipped edge of e in graph
        if nodes of a have different labels then
            Mark nodes of reinserted edges as active
    Update maxflow for subgraph G_ij
    /* Make G_ij available for merges */
    Lock synchronization object
    Unlock sub-graph G_ij
    Unlock synchronization object
```

For determining the merging order, P-QPBO uses the same approach as Liu and Sun [36]. After Phase A, we loop over each block interface and count the number of potential new augmenting paths, when merging the sub-graphs containing the blocks. This serves as a heuristic for how much work must be done when merging the sub-graphs. The list of block interfaces is then sorted in descending order based on the number of potential new augmenting paths, in the hope that threads will perform the most expensive merges first. The goal is to do as much work as possible early in Phase B, while the degree of parallelism is high.

4. Benchmark results

To test the scalability of our P-QPBO algorithm, we compare it with two serial QPBO implementations. The first is a slightly optimized implementation of the original QPBO algorithm by Kolmogorov – we call it K-QPBO. The reason we are using a slightly modified version is that the original implementation has overflow issues for large graphs. The second serial implementation is our own re-implementation of K-QPBO, which contains numerous improvements in data structures and optimizations of the code that improves performance. We call this implementation Modern QPBO (M-QPBO). M-QPBO is included to provide a more fair comparison between a serial and parallel implementation since M-QPBO contains the same performance optimization as P-QPBO. When referring to results for our parallel implementation, we use the notation P-QPBO(t), where t is the number of parallel threads used by P-QPBO.

We test the QPBO implementations on the two datasets used in [26], and use the exact energy functions shared in [27]. Our notebooks (based on [27]), used to formulate the energy functions and to benchmark the QPBO algorithms, are included in our supplementary material (DOI: 10.5281/zenodo.5201620). However, as our focus in this paper is purely on the computational performance, the energy formulations are not included in the paper.

The first dataset used for our experiments is a high-resolution μCT 3D image of nerves [28] shown in Figure 2a. This is a large segmentation task with many non-overlapping objects. It allows us to test the scalability of the parallel QPBO implementation across many CPU threads. The second dataset is the BBBC038v1 `stage1_train` (S1) nuclei image set from the Broad Bioimage Benchmark Collection [37]. An image from the dataset along with the instance segmentation results is shown in Figure 2b. Using these images, we test the performance of the QPBO implementations on a variety of small and medium-sized segmentation tasks. For both datasets, the energy function creates a nontrivial graph topology consisting of irregularly interconnected ordered multi-column sub-graphs. Unlike general maxflow problems, where a number of commonly used benchmark datasets exist [16], there are no commonly used benchmark datasets specifically for QPBO.

We use two Intel Xeon Gold 6226R (16 cores / 16 threads) CPUs in dual socket configuration for all our benchmarks. With this, we test how our implementation scales on a modern architecture with up to 32 threads executing in parallel.

4.1. Large segmentation tasks

The goal of this experiment is to compare the solve times of the K-QPBO and M-QPBO to those of P-QPBO at various parallel thread counts on large segmentation tasks. Although solve times vary between system architectures, this experiment shows the benefit of using P-QPBO, depending on the number of CPU cores available.

In the experiment, we segment the myelin and axon of 216 nerves in a $2048 \times 2048 \times 2048$ volume at two different radial sampling resolutions, using the SLG method of [26]. The first resolution (N1) is the one used by [26], while the second resolution (N2) is higher, resulting in a graph more

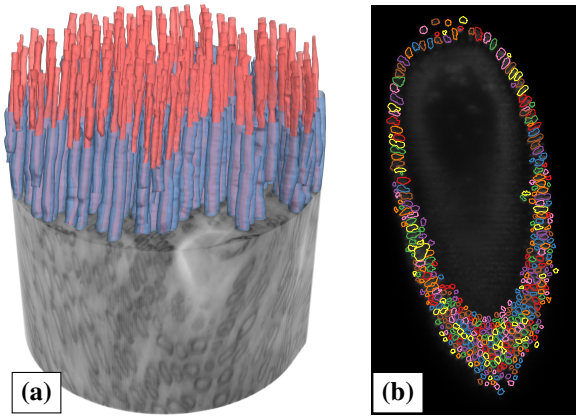

Figure 2. (a) Result of N1 nerve segmentation task. Nodes are split into blocks such that all nodes associated with either the inner (red) or outer (blue) surface of a nerve are in the same block (two blocks per nerve). (b) Example of nuclei segmentation on the image from S1 with the most nuclei. Nodes are split into blocks such that all nodes associated with a cell are in the same block (one block per cell).

	N1	N2
Nodes	363,748,800	818,434,800
Edges	2,124,073,454	4,864,255,488
Memory footprint		
K-QPBO [33]	134.2 GB	306.8 GB
M-QPBO	60.1 GB	182.7 GB
P-QPBO	70.0 GB	224.9 GB
Fastest solve time		
K-QPBO [33]	836 s	4,987 s
M-QPBO	628 s	3,704 s
P-QPBO (1)	558 s	2,429 s
P-QPBO (16)	94 s	323 s
P-QPBO (32)	80 s	264 s
P-QPBO (40)	**74 s**	245 s
P-QPBO (48)	76 s	**239 s**

Table 2. Graph details for the nerve segmentation tasks. Nodes and edges refer to the size of the full extended graph. The memory footprint is the total memory footprint of the graph and relevant bookkeeping. P-QPBO and M-QPBO use 32-bit indices for N1, but 64-bit edge indices for N2, because it has more than 2^{31} edges. K-QPBO always uses 64-bit pointers for indexing. The solve times are shown for each of the three algorithms, with a number of different thread configurations for P-QPBO. Each solve time is the minimum of ten runs for N1 and three runs for N2.

than twice the size of N1 (see Table 2). In both cases, the output is a 3D multi-label segmentation with a total of 432 interacting objects (two per nerve). For P-QPBO we use one block per object (see Figure 2a).

As shown in Table 2, our M-QPBO implementation reduces the solve time for the N1 task by 25% compared to the K-QPBO implementation, while P-QPBO(1) outperforms M-QPBO slightly for this task, with a 33% reduction compared to K-QPBO, using only a single thread. Using two threads, P-QPBO(2) provides a 62% reduction in solve time, compared to K-QPBO, and a 49% reduction compared to M-QPBO. The best result is achieved using 40 threads, in which case P-QPBO is over 11 times faster than K-QPBO. Figure 3a show the relative speed-up when using P-QPBO compared to K-QPBO. We see that the performance increases up to and beyond the number of CPU cores (32) in our test system.

For the larger N2 task, we observe even larger performance improvements and better scaling of P-QPBO than for N1 (see Figure 3b). From the solve times in Table 2, we see that the bottom-up merging strategy, even without parallelism, provides a reduction in solve time of 51% for P-QPBO(1) compared to K-QPBO. Meanwhile, M-QPBO provides a 26% reduction over K-QPBO. In other words, P-QPBO clearly improves its relative performance as the task grows, while M-QPBO performs similarly for N1 and N2, when looking at the relative improvement over K-QPBO.

Both Figures 3a and 3b show that the speed-up increases significantly less past 16 threads. This is expected, as we are testing on a dual socket system, which means we are likely to experience some degree of computational overhead when using both CPUs, especially for cache and memory intensive tasks such as computing the maximum flow. Yet, despite the overhead, the combined 32 CPU cores allow P-QPBO to scale past 32 threads for both N1 and N2, with P-QPBO(40) significantly outperforming P-QPBO(32) in both cases. This is perhaps a result of some threads idling while waiting for the synchronization lock to be released.

Another reason for the way P-QPBO scales with the number of threads is the reduction in the degree of parallelism at the end of Phase B. According to Amdahl's law [1], this puts a theoretical maximum to the speed-up, which in our case will depend on the energies and blocking strategy. For the nerve segmentation tasks we estimate a parallel fraction of 0.88 and 0.92 (including overhead) for N1 and N2, respectively, meaning that most of the work is done in parallel.

We expect that most of the performance improvement of M-QPBO over K-QPBO is due to the smaller memory footprint of the graphs shown in Table 2. We achieve this reduction by using more compact data structures for nodes and edges. Instead of 64-bit pointers, we use 32-bit indices where possible. Furthermore, we store forward and backward edges adjacent in memory to avoid storing pointers between these. P-QPBO and M-QPBO use the same fundamental data structures for nodes and edges. The increased

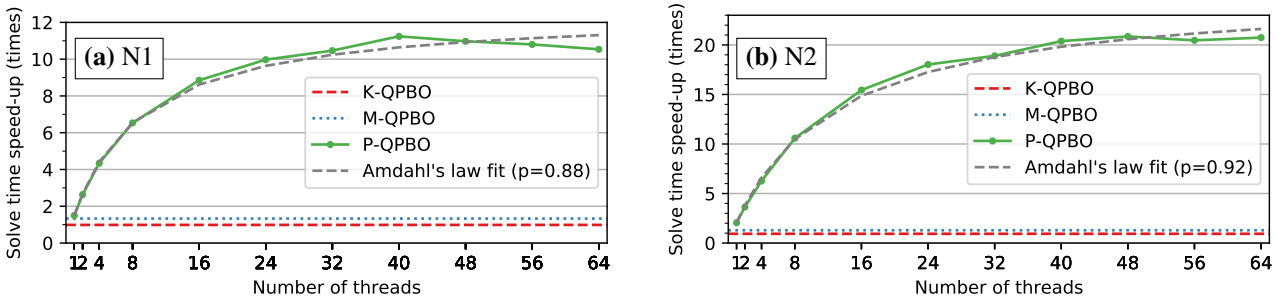

Figure 3. Plots showing the relative speed-up in the solve time when using P-QPBO compared to K-QPBO and M-QPBO. The speed-up is calculated using the fastest solve time out of ten runs for N1 and three runs for N2. K-QPBO and M-QPBO are represented as horizontal lines, as they always use a single thread. We also show a fit of Amdahl's law [1] and the parallel fraction, p. Keep in mind that two 16 core CPUs were used, which means we expect the speed-up to stagnate or even decrease when using more than parallel 32 threads. For these tasks, the stagnation appears to start at 40 threads on our test system.

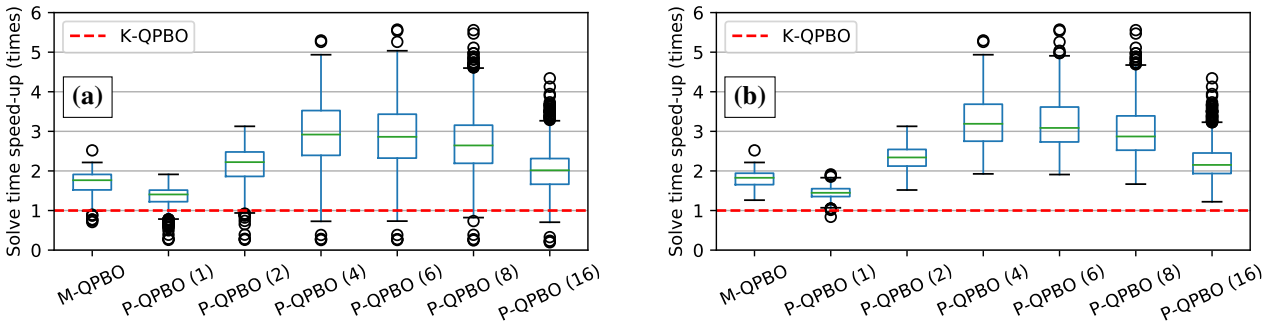

Figure 4. Box plots showing the relative speed-up for each image in the S1 dataset, for M-QPBO and P-QPBO compared to K-QPBO. Following Tukey's definition, the green line in the box is the median, the box marks the two quartiles and the whiskers show the minimum and maximum values, excluding outliers. Outliers are defined as values more than 1.5 times the interquartile range from the nearest quartile and are shown as rings. In (a) the results for all 670 images are shown, while (b) only includes the 502 images with 16 or more nuclei. The relative speed-up is calculated using the fastest solve time for each method, with each method having been run ten times.

memory footprint of the P-QPBO graph is a result of extra bookkeeping needed for the bottom-up merging. Reducing the memory footprint of the graph structures is important for two reasons. 1) It increases performance due to improved CPU cache and memory efficiency. 2) It allows us to solve larger tasks without running out of memory.

It is important to remember that the scaling depends both on the optimization problem and the system architecture. Generally, we would expect M-QPBO to outperform P-QPBO(1) on smaller tasks, due to the overhead of merging the sub-graphs. However, for large tasks, using bottom-up merging, even without parallel computations, actually turns out to be faster. This behavior was previously noted by [16, 36] and is probably due to a combination of shorter augmenting paths and better cache efficiency.

For the N1 task, [26] reported a solve time of 44 minutes for K-QPBO, which is much higher than the 14 minutes we found in our experiments. We suspect that the main reason for the big difference is that their system only had 112 GB memory, while the graph has a footprint of at least 134 GB. This could have caused memory swapping, which would likely impact performance negatively.

4.2. Smaller segmentation tasks

We use the S1 dataset, previously used in [26], to compare the performance of P-QPBO, M-QPBO, and K-QPBO on a large set of 2D (non-grid) segmentation problems of varying sizes. Figure 5 shows the distributions of graph nodes and edges for the images. With a median of 437,400 nodes and 1,598,060 edges, we consider most of these segmentation tasks relatively small. A few of the tasks are significantly larger, with the largest (shown in Figure 2b) having just over six million nodes and 60 million edges. To examine the overall performance of M-QPBO and P-QPBO for these small to medium-sized tasks, we compute the relative speed-up when using our implementations compared to K-QPBO.

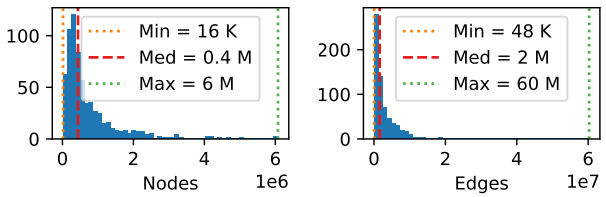

Figure 5. Histograms of the distribution of nodes and edges for the graphs used when segmenting the images in S1.

Figure 4 shows the relative speed-up for M-QPBO and P-QPBO for each image in the dataset (Figure 4a) and for each image with 16 or more nuclei (Figure 4b). Both M-QPBO and P-QPBO show a significant improvement compared to K-QPBO. When we include all images, there are cases where the relative performance drops. In these few cases, the tasks are very small (few nodes and terms), such that the overhead of P-QPBO outweighs the benefits. If we look only at the 502 images with 16 or more nuclei (259,200 nodes or more), M-QPBO and P-QPBO significantly outperform K-QPBO for all images, except when using P-QPBO with a single thread. For these smaller tasks, the overhead of merging blocks is not outweighed by the shorter augmenting paths. Thus, when using only a single thread, the best performance is achieved without bottom-up merging.

For the images with 16 or more nuclei (Figure 4b), M-QPBO gives a median speed-up of 1.8x, with a maximum of 2.5x. P-QPBO(4) achieves the best overall performance, with a median speed-up of 3.2x and a maximum of 5.3x. While P-QPBO(6) and P-QPBO(8) show the best performance in a few of the largest tasks, the overall performance decreases slightly when compared to using four threads, due to the majority of the tasks being relatively small.

4.3. Comparison with other solvers

We compare P-QPBO with other state-of-the-art maxflow/mincut algorithms. To test the effect of the two-stage QPBO strategy, we compare with our own implementation of the parallel maxflow/mincut algorithm by Liu and Sun [36]. Furthermore, we compare with the serial EIBFS algorithm [16], as it is currently the fastest serial maxflow/mincut solver, and we compare with our own parallel version of EIBFS (P-EIBFS) based on bottom-up merging. Finally, we include a best case estimate for a QPBO implementation using EIBFS as its maxflow/mincut solver (EIBFS-QPBO).

We compare the algorithms on the N1 and S1 tasks. The maxflow/mincut algorithms are evaluated by first converting the QPBO problem to the full extended graph and then running the algorithm on this graph. We do not include this conversion time in the benchmark. Results are shown in Table 3. We see that our algorithm significantly outperforms the other methods.

	Memory	Best speed-up	
	N1	N1	S1
M-QPBO	60.1 GB	1.33	1.72±0.28
P-QPBO	70.0 GB	**10.47**	**2.96±0.89**
Liu-Sun [36]	70.0 GB	6.07	0.78±0.13
EIBFS [16]	175.1 GB	1.08	0.33±0.08
P-EIBFS	175.8 GB	0.63	0.68±0.14
EIBFS-QPBO*	175.1 GB	2.17	0.66±0.08

*Best case estimate (half of EIBFS solve time).

Table 3. Results of ablation experiment. We consider up to 32 threads for N1 and up to 16 for S1. We report memory and best speed-up for N1 and best mean±std. speed-up for the S1 data. Speed-ups are computed w.r.t. K-QPBO. The maxflow/mincut solvers were run on the full extended graph. We do not include the time used to convert the QPBO problem to a graph.

5. Conclusion

Our P-QPBO algorithm is the first parallel QPBO algorithm. It scales much better than the serial K-QPBO algorithm on modern multi-core hardware, by partitioning the task into sub-tasks and solving them in parallel. It uses a bottom-up merging strategy to combine the solutions, also in parallel. This allows P-QPBO to solve tasks, such as image segmentation, significantly faster than current algorithms.

Our experiments show that P-QPBO solves large multi-object segmentation tasks over 20 times faster than K-QPBO, with lower memory usage. It does so while remaining fully compatible with K-QPBO, making no constraining assumptions about the graph structure. Even for smaller tasks, with just a few hundred thousand nodes, P-QPBO is 2-5 times faster than K-QPBO, using only four threads. This indicates that P-QPBO will significantly outperform K-QPBO, even on consumer hardware.

The scalability of P-QPBO, when combined with modern hardware, makes P-QPBO suitable for solving much larger optimization tasks than previously possible. Furthermore, because it is a parallel algorithm, we expect the relative performance of P-QPBO to keep increasing in the future. Finally, P-QPBO is a general algorithm, which is suitable for many binary optimization tasks, not just image segmentation. Thus, we are confident that P-QPBO can be used not just for faster image segmentation, but also for a wide range of other tasks, both in computer vision and other fields.

Acknowledgements

This work is supported by FORCE Technology and The Center for Quantification of Imaging Data from MAX IV (QIM).

References

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967. 6, 7

[2] Richard Anderson and Joao C. Setubal. A parallel implementation of the push-relabel algorithm for the maximum flow problem. *Journal of Parallel and Distributed Computing (JPDC)*, 29(1):17–26, 1995. 2

[3] Chetan Arora, Subhashis Banerjee, Prem Kalra, and SN Maheshwari. An efficient graph cut algorithm for computer vision problems. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 552–565, 2010. 1

[4] Bengt Aspvall, Michael F Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979. 4

[5] David A Bader and Vipin Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. Technical report, Georgia Institute of Technology, 2006. 2

[6] Niklas Baumstark, Guy Blelloch, and Julian Shun. Efficient implementation of a synchronous parallel push-relabel algorithm. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 106–117, 2015. 2

[7] Endre Boros, Peter L Hammer, and Xiaorong Sun. Network flows and minimization of quadratic pseudo-boolean functions. Technical report, Technical Report RRR 17-1991, RUTCOR, 1991. 1

[8] Yuri Boykov and Gareth Funka-Lea. Graph Cuts and Efficient N-D Image Segmentation. *International Journal of Computer Vision*, 70(2):109–131, nov 2006. 1

[9] Yuri Boykov and Vladimir Kolmogorov. An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 26(9):1124–1137, 2004. 1, 2

[10] Boris V Cherkassky and Andrew V Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997. 1

[11] Andrew Delong and Yuri Boykov. A scalable graph-cut algorithm for nd grids. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, 2008. 1, 2

[12] Daniel Freedman and Petros Drineas. Energy minimization via graph cuts: Settling what is possible. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 939–946, 2005. 2

[13] Andrew V Goldberg. Processor-efficient implementation of a maximum flow algorithm. *Information Processing Letters*, 38(4):179–185, 1991. 2

[14] Andrew V Goldberg. The partial augment–relabel algorithm for the maximum flow problem. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 466–477, 2008. 1

[15] Andrew V Goldberg. Two-level push-relabel algorithm for the maximum flow problem. In *International Conference on Algorithmic Applications in Management*, pages 212–225, 2009. 1

[16] Andrew V Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert E Tarjan, and Renato F Werneck. Faster and More Dynamic Maximum Flow by Incremental Breadth-First Search. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 619–630, 2015. 1, 2, 5, 7, 8

[17] Andrew V Goldberg, Sagi Hed, Haim Kaplan, Robert E Tarjan, and Renato F Werneck. Maximum Flows by Incremental Breadth-First Search. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 457–468, 2011. 1

[18] Zhihui Guo, Ling Zhang, Le Lu, Mohammadhadi Bagheri, Ronald M Summers, Milan Sonka, and Jianhua Yao. Deep LOGISMOS: deep learning graph-based 3D segmentation of pancreatic tumors on CT scans. pages 1230–1233, 2018. 1

[19] Peter L Hammer, Pierre Hansen, and Bruno Simeone. Roof duality, complementation and persistency in quadratic 0–1 optimization. *Mathematical Programming*, 28(2):121–155, 1984. 1

[20] Dorit S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations Research*, 56(4):992–1009, 2008. 1

[21] Dorit S. Hochbaum and James B. Orlin. Simplifications and speedups of the pseudoflow algorithm. *Networks*, 61(1):40–57, 2013. 1

[22] Bo Hong and Zhengyu He. An asynchronous multithreaded algorithm for the maximum network flow problem with non-blocking global relabeling heuristic. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 22(6):1025–1033, 2010. 2

[23] Hossam Isack, Olga Veksler, Ipek Oguz, Milan Sonka, and Yuri Boykov. Efficient optimization for hierarchically-structured interacting segments (HINTS). In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1445–1453, 2017. 1, 2

[24] Ondřej Jamriška and Daniel Sỳkora. GridCut. Version 1.3. https://gridcut.com, 2015. Accessed 2020-06-12. 2

[25] Ondřej Jamriška, Daniel Sỳkora, and Alexander Hornung. Cache-efficient Graph Cuts on Structured Grids. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3673–3680, 2012. 2

[26] Niels Jeppesen, Anders N Christensen, Vedrana A Dahl, and Anders B Dahl. Sparse layered graphs for multi-object segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12777–12785, 2020. 1, 2, 4, 5, 7

[27] Niels Jeppesen, Anders Nymark Christensen, Vedrana Andersen Dahl, and Anders Bjorholm Dahl. Sparse Layered Graphs for Multi-Object Segmentation (notebooks). 6 2020. 5

[28] Niels Jeppesen, Anders Nymark Christensen, Vedrana Andersen Dahl, Anders Bjorholm Dahl, Hans Martin Kjer, Martin Bech, and Lars Dahlin. Sparse Layered Graphs for Multi-Object Segmentation (data). 11 2020. 5

[29] Anna Khoreva, Rodrigo Benenson, Jan Hosang, Matthias Hein, and Bernt Schiele. Simple does it: Weakly supervised instance and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 876–885, 2017. 1

[30] Alexander Kirillov, Evgeny Levinkov, Bjoern Andres, Bogdan Savchynskyy, and Carsten Rother. Instancecut: from edges to instances with multicut. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5008–5017, 2017. 1

[31] Pushmeet Kohli and Philip H.S. Torr. Dynamic graph cuts for efficient inference in Markov random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 29(12):2079–2088, 2007. 2, 3

[32] Vladimir Kolmogorov. Convergent tree-reweighted message passing for energy minimization. In *Proceedings of the International Workshop on Artificial Intelligence and Statistics*, pages 182–189, 2005. 1

[33] Vladimir Kolmogorov and Carsten Rother. Minimizing non-submodular functions with graph cuts-a review. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 29(7):1274–1279, 2007. 1, 2, 4, 6

[34] Vladimir Kolmogorov and Ramin Zabin. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 26(2):147–159, 2004. 2

[35] Kang Li, Xiaodong Wu, Danny Z Chen, and Milan Sonka. Optimal surface segmentation in volumetric images-a graph-theoretic approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 28(1):119–134, 2005. 1

[36] Jiangyu Liu and Jian Sun. Parallel Graph-cuts by Adaptive Bottom-up Merging. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2181–2188, 2010. 1, 2, 3, 4, 5, 7, 8

[37] V. Ljosa, K. L. Sokolnicki, and A. E. Carpenter. Annotated high-throughput microscopy image sets for validation. *Nature Methods*, 9(7):637–637, 2012. 5

[38] Yi Peng, Li Chen, Fang Xin Ou-Yang, Wei Chen, and Jun Hai Yong. JF-Cut: A parallel graph cut approach for large-scale image and video. *IEEE Transactions on Image Processing*, 24(2):655–666, 2015. 2

[39] Carsten Rother, Vladimir Kolmogorov, Victor Lempitsky, and Martin Szummer. Optimizing binary MRFs via extended roof duality. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, 2007. 1

[40] Alexander Shekhovtsov and Václav Hlaváč. A distributed mincut/maxflow algorithm combining path augmentation and push-relabel. *International Journal of Computer Vision (IJCV)*, 104(3):315–342, 2013. 2, 3

[41] Petter Strandmark and Fredrik Kahl. Parallel and Distributed Graph Cuts by Dual Decomposition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2085–2092, 2010. 1, 2, 3, 4

[42] Tanmay Verma and Dhruv Batra. MaxFlow Revisited: An Empirical Comparison of Maxflow Algorithms for Dense Vision Problems. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 1–12, 2012. 1

[43] Vibhav Vineet and P J Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1–8, 2008. 1, 2

[44] Miao Yu, Shuhan Shen, and Zhanyi Hu. Dynamic Parallel and Distributed Graph Cuts. *IEEE Transactions on Image Processing*, 25(12):5511–5525, 2015. 2, 3

[45] Miao Yu, Shuhan Shen, and Zhanyi Hu. Dynamic Graph Cuts in Parallel. *IEEE Transactions on Image Processing*, 26(8), 2017. 2, 3