

# A Parallel Framework for Parametric Maximum Flow Problems in Image Segmentation

Vlad OLARU<sup>1,2\*</sup>, Mihai FLOREA<sup>1</sup>, and Cristian SMINCHISESCU<sup>1</sup>

<sup>1</sup>Institute of Mathematics of the Romanian Academy, 21 Calea Grivitei St., 010702 Bucharest, Romania

<sup>2</sup>University of Bucharest, 90 Panduri St., 050663 Bucharest, Romania

Email: vlad.olaru@imar.ro\*, vlad.olaru@unibuc.ro,  
mihai.florea@imar.ro, cristian.sminchisescu@imar.ro

\* Corresponding author

**Abstract.** This paper presents a framework that supports the implementation of parallel solutions for the parametric maximum flow computational models widely used in image segmentation algorithms. The framework is based on supergraphs, a special construction combining several image graphs into a larger one, and works on various architectures (multi-core or GPU), either locally or remotely in a cluster of computing nodes. The framework can also be used for performance evaluation of parallel implementations of maximum flow algorithms. We present the case study of a state-of-the-art image segmentation algorithm based on graph cuts, Constrained Parametric Min-Cut (CPMC), which uses the parallel framework to solve parametric maximum flow problems, based on a GPU implementation of the well-known push-relabel algorithm. Our results indicate that real-time implementations based on the proposed techniques are possible.

**Key-words:** Computer vision; image segmentation; parametric maximum flows in graphs.

## 1. Introduction

Recent advances in image segmentation [1] have led to improved accuracy over large and diverse image datasets [2], [3], by almost doubling the performance figures. This development has spurred the interest for the widespread use of image segmentation models (Fig. 1) as a component for key tasks in computer vision, such as video segmentation, large-scale applications for recognition and classification or mobile computing. In this context, of particular importance becomes the real-time performance of image segmentation algorithms. Although reliant on advanced methodology and data structures, the running times of the best performing algorithms still lag behind real-time, taking a few minutes for usual images, on average.

The most advanced image segmentation algorithms involve repeatedly solving multiple maximum-flow problems over monotonic schedules of parameter scales (parametric max-flow) constrained at image “seeds” corresponding to different locations in an image. Each image can be represented as a graph, where each pixel is a node connecting locally with spatially adjacent ones (e.g. up, down, left and right), and connection strengths are modulated by pixel intensity similarity, or the presence of image contours. Solving each max-flow problem for one setting of the parameters is equivalent to computing a binary partition on the image graph. Performed systematically, at different locations and for monotonic schedules of parameters, it has been empirically observed that the process generates multiple binary segmentation hypotheses with good spatial overlap with the different objects and scene structures present in images (see Fig. 1). Often, the hypothesis generation is initiated from different seeds independently, suggesting an inherently high degree of parallelism. Therefore, a trivially parallel implementation that generates solutions by running parametric maximum flow [4], [5] independently for each seed seems appropriate.



**Fig. 1.** (best seen in colors) Original image, pool of segments generated by a max-flow segmentation algorithm, and ground truth respectively.

However, the high computational cost of generating segment hypotheses once a location (seed) has been selected suggests parallelizing the parametric maximum flow procedure as an alternative way to speed up image segmentation. In this paper, we present the design of a general framework that can use existing parallel graph cut solutions such as GridCut [6] or CUDA NPPI [7], [8] to implement a parallel algorithm that approximates parametric maximum flow behavior. To this end, we use supergraphs, a special construction that combines several image graphs, each having edge weights (or capacities) that depend on a different parameter, into a larger one. The framework is general in terms of the architectures it can use. Supergraphs can run on multicore processors or GPU boards, either locally or distributed in a cluster. A parametric maximum flow problem encoded as a collection of supergraph cut problems can be dynamically scheduled on a heterogeneous collection of computing nodes. The dynamic scheduler efficiently adapts both to the imbalances induced by the heterogeneous architectures used, and to those intrinsic to the problem, as each problem takes a different amount of time, depending on the image complexity.

The paper also presents a case study of a state-of-the-art image segmentation algorithm, Constrained Parametric Min-Cut (CPMC) [1], that uses the parallel framework with NVIDIA’s GPU implementation of the well-known push-relabel maximum flow algorithm [9]. The comparison to CPMC based on a sequential pseudoflow algorithm [5] in a trivially parallel setup (where instances of a segmentation problem are executed independently on several cores of a processor) helps understand how close one can get to a real-time solution for image segmentation.

To summarize, the paper contributions are: (1) a parallel solution for parametric max-flow problems based on supergraphs that can be used in image segmentation algorithms; (2) a general, parallel framework for parametric maximum-flow problems that can (a) handle various hardware

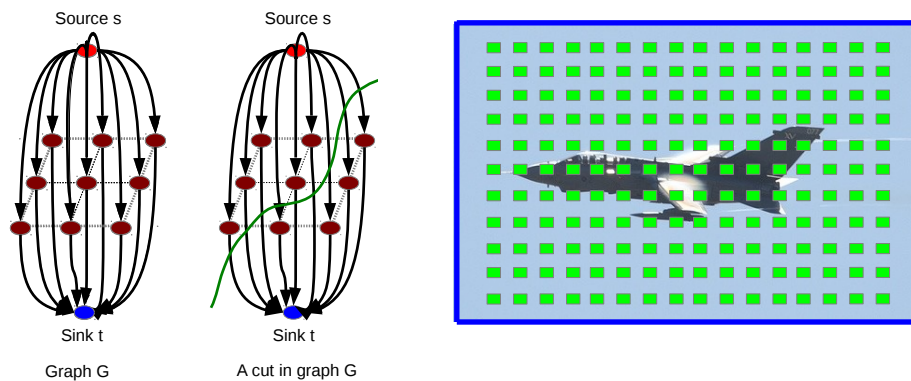
architectures, multi-core or GPU, both locally and remote in a cluster, (b) efficiently schedule problems to achieve improved segmentation times, and (c) act as a performance evaluation tool by allowing the use of various implementations of parallel maximum flow algorithms; and (3) a case study for a state-of-the-art image segmentation algorithm (CPMC).

The paper is organized as follows: Section 2 describes previous work on graph cuts and image segmentation. Section 2.2 discusses the multi-core implementation of a trivially parallel solution for CPMC. Section 3 presents parallel solutions for parametric max flow algorithms using supergraphs. Section 4 shows the results of our case study evaluation of CPMC.

## 2. Graph-Cut based Image Segmentation

Graph cuts can be used to segment an image into a foreground object and the rest of the image, usually referred to as background, in order to obtain a figure-ground segmentation. This is a form of binary classification, with 1 assigned to foreground pixels and 0 to background.

The binary inference (labeling) process is performed by running a maximum flow/minimum cut algorithm on a graph whose vertices represent the pixels in the image. Two special vertices, the *source*  $s$  and the *sink*  $t$  are connected to every vertex of the graph by means of weighted edges (see Fig. 2 left); the weights are called edge capacities. For image segmentation, the source and sink are associated with the two labels that will be used to distinguish the foreground object from the background. The weights of the edges that link  $s$  and  $t$  to the graph vertices (the pixels) quantify a penalty expressing how correct it is to assign that pixel to either of the two classes of labels represented by the source and the sink.



**Fig. 2.** Left: associated image graph and a cut example. Right: a regular grid of seeds in an image. Binary partitions (segmentations) are extracted around regularly placed foreground seeds (green dots) that express the foreground bias, while background seeds (in blue) are placed usually on the image border. Each generated segment corresponds to a graph cut segmentation problem, which results in multiple independent, heavy-cost solutions for an entire image.

Regular graph vertices (corresponding to image pixels) are linked to each other by weighted edges as well. Typically, image segmentation models use the weights of the edges that connect each vertex to its nearby neighbors (up, down, and laterally) to model smoothness, i.e. the assumption that nearby pixels are likely to have similar labels.

An  $s$ - $t$  cut of the graph is a partitioning of the vertices into two disjoint subsets: one containing vertex  $s$  and the other one containing vertex  $t$ . The *cost* of the cut is defined to be the sum of the

weights of those edges in the graph that have one vertex in the  $s$ -partition and the other in the  $t$ -partition. A *minimum cut* corresponds to those graph cuts that have minimum cost.

A graph cut induces a labeling of the image pixels, depending on which partition they were inferred to. The problem of finding a cut is equivalent to the one of minimizing an energy defined on the graph. The energy has two terms, depending on which type of edges the cut crosses: edges linking either  $s$  or  $t$  to a regular vertex (pixel), or regular edges that link neighboring pixels. The first category of terms is called “data” or “unary” terms, while the second accounts for the “pairwise” terms (regularization terms). A minimum cut in such an image graph corresponds to a minimum energy among all of the possible label configurations of the image graph.

Greig *et al.* [10] have used this method for the first time to smooth noisy images and showed that the maximum a posteriori estimate of a binary image corresponds exactly to the maximum flow in the associated image graph constructed as previously described. According to the Ford-Fulkerson theorem [11], a maximum flow from  $s$  to  $t$  *saturates* the sum of the capacities of a set of edges in the graph that partitions the vertices into two disjoint sets that actually correspond to a minimum cut in the graph.

There are many polynomial time algorithms that solve the maximum flow problem (see [12]), including augmenting path (Ford-Fulkerson [11]) and push-relabel [9] algorithms, but their presentation is beyond the scope of this paper. An augmenting path algorithm widely used in computer vision is due to Boykov and Kolmogorov [13]. An extended view on the use of graph cuts in computer vision can be found in [14].

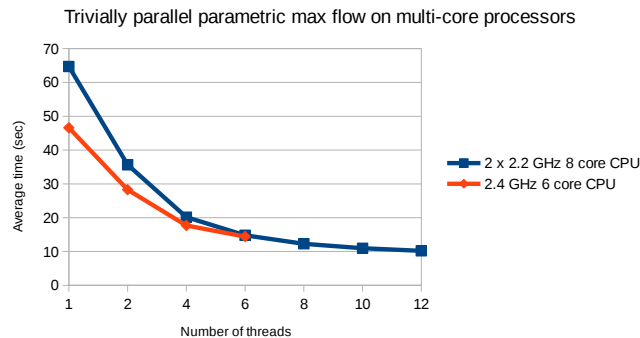
## 2.1. Parametric max flow & image segmentation

Parametric max flow algorithms [4], [15] are used in image segmentation to generate a set of hypotheses for plausible object segments in a given image. They optimize energies where the unknowns are the binary labels of pixels and the weighting (scale)  $\lambda$  between the unary and pairwise terms of the energy model. The  $\lambda$  values for which the corresponding energy value changes are called *breakpoints* and mark the optimal solutions of a parametric max flow problem. If the factors multiplying the parameter  $\lambda$  in the unary energy terms are all nonnegative or nonpositive (the monotonic case), the optimal solutions are nested [4] and an efficient implementation of the parametric max flow is possible. Monotonicity makes the algorithm significantly more efficient, as earlier computations are reused. The algorithms can compute either all breakpoints (an upper bound is the number of graph nodes) or a subset thereof. In practice, a preset list of parameter values (usually defined on a logarithmic scale), called the  $\lambda$ -*schedule*, can be used instead of computing all the breakpoints, as empirical evaluations [1] have shown that the ground truth covering stays almost the same, at significantly lower computational cost due to the reduced number of breakpoints generated (and thus, a reduced number of segment hypotheses). We say that this type of run *approximates* parametric max flow behavior.

Graph cut problems (preferably monotonic) are associated with different seeds in order to generate a pool of segments with high probability of (foreground) object overlap. A seed is a set of pixels “frozen”, by construction, to belong to either foreground or background. The foreground seeds are usually placed regularly on an image grid, while background seeds are assigned to the borders of the image (see Fig. 2 right). A collection of maximum flow problems is solved for each pair of foreground and background seeds and different  $\lambda$  values (the  $\lambda$ -schedule), that are used to express the so called foreground bias associated with the non-seed pixels. The result is a large and diverse set of segments of different sizes and structural (shape) relevance.

## 2.2. Trivial parallelism on multi-core processors

A list of problems defined by a pair of foreground and background seeds and a  $\lambda$ -schedule can be solved independently on different processing units, given that no two pairs of foreground and background seeds are the same. For instance, a trivially parallel solution can be implemented by using MATLAB's *parfor* instruction (or similar instructions in other languages) that executes each iteration independently as a thread on one of the available processor cores.



**Fig. 3.** Trivial parallelism performance for figure-ground segmentations.

The main advantage of this type of parallelization is simplicity, both in terms of programming effort and negligible need of synchronization of the worker threads (thus enabling maximum parallelism). From a programming point of view, one only needs to mark the appropriate *parfor* code blocks. The programming model is sequential for any particular *parfor* code block solving a problem, and the speed-up comes from the high usage of the available cores of the processor.

However, this parallel solution does not attempt to speed up any of the individual problems which run sequentially. Fig. 3 shows the mean time in seconds taken by the pseudoflow algorithm [5] to yield figure-ground hypotheses for a set of 500 images from the VOC [2] dataset, with a schedule of 20  $\lambda$  values and 178 seeds, each. Please note that, in practice, even with a relatively small number of processing units (cores), the speed-up of the trivially parallel solution flattens out quite quickly. As soon as 10 cores are used, the performance of the slower processor (Intel Xeon E5-2660) starts to saturate above 10 seconds, still far from real-time expectations. Even on the faster processor (Intel Xeon E5-2620 v3) the execution times get close to the other processor times as soon as 6 cores are used. This lack of scalability motivates the need to investigate parallel solutions for the parametric maximum flow solver as well.

## 3. Parallel Parametric Max Flow Solution

To the best of authors' knowledge, there is no available parallel implementation of a parametric maximum flow algorithm. A few prior articles focus on the topic of parallel implementations of maximum flow [16], [17], but do not offer code. One available implementation is GridCut [6], and works for multi-core processors only. It defines a grid of computing units that can process graph cuts in parallel based on a popular augmenting path algorithm featuring tree-reuse [13]. GridCut implements adaptive bottom-up merging [18] and cache efficient memory layout [19]. Other available implementations of max flow algorithms are GPU imple-

mentations [7], [8], [20], [21]. The NVIDIA NPPI library [7], [8] implements a push-relabel algorithm [9].

Given the circumstances, running a parallel parametric maximum flow algorithm proves challenging. One solution would be to seek an “approximation” of the parametric behavior (in the sense defined in Section 2.1) by using a preset  $\lambda$ -schedule, and run a parallel maximum flow routine once for each  $\lambda$  value in the schedule. However, this “batch” call is far from optimal, since it is proven that, in a monotonic case, a parametric maximum flow algorithm can run asymptotically close to a regular maximum flow algorithm [4], i.e. with the same theoretical complexity. Hence, this batch procedure is a poor match to what an optimal parallel parametric maximum flow algorithm could achieve in theory.

### 3.1. Parametric max flow with supergraphs

Not only fails the batch method to optimize computations in the monotonic case, but by running a single graph-cut problem at the time, it might not use the available hardware resources to the fullest. This becomes striking especially for the latest generation of GPU boards that feature thousands of computing cores.

To address the issue, please note that since the parallel graph-cut routines offered by the programming interfaces of software like GridCut or CUDA NPPI take a single graph as a parameter, running several graph-cut problems simultaneously on the available parallel computing infrastructure needs to pass larger graphs as parameters to these routines. The solution is to “knit” together several graphs representing different problems into a larger graph, which we call a “supergraph”, and to pass it as a parameter to the graph-cut calls.

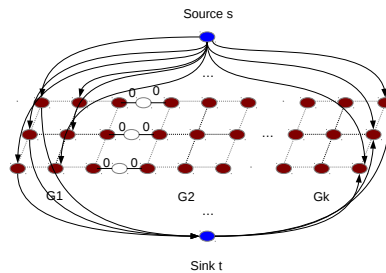
These supergraphs represent the building block of our parallel framework for parametric max flow problems in image segmentation and can be constructed at two levels:  $\lambda$ - and seed-level. A  $\lambda$ -supergraph combines graphs for several  $\lambda$  values, whereas a seed-supergraph combines several  $\lambda$ -supergraphs together. Usually, our structures combine an entire  $\lambda$ -schedule, the list of the  $\lambda$ s we run the parametric max flow with, but it is possible to have smaller supergraphs as well. However, for seed-supergraphs, we use only  $\lambda$ -supergraphs constructed for an entire  $\lambda$ -schedule.

The combination of two graphs into a supergraph is simply done by inserting additional vertices “between” the two graphs and linking them to the regular vertices of the left and right graphs by means of zero weight edges (see Fig. 4). Inductively, arbitrarily large supergraphs can be built from individual graphs. Any minimum cut in such a supergraph is a union of the disjoint minimum cuts of the original graphs knitted together, plus some zero-weight edges that do not affect the overall cost of the supergraph cut. Thus, computing the minimum energies associated with a pair of foreground-background seeds and a given  $\lambda$  value can be derived from such a supergraph by decomposing the minimum supergraph cut into its individual minimum cut components. In other words, computing a supergraph maximum flow/minimum cut approximates the behavior of a parallel parametric maximum flow algorithm running on the individual graphs.

To see why this works, consider the following situation. Let  $S$  be a supergraph composed of  $k$  individual graphs  $G_1, G_2, \dots, G_k$  (see Fig. 4). Let’s assume  $C$  is a minimum cut in  $S$ , and  $C_1, C_2, \dots, C_k$  are the minimum cuts of the individual graph components. Suppose that one of the individual cuts, say  $C_i$ , is not a minimum cut in  $G_i$ . Then, there is a minimum cut  $C'_i$  in  $G_i$ , different than  $C_i$ . Since  $G_i$  is linked by means of zero weight edges to its neighboring graphs in  $S$ , any minimum supergraph cut that crosses  $G_i$  needs to include  $C'_i$ , because the zero-weight crossing edges do not contribute to the overall cost,  $C'_i$  is the minimum cost path severing  $G_i$  into

two partitions and the supergraph cut must somehow cross  $G_i$ . Hence, there is another smaller cost supergraph cut including  $C'_i$  than  $C$ , which contradicts the assumption that  $C$  is a minimum supergraph cut. Thus, all  $C_i$ s must be minimum cuts in their corresponding  $G_i$ s.

Conversely, suppose that there is a supergraph cut  $C'$  in  $S$  with a smaller cost than  $C$ . Then, there is another union of individual minimum cuts  $C'_1 \cup C'_2 \cup \dots \cup C'_k$  that compose  $C'$ . Since  $C'_i$ s are disjoint sets, it means that at least one of them, say  $C'_j$ , is smaller than its corresponding  $C_j$ , which contradicts the assumption that  $C_j$  is a minimum cut in  $G_i$ . Therefore, it is sound to use this procedure to amass several graphs into a supergraph and the individual components of the minimum supergraph cut as individual minimum cuts. Thus, one can simultaneously solve either a single seed problem (by building a  $\lambda$ -supergraph) or several seed problems (by binding together several  $\lambda$ -supergraphs for several seeds). The ability to run custom-sized graphs results in better usage of the available computing power of the underlying hardware architecture.



**Fig. 4.** Supergraph composed out of  $k$  individual graphs  $G_1, G_2, \dots, G_k$  (best seen in colors).

We have used supergraphs both with GridCut and CUDA NPPI, but our CPMC case study focuses on the GPU solution, as our evaluations showed that GridCut performs significantly worse than CUDA in terms of runtimes (see Section 5). Nevertheless, we emphasize that the supergraph method is general and can be used with any available parallel graph cut implementation as a means to compute a parametric maximum flow in parallel. The method is especially effective when the parallel graph cut source code is not available (e.g. CUDA NPPI).

### 3.2. Exposing additional supergraph parallelism

It is known that exchanging the roles of source and sink, operation that we call an  $s$ - $t$  swap, does not affect the results of a graph cut algorithm (i.e., the maximum flow/minimum cut remain the same). However, it might help a parallel implementation of a push-relabel algorithm, like NVIDIA's *nppiGraphcut* [7], [8], run faster [22]. The reason for this behavior is that the parallel workload at every iteration of the algorithm is given by the number of regular vertices (pixel nodes) that have residual capacity on their edges to/from source independent of the edges to/from sink [22]. So, if there are more such vertices for the sink than for the source, swapping them exposes more parallelism.

Choosing the source and sink by running the algorithm twice, once with the original graph and then after an  $s$ - $t$  swap, to see which run yields faster results, is a time-consuming solution. Instead, we use an heuristic to choose the source and sink: for each regular graph vertex, the difference between the capacities of its source and sink edges is computed. Then, for each vertex, the positive and negative differences are added. If the number of negative differences is larger than that of positive ones, we apply the  $s$ - $t$  swap. This procedure leads to more active hardware

resources per iteration of the algorithm and significantly improves performance (see Section 5).

When using *s-t swaps* with supergraphs, one has to properly choose the source and sink so that every individual graph is aligned for maximum available parallelism. Thus, all the individual graphs composing a supergraph must be checked if they need to be *s-t swapped* so that the resulting supergraph has a source and a sink that allow the highest possible degree of parallelism. That is easier done for  $\lambda$ -supergraphs, because such a supergraph represents the same problem (i.e., the same foreground-background pair of seeds), but care must be taken when building seed supergraphs, that might need to reverse some of the individual  $\lambda$ -supergraphs.

### 3.3. Using the parallel framework

A collection of seed problems (each represented by a pair of foreground and background seeds and a  $\lambda$ -schedule) is encoded by means of supergraphs, as previously described. The resulting set of supergraphs may have a smaller size than the collection of seed problems if several such problems are expressed by means of seed supergraphs. Each resulted supergraph is scheduled for parametric maximum flow processing on a given computing node (a.k.a server), either locally or remote. Remote processing is achieved by means of Remote Procedure Calls (RPC) for the supergraph cut routines. A master node, which runs the image segmentation algorithm, controls the scheduling and may act as computing server as well, but in this case the local computing architecture, either CPU- or GPU-based, is going to be accessed directly (without RPC). The resulting cluster of servers solving the set of seed problems may be heterogeneous.

#### 3.3.1. Supergraph scheduling

The master node can perform two types of parallel, non-preemptive scheduling: static and dynamic. Static scheduling assigns supergraphs to computing servers using a MATLAB *parfor* instruction with  $n$  threads (each parallel loop iteration  $i$  gets task  $i \bmod n$ ). All the tasks of a class  $i \bmod n$ , e.g., those that execute an RPC to a given remote server, will be executed sequentially and non-preemptively, one after the other. Thus, the makespan (the maximum value of the tasks completion times) is determined by the time needed to run the longest class  $i$  of tasks  $i \bmod n$ .

However, in a heterogeneous computing environment with different hardware architectures (e.g., different types of GPU boards, as in our case study), significant computational load imbalances may arise. Even the same hardware, say two identical GPU boards, will not yield the same performance when accessed locally vs. over RPC. Moreover, different image seeds induce different image graphs, resulting in different computational costs for graph-cuts. Reassigning tasks between servers at runtime to achieve load balance requires complex task migration procedures (which may rely on costly mechanisms such as TCP endpoint migration [23] for transparent operation).

The dynamic scheduler, which is also multithreaded, implements a version of *List Scheduling (LS)* [24] and attempts to offset load imbalances by picking up an available server from a FIFO-ordered list and executing the RPC (or a local call, for the master node) to that node with a supergraph as a parameter. The server is removed from the list and, later on, when the supergraph cut call finishes, is inserted back into the list. Hence, the list of available servers grows and shrinks dynamically and, at times, may become empty, in which case no server is available for computation, and the master program dispatching the tasks gets blocked.

In contrast to static scheduling, the dynamic policy handles a supergraph as soon as a server is available and offers a more balanced mix of task overlaps, which in turn should contribute to

a smaller makespan. Optimal scheduling of independent, non-preemptable tasks to minimize the makespan is NP-hard [24]. However, a priori knowledge about supergraph cut processing times may improve the worst-case performance of the scheduling algorithm. For instance, sorting the task list in non-increasing order of processing times before scheduling and assigning the first available task to the first available server during scheduling, policy known as *Largest Processing Time First (LPT)*, is an effective way to minimize the worst-case makespan [24]. In our case, this a priori information is highly data-dependent and, thus, hard to obtain.

The scalability of the framework depends on two main components: the scalability of the master scheduler and that of the parallel graph cut routine processing the supergraphs. For the latter, our framework is limited by the scalability of the available software used (i.e., GridCut, CUDA NPPI, etc.). The scalability of the scheduler is influenced by the available resources on the master node and the network latency for remote communication. However, as our experiments show (see Section 5), achieving near real-time performance does not necessarily assume many computing nodes. Therefore, one can conclude that the master scheduler should not face scalability problems as long as it can use small-scale multiprocessor (6-8 cores) machines.

### 3.3.2. Supergraphs and network communication

Typical image sizes in the VOC dataset [2] used in our case study amount to approximately 80-100K pixels, and so are the sizes of the corresponding image graphs. Usually, for computational reasons, image segmentation algorithms like CPMC downsample images to half, so the resulting graph size sums up to approximately 160-200KB of memory (for 4-byte floats or integers). Packing several such graphs into a supergraph can enlarge the size of the RPC parameters even further. Moreover, library calls like NVIDIA's *nppiGraphcut* [7], [8] require five such large matrices as parameters, among others. As a result, the overall size of the RPC parameters sent over the network tends to be quite large and may negatively impact the performance of the call.

One possible solution to reduce the size of data transferred over the network is to pack several graphs into a larger supergraph (say, send a two-seed supergraph, i.e. two  $\lambda$ -supergraphs, instead of a single  $\lambda$ -supergraph parameter). Thus, the overhead of the send/receive network operations is amortized over larger amounts of data, and the transfer performance increases.

Overlapping communication with computation further helps by issuing two concurrent RPCs to the same server, which results in an overlap of the execution of the first call with the transfer of the parameters of the second call. Naturally, handling two concurrent RPCs requires multi-threaded server capabilities. Given that the TI-RPC Linux package does not include multi-threaded support for server side RPC (unlike the original Sun Microsystems/Oracle version), we had to implement a multithreaded RPC server as well, but this choice turned out to be beneficial in our case study for CPMC using NVIDIA's NPPI library (see Section 5).

## 4. Case Study: CPMC

The CPMC release [25] can use two parametric maximum flow algorithms [4], [5]. In our evaluation, we have chosen the pseudoflow algorithm [5] because it can also run "approximately" (see Section 2.1), i.e. without computing all the breakpoints, by accepting as argument a preset  $\lambda$ -schedule. Thus, the whole CPMC algorithm runs faster and the comparison to our framework is fair. The other option [26], [4] works only by computing all breakpoints online.

In our setup, CPMC iteratively solves a list of independent problems defined by a pair of foreground and background seeds and a  $\lambda$ -schedule passed to the pseudoflow algorithm. The problem solver is runs in MATLAB, while the pseudoflow solver is implemented in C. Thus, a trivially parallel solution can be easily implemented by using MATLAB’s *parfor* instruction that executes each iteration independently as a thread on one of the available CPU cores.

Motivated by the argument in Section 2.2, we compared the pseudoflow-based solution to that of the supergraph framework, which parallelizes the figure-ground stage of CPMC [1], in order to assess its utility as a tool toward real-time performance for image segmentation. To that end, we have employed two solutions. The first solution uses a cluster of GPUs managed by the framework as described in Section 3. The GPU cards have run the push-relabel implementation of the NVIDIA NPPI library [7], [8]. With no access to the library source code, we had to use the NVIDIA code as a black box, with no possibility to perform any kind of code optimization.

As already mentioned at the end of Section 3.1, we also experimented with GridCut [6], a parallel implementation of a popular augmenting path maximum flow algorithm [13], but due to lower performance figures, the following evaluation focuses on the GPU-based solution. However, in Section 5 we report the results of our GridCut-based evaluation of CPMC.

## 5. Experimental Evaluation

An extensive evaluation of our parallel parametric maximum flow solution can be found here [27]. In this section, we briefly present the main results of our work.

One of the results we would like to present here uses the VOC image subset and shows the performance of our parallel framework, described in Section 3, using locally accessible GPU cards. The experiments used  $\lambda$ -supergraphs (i.e., one seed supergraphs) with 20  $\lambda$  values. The first two rows of Table 1 show the performance of the GPU boards without using supergraphs at all (the method we called “batch” from Section 3). These results are the baseline for our next comparisons.

**Table 1.** Batch & supergraphs performance of NVIDIA’s push relabel implementation

Time	Min	Avg	Max
Tesla K40 batch	61.42 s	140.88 s	256.54 s
Titan Black batch	45.07 s	102.28 s	181.38 s
Tesla K40 no s-t swap	17.70 s	63.53 s	167.16 s
Titan Black no s-t swap	13.61 s	47.47 s	122.69 s
Tesla K40 s-t swapped	10.45 s	32.80 s	54.76 s
Titan Black s-t swapped	8.40 s	25.74 s	42.77 s
2 x Titan Black s-t swapped	4.53 s	13.93 s	22.77 s

The next two rows of Table 1 show the performance of supergraphs on two boards without the *s-t swap* optimization (see Section 3.2). The comparison to the batch results reveals that supergraphs reduce the average figure-ground segmentation latency more than two times for both types of boards. Please also note the minimal latencies, where the supergraph method yields at least three times lower figures.

The last three rows of Table 1 present the results of using properly *s-t swapped* supergraphs. On average, the Titan Black board takes roughly 78% of the K40 time and is almost 55% faster

than the trivially parallel, single-threaded algorithm (see [27]). Two Titan Blacks together outperform, on average, the trivially parallel algorithm on six threads.

Please note that *s-t swaps* cut almost to half the figure-ground segmentation latency for Tesla K40 and by roughly 46% for Titan Black, on average. Also, the maximum latencies of supergraphs without *s-t swaps* are 3, respectively, 2.86 times larger, showing how poor the degree of available parallelism can be, at times, if the source and sink are not properly swapped.

The impact of using two-seed supergraphs (see Section 3.1) can be assessed by looking at Table 2 (four-seed supergraphs have shown only marginally better figures). The improvements over one-seed supergraphs (last three rows of table 1) amount to approximately 10%, on average.

Finally, using two-seed supergraphs and small clusters of GPUs (ranging from three to five boards) revealed better overall times for the graph cuts of the figure-ground segmentation stage over CPMC using the pseudoflow solver on a 6-core machine. Table 3 that average times of the GPU cluster take roughly 72%, 62% and 55%, respectively, of the time needed to run the trivially parallel solution using the pseudoflow algorithm on a 6-core processor.

**Table 2.** Impact of seed supergraphs (2 seeds)

Time	Min	Avg	Max
Tesla K40	8.82 s	29.78 s	54.90 s
Titan Black	7.04 s	23.49 s	40.63 s
2 x Titan Black	4.03 s	12.79 s	22.45 s

**Table 3.** The performance of GPU clusters vs. the multi-core based solution

Time	Min	Avg	Max
Pseudoflow 6 threads	4.10 s	14.35 s	21.48 s
2 Titan Black + 1 K40	3.49 s	10.34 s	17.86 s
2 Titan Black + 2 K40	2.91 s	8.88 s	15.04 s
3 Titan Black + 2 K40	2.67 s	7.85 s	12.16 s

## 6. Conclusions

This paper has presented a solution to approximate parallel parametric maximum flow behavior for image segmentation problems based on supergraphs. This paper has also introduced a general, parallel framework that can run parametric maximum flow problems on various platforms (multi-core, GPU), either locally or distributed in a cluster, as instructed by a provably efficient dynamic scheduler.

The framework is also useful as an evaluation tool of the available parallel maximum flow implementations [6], [7], [8]. We report the results of using two such implementations, NVIDIA's GPU implementation of the push relabel maximum flow algorithm and GridCut, an implementation of an augmenting path maximum flow algorithm, together with CPMC [1], a state-of-the-art image segmentation algorithm, as a case study that points out the flexibility and utility of our framework. The evaluation has shown that authors' GPU-based solution achieves near real-time performance, practically without any segmentation accuracy loss.

**Acknowledgements.** The authors thank Academician Florin G. Filip for his support and encouragement in publishing this work.

## References

- [1] J. CARREIRA and C. SMINCHISESCU, *Constrained parametric min-cuts for automatic object segmentation*, Proceedings of 2010 IEEE International Conference on Computer Vision and Pattern Recognition, San Francisco, CA, USA, 2010, pp. 3241–3248.
- [2] *The PASCAL Visual Object Classes Homepage*, <http://pascallin.ecs.soton.ac.uk/challenges/VOC/>.
- [3] C. IONESCU, D. PAPAVAL, V. OLARU and C. SMINCHISESCU, *Human3.6M: large scale datasets and predictive methods for 3D human sensing in natural environments*, IEEE Transactions on Pattern Analysis and Machine Intelligence **36**(7) 2014, pp. 1325–1339.
- [4] G. GALLO, M. D. GRIGORIADIS and R. E. TARJAN, *A fast parametric maximum-flow algorithm and applications*, SIAM Journal of Computing, **18**, 1989, pp. 3–55.
- [5] D. S. HOCHBAUM, *The pseudoflow algorithm: a new algorithm for the maximum-flow problem*, Operations Research **56**, 2008, pp. 992–1009.
- [6] *Gridcut, fast graph-cuts for grids*. <http://gridcut.com>.
- [7] *NVIDIA CUDA Library Documentation*, [http://graphics.im.ntu.edu.tw/~bosslia/nvCuda\\_doxygen/html/group\\_\\_image\\_\\_labeling\\_\\_and\\_\\_segmentation.html](http://graphics.im.ntu.edu.tw/~bosslia/nvCuda_doxygen/html/group__image__labeling__and__segmentation.html).
- [8] *NVIDIA NPP Library Documentation*, <http://docs.nvidia.com/cuda/npp/index.html#abstract>.
- [9] A. GOLDBERG and R. E. TARJAN, *A new approach to the maximum-flow problem*, Journal of the ACM **35**(4), 1988, pp. 921–940.
- [10] D. GREIG, B. PORTEOUS and A. SEHEULT, *Exact maximum a posteriori estimation for binary images*, Journal of the Royal Statistical Society, Series B **51**(2), 1989, pp. 217–279.
- [11] L. FORD and D. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.
- [12] W. COOK, W. CUNNINGHAM, W. PULLEYBLANK and A. SCHRIJVER, *Combinatorial Optimization*, Wiley, New York, NY, 1998.
- [13] Y. BOYKOV and V. KOLMOGOROV, *An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision*, IEEE Transactions on Pattern Analysis and Machine Intelligence **26**(9), 2004, pp. 1124–1137.
- [14] Y. BOYKOV and O. VEKSLER, *Graph cuts in vision and graphics: Theories and applications*, in Handbook of Mathematical Models in Computer Vision, N. Paragios, Y. Chen and O. Faugeras, Eds., Springer, Boston, MA, pp. 79–96, 2006.
- [15] V. KOLMOGOROV, Y. BOYKOV and C. ROTHER, *Applications of parametric maxflow in computer vision*, Proceedings of 9<sup>th</sup> IEEE International Conference on Computer Vision, Rio de Janeiro, Brazil, 2007, pp. 1–8.
- [16] M. HUSSEIN, A. VARSHNEY and L. DAVIS, *On implementing graph cuts on CUDA*, In Proceedings of First Workshop on General Purpose Processing on Graphics Processing Units, Boston, MA, USA, 2007, pp. 1–10.
- [17] V. VINEETH and P. J. NARAYANAN, *CUDA cuts: fast graph cuts on the GPU*, Proceedings of 2008 CVPR Workshop on Visual Computer Vision on GPUs, Anchorage, AK, USA, 2008, pp. 1–8.
- [18] J. LIU and J. SUN, *Parallel graph-cuts by adaptive bottom-up merging*, Proceedings of 2010 IEEE International Conference on Computer Vision and Pattern Recognition, San Francisco, CA, USA, 2010, pp. 2181–2188.
- [19] O. JAMRISKA, D. SYKORA and A. HORNUNG, *Cache-efficient graph cuts on structured grids*, Proceedings of 2012 IEEE International Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, 2012, pp. 3673–3680.

- [20] J. YUAN, E. BAE, X. C. TAI and Y. BOYKOV, *A study on continuous max-flow and min-cut approaches*, Technical Report CAM-10-61, UCLA, 2010.
- [21] J. YUAN, E. BAE and X. C. TAI, *A study on continuous max-flow and min-cut approaches*, Proceedings of 2010 IEEE International Conference on Computer Vision and Pattern Recognition, San Francisco, CA, USA, 2010, pp. 2217–2224.
- [22] M. STICH, *NVIDIA*, Personal communication, <https://developer.nvidia.com/blog/author/mstitch/>.
- [23] V. OLARU and W. F. TICHY, *On the design and performance of kernel-level TCP connection endpoint migration in cluster-based servers*, Proceedings of 2005 IEEE International Symposium on Cluster Computing and the Grid, Cardiff, UK, 2005, vol. 2, pp. 1000–1007.
- [24] Edited by J. Y-T. LEUNG, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, Chapman & Hall/CRC Computer and Information Science Series, 2004.
- [25] J. CARREIRA and C. SMINCHISESCU, *Constrained parametric min-cuts for automatic object segmentation, Release 1*, <http://http://www.maths.lth.se/matematiklth/personal/sminchis/code/cpmc/index.html>.
- [26] M. BABENKO and A. GOLDBERG, *Experimental evaluation of a parametric flow algorithm*, Technical Report MSR-TR-2006-77, Microsoft, 2006.
- [27] *Extended experimental evaluation of our parallel parametric maximum flow solution using super-graphs*, <http://vision.imar.ro/appendix>.