A Parallel Framework for Parametric Maximum Flow Problems in Image Segmentation

Vlad Olaru\textsuperscript{1} \hspace{1cm} Mihai Florea\textsuperscript{1} \hspace{1cm} Cristian Sminchisescu\textsuperscript{2,1}

\textsuperscript{1}Institute of Mathematics of the Romanian Academy, \textsuperscript{2}Lund University

vlad.olaru@imar.ro, mihai.florea@my.fmi.unibuc.ro, cristian.sminchisescu@math.lth.se

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), that 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.

1 Introduction

Recent advances in image segmentation \cite{10} have led to improved accuracy over large and diverse image datasets \cite{4, 18}, by almost doubling the performance figures. This development has spurred the interest for the widespread use of image segmentation models (figure 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 figure 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 \cite{13, 16} independently for each seed seems appropriate.

Figure 1. (best seen in color) 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. Currently, as far as we know, there are no available parallel implementations of a parametric maximum flow algorithm. In this paper, we present the design of a general framework that can use existing parallel graph cut solutions such as GridCut \cite{1} or CUDA NPPI \cite{2, 3} 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 multi-core 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 scheduled dynamically to run on a heterogeneous collection of computing nodes. The dynamic scheduler efficiently adapts not only to the imbalances induced by the heterogeneous architectures used, but also 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) [10], that uses the parallel framework with NVIDIA’s GPU implementation of the well-known push-relabel maximum flow algorithm [14]. The comparison to a CPMC solution based on a sequential pseudoflow algorithm [16] in a trivially parallel setup (where instances of a segmentation problem are executed independently on several cores of a processor) helps us understand how close we can get to a real-time solution for image segmentation.

To summarize, the paper has three main contributions: (1) a parallel solution for parametric maximum 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: §2 describes shortly previous work on graph cuts and image segmentation for a clear understanding of the concepts. In §3 we present the framework solutions to parallelize the parametric maximum flow algorithms, using supergraphs. §4 presents the results of our case study evaluation of CPMC. We conclude in §5.

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 figure 2); the weights are called edge capacities. For image segmentation, the source and sink are associated with the two labels that will be used to distinguished 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 is to assign that pixel to either of the two classes of labels represented by the source and the sink.

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. [15] 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 and Fulkerson theorem [12], 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 [11]), including augment-
ing path (Ford-Fulkerson) and push-relabel 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 [7]. An extended view on the use of graph cuts in computer vision can be found in [8].

![Figure 3.](image)

**Figure 3.** (best seen in color) 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 border of the image. Each generated segment corresponds to a graph cut segmentation problem. For an entire image, multiple independent solutions are generated at the cost of heavy graph cut computations.

### 2.1 Parametric Max Flow & Image Segmentation

Parametric max flow algorithms [13, 21] are used in image segmentation to generate a set of hypotheses for plausible object segments in a given image. They are able to optimize energies where the unknowns are both 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. In the “monotonic” case where the factors multiplying the parameter \( \lambda \) in the unary (data) energy terms are all non-negative or non-positive, the optimal solutions are nested [13] and an efficient implementation of the parametric maximum flow algorithm is possible. The algorithms can either compute all breakpoints (an upper bound is the number of the graph nodes) or a subset of them. Either way, monotonicity makes the calculation significantly more efficient as earlier computations are reused. In practice, a preset list of parameter values (usually defined on a logarithmic scale), the so called \( \lambda \)-schedule, can be used instead of computing all the breakpoints, as empirical evaluations [10] 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 a grid in the image, whereas the background seeds are assigned on the borders of the image (see figure 3). 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. 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 standpoint, 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. Figure 4 shows the mean time in seconds taken by the pseudoflow algorithm [16] to yield figure-ground hypotheses for a set of 500 images from the VOC [4] 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 our 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 [17, 24], but don’t offer code. One available implementation is GridCut [1], 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 [7]. GridCut implements adaptive bottom-up merging [22] and cache efficient memory layout [20]. Other available implementations of max flow algorithms are GPU implementations [2, 3, 25, 26]. The NVIDIA NPPI library [3] implements a push-relabel algorithm [14].

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 (2.1)) by using a preset λ-schedule, and run a parallel maximum flow routine once for each λ 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 [13], 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

Besides the inability to optimize computations in the monotonic case, another shortcoming of the batch method is that running a single graph cut problem at the time, either on a multi-core processor or a GPU architecture, 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, consider running several graph cut problems simultaneously on the available parallel computing infrastructure. This is not straightforward, since the programming interfaces of parallel graph cut routines supplied by software like GridCut or CUDA NPPI take a single graph as parameter. The solution is to “knit” together several graphs representing different problems into a larger graph, that we call a “supergraph”, and to pass it on, 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: λ and seed level. A λ-supergraph combines together graphs for several λ values, whereas a seed supergraph combines several λ-supergraphs together. Usually, our structures combine an entire λ-schedule, the list of the λs that we run the parametric max flow with, but it is possible to have smaller supergraphs as well. In the case of seed supergraphs though, we use only λ-supergraphs constructed for an entire λ-schedule.

Combining two graphs into a supergraph can be simply done by inserting additional vertices “between” the two graphs and by linking them to the regular vertices from the left and right graph by means of zero-weight edges (see figure 5). Inductively, one can build arbitrarily large supergraphs out of individual graphs. Any minimum cut in a supergraph built like that is a union of the disjoint minimum cuts of the original graphs knitted together, plus some zero-weight edges that do not count towards the overall cost of the supergraph cut. Therefore, computing minimum energies associated with a pair of foreground-background seeds and a given λ 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.

Figure 5. Supergraph composed out of k individual graphs G1, G2, . . . , Gk (best seen in colors).

To see why this works, consider the following situation. Let S be a supergraph composed of k individual graphs G1, G2, . . . , Gk (see figure 5). Let’s assume C is a minimum cut in S, and C1, C2, . . . , Ck are the minimum cuts of the individual graph components. Suppose that one of the individual cuts, say C1, is not a minimum cut in G1. Then, there is a minimum cut C′1 in G1, different than C1. Since G1 is linked by means of zero weight edges to its neighboring graphs in S, any minimum supergraph cut that crosses G1 needs to include C′1, because the zero-weight crossing edges do not contribute to the overall cost, C′1 is the minimum cost path severing Gi into two partitions and the supergraph cut must somehow cross Gi. Therefore, there is another supergraph cut including C′1 of smaller cost than C, which contradicts the assumption that C is a minimum supergraph cut. Thus,
all of the $C_i$s must be minimum cuts in their corresponding $G_i$s.

Conversely, suppose that there is a supergraph cut $C'$ in $S$ whose cost is smaller than that of $C$. Then, there is another union of individual minimum cuts $C'_1 \cup C'_2 \cup \ldots \cup C'_k$ that compose $C'$. Since $C'_j$s are disjoint sets, it means that at least one of them from $C'$, 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 use the individual components of the minimum supergraph cut as individual minimum cuts. Thus, one can solve simultaneously 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 ends up in a better usage of the available computing power of the underlying hardware architecture.

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. 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 [2][3], run faster [23]. 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 [23]. So, if there are more such vertices for the sink than for the source, swapping them exposes more parallelism.

Choosing the source and the 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 obviously not a solution. Instead, we use a heuristic to choose the source and sink. For each regular vertex in the graph, the difference between the capacities of its source and sink edges is computed. Then, for each vertex, we separately add the positive and negative differences. If the number of negative differences turns out to be larger than that of positive ones, we apply the $s-t$ swap. This procedure leads to more hardware resources active per algorithm iteration and improves performance significantly (see [4,3]).

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 going to be encoded by means of supergraphs, as previously described. The resulting set of supergraphs can have a smaller size than the collection of seed problems if several such problems are expressed by means of seed supergraphs. Each resulted supergraph gets scheduled for parametric maximum flow processing on a given computing node (also called server), either locally or remote. Remote processing is achieved by means of Remote Procedure Calls (RPC) for the supergraph cut routines. The scheduling is controlled by a master node, which runs the image segmentation algorithm. The master node can 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 instead of performing an RPC. The resulting cluster of servers collaborating to solve the collection of seed problems may be heterogeneous, regardless whether operation is on CPUs or GPUs.

#### 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 by using a MATLAB parfor instruction with $n$ threads in which each parallel loop iteration $i$ gets allocated task $i \mod n$. All the tasks allocated to a class $i \mod n$, 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 among the completion times of the tasks, will be determined by the time needed to run the longest class $i$ of tasks $i \mod n$.

However, in a heterogeneous computing environment with different hardware architectures (for instance, different types of GPU boards, as in our case study), significant computational load imbalances may arise. Even the same hardware, say two GPU boards of the same kind, will not yield the same performance when accessed locally vs. over the network via RPC. Moreover, there is an intrinsic source
of imbalance in the segmentation problem, because different seeds of an image induce different image graphs and therefore different graph cut computational costs.

The dynamic scheduler, which is also multi-threaded, attempts to offset these load imbalances by picking up a server from a list of available ones in FIFO order and executing the RPC (or a local call, in the case of 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, this dynamic policy that handles a supergraph as soon as a server is available 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 known to be NP-hard [19]. However, a priori knowledge about the 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 [19]. In our case, this a priori information is hard to get though, because it is highly data dependent.

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 [4.6] and [4.8]), achieving near real-time performance doesn’t necessarily assume many computing nodes. Therefore, one can conclude that the master scheduler shouldn’t face scalability problems as long as it can use small-scale multiprocessor (6-8 cores) machines.

### 3.3.2 Supergraphs and Network Communication

Typical sizes of the images in the VOC dataset [4] used in our case study amount to roughly 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 [2, 3] require five such large matrices as parameters, among others. As a result, the overall size of the RPC parameters that have to be transferred over the network tends to be quite large and thus may have a negative impact on the performance of the call.

One possible solution to alleviate the consequences of transferring large amounts of data over the network is to minimize the number of transfers by packing several graphs into a larger supergraph (say, instead of sending a single \( \lambda \)-supergraph parameter, one might send a two seed supergraph, i.e. two \( \lambda \)-supergraphs). Thus, the overhead of the send/receive network operations gets amortized over larger amounts of data and the transfer performance increases.

One could also attempt to make better use of the available bandwidth by overlapping communication with computation. Issuing two concurrent RPCs to the same server 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 multi-threaded RPC server as well, but this choice turned out to be beneficial in our case study for CPMC using NVIDIA’s NPPI library (see [4.5]).

### 4 Case Study: CPMC

The CPMC release [9] can use two parametric maximum flow algorithms [13, 16]. In our evaluation, we have chosen the pseudoflow algorithm [16] because it can also run “approximately” (see [4.1]), 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 [6, 13] works only by computing all the breakpoints online.

In this 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 implemented in MATLAB, while the pseudoflow solver is implemented in C (hooked up with the MATLAB code by means of MEX libraries). 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 processor cores.

Motivated by the argument in [2.2] we compared the pseudoflow based solution to that of the supergraph framework, which parallelizes the figure-ground stage of CPMC [10], in order to assess its utility as a tool towards real-time performance for image segmentation. To that end, we have employed a cluster of GPUs managed by the framework as described in [1]. The GPU cards have run the push-relabel implementation of the NVIDIA NPPI library [2, 3]. 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.

4.1 Evaluation Setup

The evaluation has been driven on two types of HP workstations: three Z840 stations equipped with one Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz processor (6 cores) and 32GB RAM, and one Z420 station equipped with an Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50GHz processor (6 cores) and 32 GB RAM. We used five NVIDIA GPUs for the experiments, two Tesla K40 (one per Z840 station) and three Titan Black boards (two hosted on a Z840 machine, and the third on the Z420 station). A Tesla K40 board features 2880 cores clocked at 745MHz and 12 GB of RAM, while a Titan Black board has 2880 cores running at 889 MHz and 6 GB of RAM. We used CUDA 6.5 for the experiments. All the systems run Linux and are connected by Gigabit Ethernet.

Unless otherwise stated, all the experiments use a 500 image subset of the VOC2012 dataset and evaluate over this subset of images the minimum, maximum, and average time values, taken by the graph cuts of the CPMC [10] figure-ground segmentation stage that yields the segment hypotheses. This stage uses three different Segmenter methods for a total of 178 seeds and 20 λ values for each of these seeds (these are the default values, for details see [9]).

4.2 Performance of the Pseudoflow Algorithm

The first experiment attempts to assess the performance of the pseudoflow algorithm [16] on multi-core architectures. As already mentioned, this is a sequential parametric maximum flow algorithm and can be easily used with the parfor MATLAB instruction to implement a form of trivial parallelism on multi-core architectures. This parallel algorithm will represent our multi-core baseline performance.

| Time  | Min   | Avg   | Max   |
|-------|-------|-------|-------|
| 1 parfor thread | 13.35 s | 46.59 s | 72.59 s |
| 2 parfor threads | 8.14 s | 28.29 s | 42.70 s |
| 4 parfor threads | 5.35 s | 17.71 s | 26.23 s |
| 6 parfor threads | 4.10 s | 14.42 s | 21.48 s |

Table 1. Parametric pseudoflow performance on multi-core architectures.

We ran CPMC with the pseudoflow solver on a 6-core Z840 workstation with a λ-schedule of 20 values by varying the number of threads of the parfor instruction from 1 to 6. The results are shown in table 1 and represent the average, maximum and minimum times for performing the graph cuts of the CPMC figure-ground segmentation stage. These numbers appear also in figure 3 and show that increasing the number of cores doesn’t help in the long run.

4.3 Basic Performance of Push Relabel on GPUs

Our next experiment on the VOC image image subset attempts to shed some light on the local performance of our GPU cards (that is, without RPC) when using the methods described in 3. We ran the experiments on the Z840 workstations using λ-supergraphs (i.e., one seed supergraphs) with 20 λ values when calling the NVIDIA NPPI nppiGraphCut routine. First, we show in table 2 the performance of the GPU boards without using supergraphs at all. This is the method we called “batch” in 3 that calls iteratively the nppiGraphCut routine for every λ value in the schedule. These results are the baseline for our next comparisons.

| Time  | Min   | Avg   | Max   |
|-------|-------|-------|-------|
| Tesla K40 | 61.42 s | 140.88 s | 256.54 s |
| Titan Black | 45.07 s | 102.28 s | 181.38 s |

Table 2. Batch performance of NVIDIA’s push relabel implementation.

The comparison to table 3 shows that the use of supergraphs reduces the average figure-ground segmentation latency more than two times for both type of boards, thus making a strong case for the use of supergraphs. Please also note the minimal latencies, where the supergraph method yields at least three times lower figures.

| Time  | Min   | Avg   | Max   |
|-------|-------|-------|-------|
| Tesla K40 | 17.70 s | 63.53 s | 167.16 s |
| Titan Black | 13.61 s | 47.47 s | 122.69 s |

Table 3. Performance of NVIDIA’s push relabel implementation without s-t swaps.

Table 4 presents the results of using supergraphs that are properly s-t swapped supergraphs.

| Time  | Min   | Avg   | Max   |
|-------|-------|-------|-------|
| Tesla K40 | 10.45 s | 32.80 s | 54.76 s |
| Titan Black | 8.40 s | 25.74 s | 42.77 s |

Table 4. Performance of NVIDIA’s push relabel implementation using properly s-t swapped supergraphs.

Table 2 presents the results of using supergraphs without applying the s-t swap optimization (see 3.2). The comparison to table 2 reveals that the use of supergraphs reduces the average figure-ground segmentation latency more than two times for both type of boards, thus making a strong case for the use of supergraphs. Please also note the minimal latencies, where the supergraph method yields at least three times lower figures.
Titan Black, on average. Also noteworthy is that the maximum latency values of supergraphs that don’t use s-t swaps are 3, respectively 2.86 times larger, which shows how poor can be, at times, the degree of available parallelism if the source and sink are not properly swapped.

4.4 Impact of Seed Supergraphs

Using seed supergraphs (see §3.3.2) should achieve better usage of the underlying hardware. We assessed the performance of NVIDIA’s push relabel implementation on seed supergraphs when varying the number of seeds. Table 5 shows the results on a Z840 station for two seeds (four seed supergraphs have shown only marginally better figures). The comparison to one seed supergraph figures (table 4) proves roughly 10% improvements, on average.

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

Table 5. Impact of seed supergraphs (2 seeds).

4.5 RPC Performance

We also conducted experiments to evaluate the performance of our RPC version of the NVIDIA nppiGraphcut call to access remote GPU boards. The experiments have used a Tesla K40 board, Z840 workstations and λ-schedules of 20 values. The first row of table 6 presents the graph cut times to get the figure-ground segment hypotheses when using λ-supergraphs only. We also evaluated the optimizations discussed in §3.3.2 by running larger supergraphs (two seed supergraphs) to make a better usage of the available network. One can see that the difference between one and two seed performance in table 5 is larger than it is for local Tesla K40 calls (see tables §3.3.1 and §3.3.2). This difference can be accounted to better network usage in the case of larger supergraphs. The last rows of table 6 show that multi-threaded RPC servers improve the performance when two consecutive GPU RPCs are submitted to the same server in order to overlap communication with computation.

| Time             | Min   | Avg   | Max   |
|------------------|-------|-------|-------|
| 1 seed           | 17.28 s| 48.14 s| 74.99 s|
| 2 seeds          | 13.91 s| 42.71 s| 72.29 s|
| MT server (1 seed)| 14.59 s| 39.14 s| 62.51 s|
| MT server (2 seeds)| 11.14 s| 33.87 s| 59.82 s|

Table 6. Performance of remote GPU RPCs.

4.6 Parametric Max Flow using Supergraphs and Clusters of GPUs

Once the performance of the individual boards and mechanisms has been assessed, we proceeded to set up a cluster of GPUs that would act as a parallel, cluster-wide parametric maximum flow solver based on supergraphs and the insights gained from the previous experiments. Thus, we decided to use the Z840 workstation equipped with two Titan Black cards as a master node, since these cards are faster than the K40s and local access to them should be faster than by means of RPC. The other three stations, two Z840 equipped with one Tesla K40 card each and the Z420 machine hosting the third Titan Black board have been used to build the cluster of GPU servers. We tested with λ-schedules with 20 values and two-seed supergraphs, as using such supergraphs has shown the best performance. We varied the number of boards in the cluster from three to five and compared the results with those of CPMC using the pseudoflow solver on a 6-core Z840 machine. The figures are shown in table 7 and reveal better overall times for the graph cuts of the figure-ground segmentation stage based on clusters of GPUs. In terms of average times, the GPU cluster solutions 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.

| 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|

Table 7. The performance of GPU clusters vs. the multi-core based solution.

4.7 Graph Cut Accuracy

So far, our evaluations concerned running times, but CPMC and image segmentation algorithms in general need also to fulfill their primary goal of accuracy. The accuracy of an image segmentation algorithm is influenced by several factors, the graph cut calculation accuracy being an important one. Therefore we proceeded to an evaluation of the performance of the previously tested algorithms also in terms of accuracy.

An image segmentation accuracy measure is a similarity measure, defined according to the VOC challenge rules as the degree of overlap between the set of segments (pixel masks) S resulted from the image segmentation algorithm and the ground truth G (correct image segmentation masks delineated by hand, provided for reference). Alternative measures include the F-measure. The overlap measure is computed as follows:

$$\text{Overlap}(S, G) = \frac{|S \cap G|}{|S \cup G|}$$

The results are presented in table 8. Note that the overlap measure quantifies the accuracy of the whole method (so far, we have presented running times for graph cuts in the figure-ground segmentation stage of CPMC).

The difference between the accuracy of CPMC running the push-relabel algorithm on supergraphs mapped on a
cluster of GPU boards to that of the pseudoflow algorithm is around 1% and can be probably accounted to the fact that a pseudoflow algorithm is slightly less precise than a push-relabel algorithm. The point of this experiment is to show, however, that the NVIDIA implementation can provide reliable, high quality segments for CPMC. The comparison to the pseudoflow figure ascertains that.

The CPMC release [9] sets the default $\lambda$-schedule size to 20 values, as this choice has proven to yield the best results in the VOC challenges [4]. However, one can always trade off segmentation accuracy for improved running times. In this section, we present the results of halving the $\lambda$-schedule size to 10 values, as this choice has proven to yield the best results in the VOC challenges [4]. However, one can always trade off segmentation accuracy for improved running times. In this section, we present the results of halving the $\lambda$-schedule size to 10 values, as this choice has proven to yield the best results in the VOC challenges [4]. However, one can always trade off segmentation accuracy for improved running times.

4.8 Segmentation Accuracy vs. Speed Trade-Off

The CPMC release [9] sets the default $\lambda$-schedule size to 20 values, as this choice has proven to yield the best results in the VOC challenges [4]. However, one can always trade off segmentation accuracy for improved running times. In this section, we present the results of halving the $\lambda$-schedule size to 10 values, as this choice has proven to yield the best results in the VOC challenges [4]. However, one can always trade off segmentation accuracy for improved running times. In this section, we present the results of halving the $\lambda$-schedule size to 10 values, as this choice has proven to yield the best results in the VOC challenges [4]. However, one can always trade off segmentation accuracy for improved running times.

| Schedule Case | Avg. overlap (10 $\lambda$) |
|---------------|-----------------------------|
| Pseudoflow    | 0.729                       |
| Push-relabel  | 0.732                       |

Table 10. Image segmentation accuracy for 10 value $\lambda$-schedules.

The gain becomes even more important if one considers table 10 that depicts the segmentation accuracy (all of the GPU-based solutions yield at least 0.732 overlap, so we reported just one figure). Note that, even if the accuracy of the segmentation drops by roughly 1% compared to that reported in table 8, it practically equals that of the pseudoflow algorithm using twice as many $\lambda$ values. Thus, one can use fewer $\lambda$ values and get significantly improved running times at almost no accuracy loss.

4.9 Scheduling

4.9.1 Static vs. Dynamic Scheduling As pointed out in [3.3.1] we schedule parametric max flow computations on supergraphs using a FIFO ordered, dynamically managed list of computing servers. In contrast, a static solution would assign each iteration of a parfor MATLAB loop computing parametric max flows to a given server in the list. We instrumented four experiments using parfor loops with up to five threads and all of the GPU board combinations from [4.6]. Each parfor iteration was statically scheduled mod $n$, where $n$ was the number of threads/boards used, to run supergraph max flows on the GPU boards. We used two seed supergraphs and $\lambda$-schedules of 20 values. Table 11 shows the results. The comparison to the dynamic solution (tables 7 and 8 for 2 x Titan Black) reveals that dynamic scheduling improves average times over static solutions by roughly 10%, 32%, 34% and 30%, respectively.

| Schedule Case | Time | Min | Avg | Max |
|---------------|------|-----|-----|-----|
| 2 x Titan B. parfor | 4.58 s | 15.48 s | 27.68 s |
| 2 Titan B. + 1 K40 parfor | 4.70 s | 15.25 s | 30.11 s |
| 2 Titan B. + 2 K40 parfor | 4.14 s | 13.42 s | 27.98 s |
| 2 Titan B. + 3 K40 parfor | 3.57 s | 11.24 s | 18.90 s |

Table 11. Static scheduling.

4.9.2 Scheduling Efficiency Since finding the optimal schedule is proven to be NP-hard (see [3.3.1]), we aimed to find how much worse performs our dynamic scheduler compared to a theoretically superior algorithm such as LPT (see [3.3.1]). To match the simplest paradigm of parallel, non-preemptive scheduling for makespan minimization, namely that of Parallel and Identical Machines [19], we gathered the running times of our one seed supergraph cuts for the Z840 machine equipped with two identical Titan Black boards reported in the last row of table 5. We sorted the values in non-increasing order and applied the LPT algorithm offline. The results have shown that LPT makespans are 1.7% smaller, on average, than those of our dynamic scheduler (the minimal difference among the 500 images being 0.01% and the maximum 8.8%). These results show the efficiency and utility of our scheduler, given that LPT used a priori known information and thus cannot be run online, in our case.

5 Conclusions

In this paper we have presented a solution to approximate parallel parametric maximum flow behavior for image segmentation problems based on supergraphs. We have 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 sched-
The framework is also useful as an evaluation tool of the available parallel maximum flow implementations [1][2][3]. We report the results of using NVIDIA’s GPU implementation of the push relabel maximum flow algorithm together with CPMC [10], a state-of-the-art image segmentation algorithm, as a case study that points out the utility of our framework. The evaluation has shown that our solution achieves near real-time performance, practically without any segmentation accuracy loss.

Acknowledgements

This work was supported in part by CNCS-UEFISCDI under CT-ERC-2012-1, PCE-2011-3-0438, and JRP-ROFR-2014-16. The authors would like to thank NVIDIA Corporation for their generous hardware donation.

References

[1] Gridcut, fast graph-cuts for grids. http://gridcut.com
[2] Nvidia cuda library documentation. http://graphics.im.ntu.edu.tw/~bossliaw/nvCuda_doxygen/html/group__image__labeling_and__segmentation.html
[3] Nvidia npp library documentation. http://docs.nvidia.com/cuda/npp/index.html
[4] The pascal visual object classes homepage. http://pascallin.ecs.soton.ac.uk/challenges/VOC/
[5] S. Alpert, M. Galun, R. Basri, and A. Brandt. Image Segmentation by Probabilistic Bottom-Up Aggregation and Cue Integration. In Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition, 2007.
[6] M. Babenko and A. Goldberg. Experimental Evaluation of a Parametric Flow Algorithm. Technical Report MSR-TR-2006-77, Microsoft, 2006.
[7] 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):1124–1137, September 2004.
[8] Y. Boykov and O. Veksler. Graph cuts in vision and graphics: Theories and applications. In N. Paragios, Y. Chen, and O. Faugeras, editors, Handbook of Mathematical Models in Computer Vision. Springer, 2006.
[9] 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
[10] J. Carreira and C. Sminchisescu. Constrained Parametric Min-Cuts for Automatic Object Segmentation. In Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition, June 2010.
[11] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver. Combinatorial Optimization. John Wiley & Sons, 1998.
[12] L. Ford and D. Fulkerson. Flows in Networks. Princeton University Press, 1962.
[13] G. Gallo, M.D. Grigoriadis, and R.E. Tarjan. A fast parametric maximum-flow algorithm and applications. SIAM Journal of Computing, 18:3–55, February 1989.
[14] A. Goldberg and R.E. Tarjan. A new approach to the maximum-flow problem. Journal of the Association for Computing Machinery, 35(4):921–940, October 1988.
[15] 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):217–279, 1989.
[16] D. S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. Operations Research, 56:992–1009, July 2008.
[17] M. Hussein, A. Varshney, and L. Davis. On Implementing Graph Cuts on CUDA. In First Workshop on General Purpose Processing on Graphics Processing Units, Boston, MA, October 2007.
[18] C. Ionescu, D. Papava, 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):1325–1339, July 2014.
[19] Edited by J. Y-T. Leung. Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Chapman & Hall/CRC Computer and Information Science Series, 2004.
[20] O. Jamriska, D. Sykora, and A. Hornung. Cache-efficient Graph Cuts on Structured Grids. In Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition, 2012.
[21] V. Kolmogorov, Y. Boykov, and C. Rother. Applications of parametric maxflow in computer vision. In Proceedings of the 11th IEEE International Conference on Computer Vision, October 2007.
[22] J. Liu and J. Sun. Parallel Graph-cuts by Adaptive Bottom-up Merging. In Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition, June 2010.
[23] T. Stich. Nvidia. Personal communication.
[24] V. Vineeth and P. J. Narayanan. CUDA Cuts: Fast Graph Cuts on the GPU. In CVPR Workshop on Visual Computer Vision on GPUs, Anchorage, USA, June 2008.
[25] J. Yuan, E. Bae, X.-C. Tai, and Y. Boycov. A study on continuous max-flow and min-cut approaches. Technical Report CAM-10-61, UCLA, 2010.

[26] J. Yuan, E. Bae, and X.C. Tai. A study on continuous max-flow and min-cut approaches. In Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition, June 2010.