Parallel and Distributed Graph Cuts by Dual Decomposition

Petter Strandmark Fredrik Kahl
Centre for Mathematical Sciences, Lund University, Sweden
{petter,fredrik}@maths.lth.se

Abstract

Graph cuts methods are at the core of many state-of-the-art algorithms in computer vision due to their efficiency in computing globally optimal solutions. In this paper, we solve the maximum flow/minimum cut problem in parallel by splitting the graph into multiple parts and hence, further increase the computational efficacy of graph cuts. Optimality of the solution is guaranteed by dual decomposition, or more specifically, the solutions to the subproblems are constrained to be equal on the overlap with dual variables.

We demonstrate that our approach both allows (i) faster processing on multi-core computers and (ii) the capability to handle larger problems by splitting the graph across multiple computers on a distributed network. Even though our approach does not give a theoretical guarantee of speed-up, an extensive empirical evaluation on several applications with many different data sets consistently shows good performance. An open source implementation of the dual decomposition method is also made publicly available.

1. Introduction

Maximum flow algorithms are the foundations for many algorithms in computer vision. Examples include segmentation, image restoration, dense stereo estimation and shape matching, see [3, 4, 12, 20]. The approach is also useful for inferring the maximum *a posteriori* solution of a discrete MRF [4, 8].

Our work builds on the following two trends: the ubiquity of maximum flow computations in computer vision and the tendency of modern microprocessor manufacturers to increase the number of cores in mass-market processors. This implies that an efficient way of parallelizing maximum flow algorithms would be of great use to the community. Due to a result from Goldschlager et al. [7], there is little hope in finding a general algorithm for parallel maximum flow with guaranteed performance gains. However, the graphs encountered in computer vision problems are often sparse with much fewer edges than the maximum $n^2 - n$ in a graph with n vertices. The susceptibility to parallelization depends on the structure and costs of the graph.

1.1. Related work

There are essentially three types of approaches used in computer vision for solving the maximum flow/minimum cut problem:

(i) Augmenting paths. The most popular method due to its computational efficiency for 2D problems and moderately sized 3D problems with low connectivity (i.e., sparse graphs) is the augmenting path method by Boykov and Kolmogorov (BK) introduced in [3]. However, as augmenting path algorithms use nonlocal operations, they have not been considered as a viable candidate for parallelization. One way of making multiple threads cooperate is to divide the graph into disjoint parts. This is the approach taken by Liu et al. [15], in which the graph is split, solved and then split differently in an iterative fashion until no augmenting paths can be found. The key observation is that the search trees of the subgraphs can merged relatively fast. The more recent work [14] splits the graph into many pieces which multiple threads then solve and merge until only one remains and all augmenting paths have been found. In this paper the graph is also split into multiple pieces, but our approach differs in that we do not require a shared-memory model, which makes distributed computation possible.

(ii) Push-relabel. Successful attempts to parallelize maximum flow computations have focused on the push-relabel algorithm. The implementation in [5] is shown to perform well for huge 3D grids, where the entire graph could not fit within the physical memory of the computer. For 3D grids with high connectivity, push-relabel generally outperforms methods based on augmenting paths. [3] Another implementation that combines push-relabel with GPU processing is [22]. However, our tests produced incorrect results unless the amount of regularization was low using the authors' own implementation of [22].

(iii) Convex optimization. Another approach to parallel graph cuts is to formulate the problem as a linear program. Under the assumption that all edges are bidirectional, the

problem can then be reformulated as an ℓ_1 minimization problem. The work in [2] attempts to solve this problem with Newton iterations using the conjugate gradient method with a suitable preconditioner. Matrix-vector multiplications can be highly parallelized, but this approach has not proven to be significantly faster than the single-threaded algorithm in [3] for any type of graph, even though [2] used a GPU in their implementation. Convex optimization based on a GPU has also been used to solve continuous versions of graph cuts, e.g. [9]. However, the primary advantage of continuous cuts has been to reduce metrication errors due to discretization.

Graph cuts is also a popular method for multi-label problems using, e.g., iterated α-expansion moves. Such local optimization methods can naturally be parallelized by performing two different moves in parallel and then trying to fuse the solutions, as done in [13]. Therefore, in this paper, we will not consider multi-label problems.

1.2. Contributions

The main contribution of this paper is a parallel implementation of graph cuts based on the BK-method. The parallelization is obtained via dual decomposition [6]. The approach has a number of advantages compared to other parallelization methods:

1. The BK-method has shown to have superior performance compared to competing methods on a number of application problems [3], most notably sparse 2D graphs and moderately sized 3D graphs.

2. It is possible to reuse the search trees in the BK-method [10] which makes the dual decomposition approach attractive. Further, we show that the dual function can be optimized in integer arithmetic. This makes the dual optimization problem easier to solve.

3. Perhaps most importantly, we demonstrate good empirical performance with significant speed-ups compared to single-threaded computations, both on multi-core platforms and multi-computer networks.

Naturally, there are also some disadvantages:

1. There is no theoretical guarantee that the parallelization will be faster for every problem instance. Such bounds can be obtained with the push-relabel algorithm [5].

2. Our approach is only effective for graphs for which the BK-method is effective. For example, the push-relabel method is faster for problem instances with huge 3D graphs or highly connected 2D graphs [3].

2. Dual decomposition

Our approach to parallel graph cuts is based on splitting a large graph into two or more subgraphs. Each subgraph is assigned to its own computational thread or computer. The subgraphs partially overlap and they are forced to agree in the overlap by dual variables. Several iterations are needed to find the global solution, but the reuse of search trees in the augmenting path algorithm ensures efficiency. The technique of decomposing problems with dual variables was introduced by Everett [6] and it has been used in many different contexts, e.g., in control [19]. The application within computer vision that bears the most resemblance to this paper is the work of Komodakis et al. [11] where dual decomposition is used for computing approximate solutions to general MRF problems.

The basic idea is to convert a convex minimization problem of the form

$$\underset{\boldsymbol{x}_1,\boldsymbol{x}_2,\boldsymbol{y}}{\text{minimize}} \quad f_1(\boldsymbol{x}_1,\boldsymbol{y}) + f_2(\boldsymbol{x}_2,\boldsymbol{y}) \tag{1}$$

into an equivalent formulation

$$\begin{aligned} \underset{\boldsymbol{x}_1,\boldsymbol{x}_2,\boldsymbol{y}_1,\boldsymbol{y}_2}{\text{minimize}} \quad & f_1(\boldsymbol{x}_1,\boldsymbol{y}_1) + f_2(\boldsymbol{x}_2,\boldsymbol{y}_2) \\ \text{subject to} \quad & \boldsymbol{y}_1 = \boldsymbol{y}_2. \end{aligned} \tag{2}$$

This alternative formulation is then solved in dual form. The Lagrange dual function is [1]:

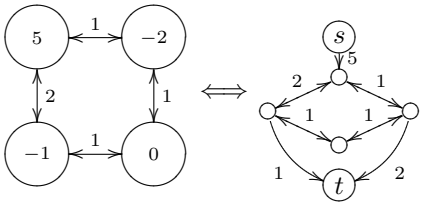
$$\begin{aligned} g(\boldsymbol{\lambda}) &= \min_{\substack{\boldsymbol{x}_1,\boldsymbol{y}_1 \\ \boldsymbol{x}_2,\boldsymbol{y}_2}} \Big(f_1(\boldsymbol{x}_1,\boldsymbol{y}_1) + f_2(\boldsymbol{x}_2,\boldsymbol{y}_2) + \boldsymbol{\lambda}^{\mathrm{T}}(\boldsymbol{y}_1 - \boldsymbol{y}_2) \Big) \\ &= \min_{\boldsymbol{x}_1,\boldsymbol{y}_1} \Big(f_1(\boldsymbol{x}_1,\boldsymbol{y}_1) + \boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{y}_1 \Big) \\ &\quad + \min_{\boldsymbol{x}_2,\boldsymbol{y}_2} \Big(f_2(\boldsymbol{x}_2,\boldsymbol{y}_2) - \boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{y}_2 \Big). \end{aligned} \tag{3}$$

The dual function is concave and evaluating the dual function amounts to solving two independent minimization problems. If $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are high-dimensional and the dimension of $\boldsymbol{y}$ is low, the two smaller problems will hopefully be solved much easier (and faster) than the original problem. Since they are independent of each other, they can be solved in parallel.

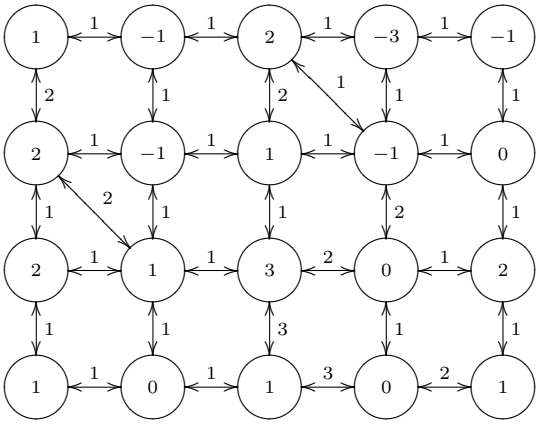
In order for g to be maximized, it needs to be evaluated several times. Kohli and Torr [10] have shown that the minimum cut problem is highly suitable for repeated evaluations when a small number of edge costs change. Next, we will see in detail how this helps us.

3. Graph cut as a linear program

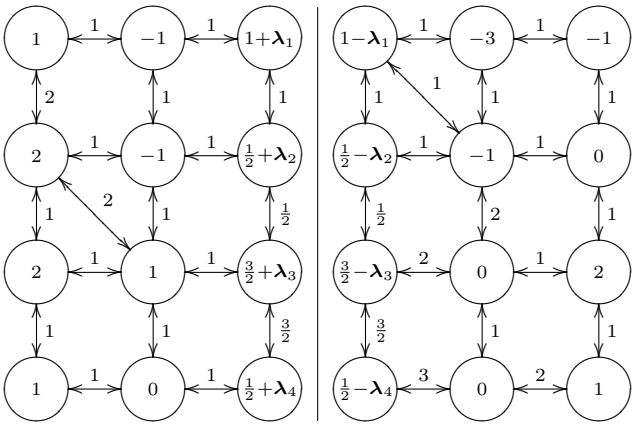
Finding the maximum flow, or, by duality, the minimum cut in a graph can be formulated as a linear program. Let $G = (V, \boldsymbol{c})$ be a graph where $V = \{s, t\} \cup \{1, 2, \ldots, n\}$ are the source, sink and vertices, respectively, and $\boldsymbol{c}$ the edge costs. A cut is a partition S, T of V such that $s \in S$ and $t \in T$. The minimum cut problem is finding the partition where the sum of all costs of edges between the two sets is



(a) Convention for this figure. Numbers inside the nodes indicate s/t connections, positive for s, negative for t.



(b) Original graph.



(c) Subproblems with vertices in M and N, respectively.

Figure 1: The graph decomposition into sets M and N. The pairwise energies in $M \cap N$ are part of both E_M and E_N and has to be weighted by $\frac{1}{2}$. Four dual variables $\boldsymbol{\lambda}_1 \dots \boldsymbol{\lambda}_4$ are introduced as s/t connections.

minimal. It can be formulated as, cf. [23],

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & \sum_{i,j \in V} \boldsymbol{c}_{i,j} \boldsymbol{x}_{i,j} \\ \text{subject to} \quad & \boldsymbol{x}_{i,j} + \boldsymbol{x}_i - \boldsymbol{x}_j \geq 0, \quad i,j \in V \\ & \boldsymbol{x}_s = 0 \\ & \boldsymbol{x}_t = 1 \\ & \boldsymbol{x} \geq 0. \end{aligned} \tag{4}$$

The variable $\boldsymbol{x}_i$ indicates whether vertex i is part of S or T ($\boldsymbol{x}_i = 0$ or 1, respectively) and $\boldsymbol{x}_{i,j}$ indicates whether the edge (i,j) is cut or not. The variables are not constrained to be 0 or 1, but there always exists one such solution, according to the duality between maximum flow and minimum cut, cf. [18, p. 119]. We write $\mathcal{D}_V$ for the convex set defined by the constraints in (4).

3.1. Splitting the graph

Now pick two sets M and N such that $M \cup N = V$ and $\{s,t\} \subset M \cap N$. We assume that when $i \in M \setminus N$ and $j \in N \setminus M$, $\boldsymbol{c}_{i,j} = \boldsymbol{c}_{j,i} = 0$. That is, every edge is either within M or N, or within both. See Fig. 1.

We now observe that the objective function in (4) can be rewritten as:

$$\begin{aligned} &\sum_{i,j \in V} \boldsymbol{c}_{i,j} \boldsymbol{x}_{i,j} = \\ &\sum_{i,j \in M} \boldsymbol{c}_{i,j} \boldsymbol{x}_{i,j} + \sum_{i,j \in N} \boldsymbol{c}_{i,j} \boldsymbol{x}_{i,j} - \sum_{i,j \in M \cap N} \boldsymbol{c}_{i,j} \boldsymbol{x}_{i,j}. \end{aligned} \tag{5}$$

Define

$$\begin{aligned} E_M(\boldsymbol{x}) &= \sum_{i,j \in M} \boldsymbol{c}_{i,j} \boldsymbol{x}_{i,j} - \frac{1}{2} \sum_{i,j \in M \cap N} \boldsymbol{c}_{i,j} \boldsymbol{x}_{i,j} \\ E_N(\boldsymbol{y}) &= \sum_{i,j \in N} \boldsymbol{c}_{i,j} \boldsymbol{y}_{i,j} - \frac{1}{2} \sum_{i,j \in M \cap N} \boldsymbol{c}_{i,j} \boldsymbol{y}_{i,j}. \end{aligned} \tag{6}$$

This leads to the following equivalent linear program analogous to (2):

$$\begin{aligned} \underset{\substack{\boldsymbol{x} \in \mathcal{D}_M \\ \boldsymbol{y} \in \mathcal{D}_N}}{\text{minimize}} \quad & E_M(\boldsymbol{x}) + E_N(\boldsymbol{y}) \\ \text{subject to} \quad & \boldsymbol{x}_i = \boldsymbol{y}_i, \quad i \in M \cap N. \end{aligned} \tag{7}$$

Here $\boldsymbol{x}$ is the variable belonging to the set M (left in Fig. 1c) and $\boldsymbol{y}$ belongs to N. The two variables $\boldsymbol{x}$ and $\boldsymbol{y}$ are constrained to be equal in the overlap. The Lagrange dual function of this optimization problem is:

$$\begin{aligned} &g(\boldsymbol{\lambda}) = \\ &\min_{\substack{\boldsymbol{x} \in \mathcal{D}_M \\ \boldsymbol{y} \in \mathcal{D}_N}} \left(E_M(\boldsymbol{x}) + E_N(\boldsymbol{y}) + \sum_{i \in M \cap N} \boldsymbol{\lambda}_i (\boldsymbol{x}_i - \boldsymbol{y}_i) \right) = \\ &\min_{\boldsymbol{x} \in \mathcal{D}_M} \left(E_M(\boldsymbol{x}) + \sum_{i \in M \cap N} \boldsymbol{\lambda}_i \boldsymbol{x}_i \right) + \\ &\min_{\boldsymbol{y} \in \mathcal{D}_N} \left(E_N(\boldsymbol{y}) - \sum_{i \in M \cap N} \boldsymbol{\lambda}_i \boldsymbol{y}_i \right). \end{aligned} \tag{8}$$

We now see that evaluating the dual function g amounts to solving two independent minimum cut problems. The

extra unary terms $\boldsymbol{\lambda}_i \boldsymbol{x}_i$ are shown in Fig. 1c. Let $\boldsymbol{x}^*, \boldsymbol{y}^*$ be the solution to (7) and let $\boldsymbol{\lambda}^*$ maximize the dual function g. Because strong duality holds, we have $g(\boldsymbol{\lambda}^*) = E_M(\boldsymbol{x}^*) + E_N(\boldsymbol{y}^*)$. [1] The solution to the subproblems may in general have multiple solutions, so to obtain a unique solution from these we always set our optimal $\boldsymbol{x}^*$ and $\boldsymbol{y}^*$ equal to 1, wherever possible.

3.2. Algorithm

For an optimal choice of dual variables, the constraints in (7) will be satisfied. Solving the original problem (7) then amounts to finding the maximum value of the dual function. Since the dual function is concave, it can be maximized with an ascent method. However, it is not differentiable, but we can find a supergradient[1] in the following way:

Lemma 1. *Let $\boldsymbol{\lambda}_0$ be given and let $\boldsymbol{x}^*$ be the optimal solution to $h(\boldsymbol{\lambda}_0) = \min_{\boldsymbol{x}} \left(f_1(\boldsymbol{x}) + \boldsymbol{\lambda}_0^{\mathrm{T}} \mathbf{f}_2(\boldsymbol{x}) \right)$. Then $\mathbf{f}_2(\boldsymbol{x}^*)$ is a supergradient to h at $\boldsymbol{\lambda}_0$.*

Proof. For any $\boldsymbol{\lambda}$, we have

$$\begin{aligned} h(\boldsymbol{\lambda}) &\leq f_1(\boldsymbol{x}^*) + \boldsymbol{\lambda}^{\mathrm{T}} \mathbf{f}_2(\boldsymbol{x}^*) \\ &= f_1(\boldsymbol{x}^*) + \boldsymbol{\lambda}_0^{\mathrm{T}} \mathbf{f}_2(\boldsymbol{x}^*) + (\boldsymbol{\lambda} - \boldsymbol{\lambda}_0)^{\mathrm{T}} \mathbf{f}_2(\boldsymbol{x}^*) \\ &= \min_{\boldsymbol{x}} \left(f_1(\boldsymbol{x}) + \boldsymbol{\lambda}_0^{\mathrm{T}} \mathbf{f}_2(\boldsymbol{x}) \right) + (\boldsymbol{\lambda} - \boldsymbol{\lambda}_0)^{\mathrm{T}} \mathbf{f}_2(\boldsymbol{x}^*) \\ &= h(\boldsymbol{\lambda}_0) + (\boldsymbol{\lambda} - \boldsymbol{\lambda}_0)^{\mathrm{T}} \mathbf{f}_2(\boldsymbol{x}^*), \end{aligned}$$

which is the definition of a supergradient. □

It follows from this lemma that $\boldsymbol{x}_i - \boldsymbol{y}_i$, for $i \in M \cap N$, is a supergradient to g. The algorithm for maximizing g becomes:

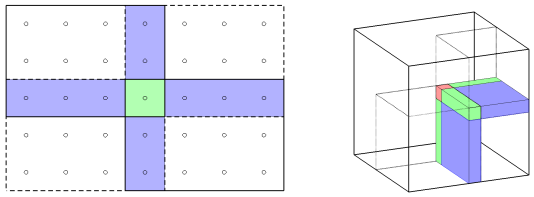
Start with $\boldsymbol{\lambda} = \mathbf{0}$
repeat
  Update $\boldsymbol{x}$ and $\boldsymbol{y}$ by evaluating $g(\boldsymbol{\lambda})$
  $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} + \tau(\boldsymbol{x}_i - \boldsymbol{y}_i)$.
until $\boldsymbol{x}_i = \boldsymbol{y}_i$, $i \in M \cap N$

This iterative scheme is very efficient, since the search trees of the already solved graphs can be reused. Only a small number of costs are changed between iterations and our experiments show that these subsequent max-flow computations can be completed within microseconds, see Table 1.

The step size τ needs to be chosen in each iteration. One possible choice is $\tau = 1/k$, where k is the current iteration number. We have found that this scheme and others appearing in the literature [1, 11] are a bit too conservative for our purposes. We achieved faster convergence with the empirical scheme introduced in Section 4.

3.3. More than two subproblems

Splitting a graph into more than two components can be achieved with the same approach. The energy functions analogous to (6) might then contain terms weighted by $1/4$ and $1/8$, depending on the geometry of the split. See Fig. 2.



(a) 2×2 (b) $2 \times 2 \times 2$

Figure 2: Splitting a graph into several components. The blue, green and red parts are weighted by $1/2$, $1/4$ and $1/8$, respectively.

4. Implementation

While the algorithm in the previous section works satisfactory, the actual implementation we ended up using for our experiments differs slightly. Instead of using a single step length τ, we associate each vertex in the overlap with its own step length τ_i. This is because different parts of the graph behave in different ways.

In each iteration, we ideally want to choose $\boldsymbol{\lambda}_i$ so that $\boldsymbol{x}_i = \boldsymbol{y}_i$; therefore, if $\boldsymbol{x}_i - \boldsymbol{y}_i$ changed sign from the previous iteration, our previous step was too large and we should move in the opposite direction with a reduced step length.

foreach $i \in M \cap N$ **do**
  if $\boldsymbol{x}_i - \boldsymbol{y}_i \neq 0$ **then**
    $\boldsymbol{\lambda}_i \leftarrow \boldsymbol{\lambda}_i + \tau_i(\boldsymbol{x}_i - \boldsymbol{y}_i)$
    if $\boldsymbol{x}_i - \boldsymbol{y}_i \neq$ *previous difference* **then**
      $\tau_i \leftarrow \tau_i / 2$
    end
  end
end

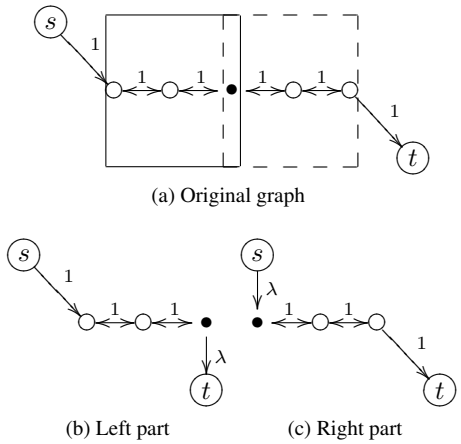
To handle cases like the one shown in Fig. 8, we also increase the step length if nothing happens between iterations. Empirical tests show that keeping an individual step length improves convergence speed for all graphs we tried. The extra memory requirements are insignificant.

4.1. Convergence

Some graphs may have problems converging to the optimal solution. This can occur for graphs admitting multiple solutions. Fig. 3 shows an illustrative example. While the proposed scheme will converge toward $\lambda = 1$ for this graph, it will not reach it in a finite number of iterations. If $\lambda \neq 1$, the two partial solutions will not agree for the vertex marked black. In practice, we observed this phenomenon for a few pixels when processing large graphs with integer costs.

One possible solution is to add small, positive, random numbers to the edge costs of the graph. If the graph only has integer costs, this is not a problem. Increasing edge costs

[1] completely analogous to subgradients for convex functions



(a) Original graph

(b) Left part (c) Right part

Figure 3: Convergence problem. The original graph (a) has multiple solutions, i.e., multiple minimum cuts. If $\lambda \neq 1$, the solutions for the two graphs (b) and (c) will not agree.

only increases the maximum flow, so the global maximum flow in the original graph is the integer part of the flow in the modified graph provided the sum of all values added is less than 1. However, there is an alternative way of handling graphs with integer costs:

Theorem 2. *If all the edge costs $\boldsymbol{c}_{i,j}$ are* even *integers, then there is an integer vector $\boldsymbol{\lambda}$ maximizing the dual function* (8).

Proof. The constraint sets $\mathcal{D}_M$ and $\mathcal{D}_N$ in (7) can be described by $\mathsf{A}\boldsymbol{z} \geq \mathbf{b}$, where A is an (integer) matrix, $\boldsymbol{z} = \begin{bmatrix} \boldsymbol{x}^{\mathrm{T}} & \boldsymbol{y}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}}$ and $\mathbf{b}$ is an (integer) vector. Therefore, the optimization problem (7) can be written as:

$$\begin{aligned} \underset{\boldsymbol{z}}{\text{minimize}} \quad & \mathbf{d}^{\mathrm{T}}\mathbf{z} \\ \text{subject to} \quad & \mathsf{A}\mathbf{z} \geq \mathbf{b} \\ & \boldsymbol{x}_i - \boldsymbol{y}_i = 0, \quad i \in M \cap N. \\ & \boldsymbol{z} \geq 0. \end{aligned} \tag{9}$$

Here, $\mathbf{d}$ is a vector describing the objective function in (7). It is integral because every edge cost $\boldsymbol{c}_{i,j}$ is an even integer. Forming the dual function for the entire problem, we get:

$$\tilde{g}(\boldsymbol{\nu}, \boldsymbol{\lambda}) = \min_{\boldsymbol{z}} \left\{ \mathbf{d}^{\mathrm{T}}\boldsymbol{z} + \boldsymbol{\nu}^{\mathrm{T}}(\mathbf{b} - \mathsf{A}\mathbf{z}) + \sum \boldsymbol{\lambda}_i(\boldsymbol{x}_i - \boldsymbol{y}_i) \right\}.$$

We also write $\mathsf{B}\boldsymbol{z} = 0$ for the equality constraints in this program. Because the optimization problem is a linear program, maximizing the dual function for $\boldsymbol{\nu} \geq 0$ can be written as a dual program:

$$\begin{aligned} \underset{\boldsymbol{\nu},\boldsymbol{\lambda}}{\text{maximize}} \quad & \mathbf{b}^{\mathrm{T}}\boldsymbol{\nu} \\ \text{subject to} \quad & \mathsf{A}^{\mathrm{T}}\boldsymbol{\nu} + \mathsf{B}^{\mathrm{T}}\boldsymbol{\lambda} \leq \mathbf{d} \\ & \boldsymbol{\nu} \geq 0. \end{aligned} \tag{10}$$

Since A is totally unimodular (TUM) (c.f. [18, Section 13.2]), so is A^{T}. Also B is TUM and $\mathbf{d}$ is integer, and thus the dual function $\tilde{g}$ has an integer optimum $(\boldsymbol{\nu}^*, \boldsymbol{\lambda}^*)$. Since we have $g(\boldsymbol{\lambda}) = \max_{\boldsymbol{\nu} \geq 0} \tilde{g}(\boldsymbol{\nu}, \boldsymbol{\lambda})$, we are done. □

Remark. The theorem does not necessarily hold if the costs are integers. The graph $s \xrightarrow{\infty} \circ \xrightarrow{1} \circ \xrightarrow{1} t$, split at the second node, provides a counterexample. The subproblems are $s \xrightarrow{\infty} \circ \xrightarrow{1} \circ \xrightarrow{1/2+\lambda} t$ and $s \xrightarrow{\lambda} \circ \xrightarrow{1/2} t$. We have $g(0) = g(1) = 1/2$, but $g(1/2) = 1$.

For a general graph with integer costs split into two pieces, we may multiply each edge by 2 and obtain an equivalent problem with an integer maximizer $\boldsymbol{\lambda}$. The graph may be split in more than two pieces in such a way that smaller costs than $1/2$ are used, as in Fig. 2. When setting up these problems we multiply every cost by 4 and 8, respectively, to ensure integer maximizers.

5. Experiments on a single machine

In this section we describe experiments performed in parallel on a single machine executed across multiple threads. We used the BK-method (v3.0 available for download) both for the single-threaded experiments and for solving the subproblems. All timings of the multi-threaded algorithm include any overhead associated with starting and stopping threads, allocation of extra memory etc. In all experiments we have only considered the time for actual maximum flow calculations, which means that the time required to construct the graphs is not taken into account. We note, however, that graph construction trivially benefit from parallel processing.

5.1. Image segmentation

We applied our parallel method for the 301 images in the Berkeley segmentation database [16], see Fig. 6 for examples. The segmentation model used was a piecewise constant model with the boundary length as a regulating term. The boundary length can be approximated with a neighborhood of edges around each pixel, usually of sizes 4, 8 or 16 in the two-dimensional case.

The relative times ($t_{\text{multi-thread}}/t_{\text{single}}$) using two computational threads are shown in Figs. 4 and 5. Since the images in the database are quite small, the total processing time for a single image is around 10 milliseconds. Even with the overhead of creating threads and iterating to find the global minimum, we were able to get a significant speed improvement for almost all images. The exceptions are discussed in Section 5.4.

Table 1 shows how the processing time varies with each iteration. In the last steps, very few vertices change and solving the maximum flow problems can therefore be done very quickly within microseconds.

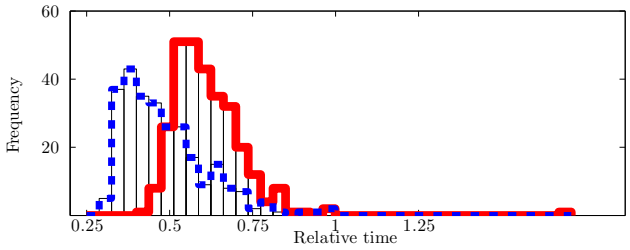


Figure 4: Relative times with 2 (red) and 4 (dotted blue) computational threads for the 301 images in the Berkeley segmentation database, using 4-connectivity. The medians are 0.596 and 0.455. See Sections 5.1 and 5.4.

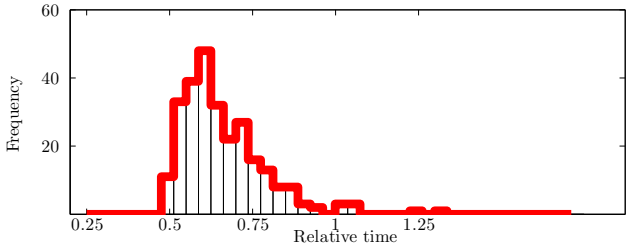


Figure 5: Relative times using 8-connectivity and 2 computational threads. The median is 0.628.

Figure 6: Examples from the Berkeley database [16].

It is very important to note that the problem complexity depends heavily on the amount of regularization used. That is, a segmentation problem in which boundary length is given a low cost is easy to solve and to parallelize. In the extreme case where no regularization is used, the problem reduces to simple thresholding, which of course is trivial to parallelize. Therefore, it is relevant to investigate how the algorithm performs with different amounts of regularization. We have done this and can conclude that our graph decomposition scheme performs well for a wide range of different settings, see Fig. 7. We see that the relative improvement in speed remains roughly constant over a large interval of different regularizations, whereas the absolute processing times vary by an order of magnitude.

When the number of computational threads increase, the computation times decrease as shown in Fig. 4.

Figure 8: "Worst-case" test. The left and right side of the image is connected to the source and sink, respectively. The edge costs are determined by the image gradient. All flow must be communicated between the two computational threads when splitting the graph vertically.

5.2. Stereo problems

The "Tsukuba" dataset (which we obtained from [17]) consists of a sequence of max-flow instances corresponding to the first iteration of α-expansions [4]. First, we solved the 16 problems without any parallelization and then, with two computational threads. The relative times ranged from 0.51 to 0.72, with the average being 0.61.

5.3. Three-dimensional graphs

We used the graph construction described in [12] with data downloaded from [17] to evaluate the algorithm in three dimensions. For the "bunny" dataset from [17] the relative time was 0.67 with two computational threads.

5.4. Analysis of limitations

We have tried to explore the limitations of the method by examining cases with (i) poor splits and (ii) poor speed-ups.

To see how our algorithm performs when the choice of split is very poor, we took a familiar image and split it in half from top to bottom as depicted in Fig. 8. We then attached the leftmost pixel column to the source and the rightmost to the sink. Splitting horizontally would have been much more preferable, since splitting vertically severs every possible s-t path and all flow has to be communicated between the threads. Still, the parallel approach finished processing the graph 30% *faster* than the single-threaded approach. This is a good indication that the choice of the split is not crucial for a speed improvement.

Figs. 4 and 5 contain a few examples ($< 1\%$) where the multi-threaded algorithm actually performs slower or almost the same as the single-threaded algorithm. The single example in Fig. 4 is interesting, because solving one of the sub-

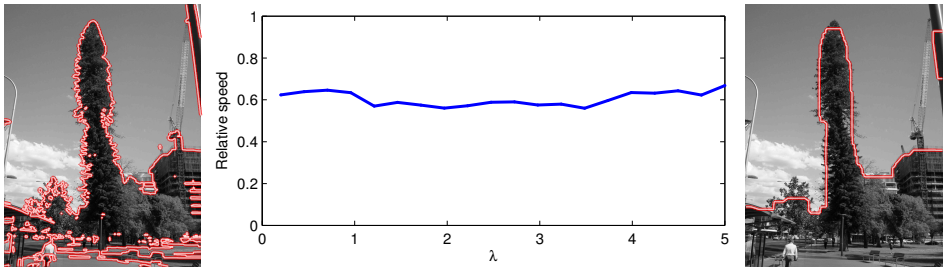


Figure 7: Relative improvement in speed with two computational threads when the regularization parameter changes. Although the processing time ranged from 230 ms to 4 seconds, the relative improvement was not affected.

Iteration	1	2	3	4	5	6	7	8	9	10	11
Differences	108	105	30	33	16	16	16	16	9	9	0
Time (ms)	245	1.5	1.2	0.1	0.08	0.09	0.07	0.15	0.06	0.07	0.47

Table 1: Detailed information about the processing time for each iteration for a 1152×1536 example image (shown in Fig. 7). The number of overlapping pixels ($M \cap N$) was 1536 (one column). Deallocating memory and terminating threads is the cause of the processing time increase in the last iteration. The advantage of reusing search trees is clearly seen in the short processing times after the first iteration.

graphs *once* takes significantly *longer* than solving the entire original graph. This can happen for the BK algorithm, but is very uncommon in practice. We have noted that slightly perturbing any of the problem parameters (regularization, image model, split position etc.) makes the multi-threaded algorithm faster also for this example.

The other slow examples have a simpler explanation: there is simply nothing interesting going on in one of the two halves of the graph, see e.g. the first image in Fig. 6. Therefore, the overhead of creating and deallocating the threads and extra memory gives the multi-threaded algorithm a slight disadvantage. The approach in [14] (using smaller pieces) is better suitable for these graphs.

6. Splitting across different machines

We now turn to another application of graph decomposition. Instead of assigning each part of the graph to a computational thread, one may assign each subgraph to a different *machine* and let the machines communicate the flow over a network.

Memory is often a limiting factor for maximum flow calculations. Using splitting we were able to segment 4-dimensional (space+time) MRI heart data with $95 \times 98 \times 30 \times 19 = 5.3$M voxels. The connectivity used was 80, requiring 12.3 GB memory for the graph representation. By dividing this graph among 4 (2-by-2) different machines and using MPI [21] for communication, we were able to solve this graph in 1980 seconds. Since only a small amount of data (54 kB in this case) needs to be transmitted between machines each iteration, this is an efficient way of processing large graphs. On the system we used[2], the communication time was about 7-10 ms per iteration, for a total of 68 iterations until convergence.

We also evaluated the algorithms for some of the big problems available at [17]. The largest version of the "bunny" dataset is $401 \times 396 \times 312 = 50$M with 300M edges was solved in 7 seconds across 4 machines. As a reference, a slightly larger version of the same dataset (not publicly available) was solved in over a minute with an (iterative) touch-and-expand approach in [12].

The largest data set we used was a $512 \times 512 \times 2317 = 607$M voxel CT scan with 6-connectivity. Storing this graph required 131 GB of memory divided among 36 ($3 \times 3 \times 4$) machines. We are not aware of any previous methods designed to handle graphs of this magnitude. The regularization used was low, which ensured convergence in 38 seconds with fairly even load distribution. Even with low regularization, the computation required 327 iterations.

Splitting graphs across multiple machines also saves computation time, even though the MPI introduces some overhead. For the small version of the "bunny" dataset, a single machine solved the problem in 268 milliseconds, while two machines used 152 ms. Four machines (2-by-2) required 105 ms. For the medium sized version the elapsed times were 2.3, 1.34 and 0.84 seconds, respectively.

It should be noted that in many cases the BK algorithm is not the fastest possible choice, especially for graphs with higher dimensionality than 2 and connectivity greater than the minimum [3]. However, the method described in this paper could just as easily be combined with a push-relabel algorithm better suited for graphs with 3 or 4 dimensions. Using a method optimized for grid graphs with fixed connectivity instead of the general BK would also reduce memory requirements significantly.

[2] LUNARC Iris, http://www.lunarc.lu.se/Systems/IrisDetails

7. Conclusions and future work

We have shown that it is possible to split a graph and obtain the global maximum flow by iteratively solving subproblems in parallel. Two applications of this technique were demonstrated:

- Faster maximum flow computations when multiple CPU cores are available (Section 5).
- The ability to handle graphs which are too big to fit in the computer's RAM, by splitting the graph across multiple machines (Section 6).

Good results were demonstrated even if the split severs many, or even all s-t paths of the graph (Fig. 8). Experiments with different amounts of regularization suggest that the speed-up is relatively insensitive to regularization (Fig. 7).

The technique was applied to different graphs arising in computer vision. Our experiments included surface reconstruction, stereo estimation and image segmentation with two, three and four dimensional data.

Methods based on push-relabel generally perform better than BK for large, high dimensional and highly connected graphs as discussed in [3]. Therefore, using our approach with push-relabel should be investigated in the future.

The source code used for the experiments in this paper has been made publicly available and may be downloaded from our webpage.[3]

Acknowledgments

This work has been funded by the Swedish Research Council (grant no. 2007-6476), by the Swedish Foundation for Strategic Research (SSF) through the programme Future Research Leaders, and by the European Research Council (GlobalVision grant no. 209480). We thank Einar Heiberg and Jane Sjögren for sharing MRI and CT data sets.

References

[1] D. P. Bertsekas. *Nonlinear programming*. Athena Scientific, 1999. 2, 4

[2] A. Bhusnurmath and C. J. Taylor. Graph cuts via ℓ_1 norm minimization. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 30(10):1866–1871, 2008. 2

[3] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, Sept. 2004. 1, 2, 7, 8

[4] Y. Boykov, O. Veksler, and R. Zabih. Markov random fields with efficient approximations. In *Conf. Computer Vision and Pattern Recognition*, 1998. 1, 6

[5] A. Delong and Y. Boykov. A scalable graph-cut algorithm for N-D grids. In *Conf. Computer Vision and Pattern Recognition*, 2008. 1, 2

[6] H. Everett. Generalized Lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 11:399–417, 1963. 2

[7] L. M. Goldschlager, R. A. Shaw, and J. Staples. The maximum flow problem is log space complete for P. *Theoretical Comput. Sci.*, 21(1):105–111, Oct. 1982. 1

[8] D. M. Greig, B. T. Porteous, and A. H. Seheult. Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society*, 1989. 1

[9] M. Klodt, T. Schoenemann, K. Kolev, M. Schikora, and D. Cremers. An experimental comparison of discrete and continuous shape optimization methods. In *European Conf. Computer Vision*, 2008. 2

[10] P. Kohli and P. H. S. Torr. Efficiently solving dynamic markov random fields using graph cuts. In *Int. Conf. Computer Vision*, 2005. 2

[11] N. Komodakis, N. Paragios, and G. Tziritas. MRF optimization via dual decomposition: Message-passing revisited. In *Int. Conf. Computer Vision*. IEEE, 2007. 2, 4

[12] V. Lempitsky and Y. Boykov. Global optimization for shape fitting. In *Conf. Computer Vision and Pattern Recognition*, Minneapolis, USA, June 2007. 1, 6, 7

[13] V. Lempitsky, C. Rother, S. Roth, and A. Blake. Fusion moves for markov random field optimization. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2009. To appear. 2

[14] J. Liu and J. Sun. Parallel graph-cuts by adaptive bottom-up merging. In *Conf. Computer Vision and Pattern Recognition*, 2010. 1, 7

[15] J. Liu, J. Sun, and H.-Y. Shum. Paint selection. *ACM Transactions on Graphics*, 28(3):1–7, 2009. 1

[16] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Int. Conf. Computer Vision*, 2001. 5, 6

[17] U. of Western Ontario. Max-flow problem instances in vision. http://vision.csd.uwo.ca/maxflow-data. 6, 7

[18] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization; Algorithms and Complexity*. Dover Publications, 1998. 3, 5

[19] A. Rantzer. Dynamic dual decomposition for distributed control. In *American Control Conference*, 2009. 2

[20] C. Rother, V. Kolmogorov, and A. Blake. Grabcut -interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, 23(3):309–314, 2004. 1

[21] M. Snir and S. Otto. *MPI — The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998. 7

[22] V. Vineet and P. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *Computer Vision and Pattern Recognition Workshops, CVPRW*, June 2008. 1

[23] Wikipedia. Max-flow min-cut theorem — Wikipedia, the Free Encyclopedia. [Online; accessed 9-January-2010]. 3

[3] http://www.maths.lth.se/~petter