A Scalable Graph-Cut Algorithm for N-D Grids

Andrew Delong
University of Western Ontario
andrew.delong@gmail.com

Yuri Boykov
University of Western Ontario
yuri@csd.uwo.ca

Abstract

Global optimisation via s-t graph cuts is widely used in computer vision and graphics. To obtain high-resolution output, graph cut methods must construct massive N-D grid-graphs containing billions of vertices. We show that when these graphs do not fit into physical memory, current max-flow/min-cut algorithms—the workhorse of graph cut methods—are totally impractical. Others have resorted to banded or hierarchical approximation methods that get trapped in local minima, which loses the main benefit of global optimisation.

We enhance the push-relabel algorithm for maximum flow [14] with two practical contributions. First, true global minima can now be computed on immense grid-like graphs too large for physical memory. These graphs are ubiquitous in computer vision, medical imaging and graphics. Second, for commodity multi-core platforms our algorithm attains near-linear speedup with respect to number of processors. To achieve these goals, we generalised the standard relabeling operations associated with push-relabel.

1. Introduction

Maximum flow algorithms are the workhorse behind many global optimisation methods popular in computer vision, such as graph-cuts [4, 21] and quadratic psuedo-boolean optimisation (QBPO) [15, 18]. Common applications include image restoration, segmentation, stereo, and multiview reconstruction. We introduce a new maximum flow algorithm to improve the main performance bottleneck at the heart of these global optimisation methods.

There are many problems for which global methods are currently impractical despite potential for high quality results. High-resolution biomedical imaging and multiview reconstruction involve massive 3D or 4D grids containing billions of vertices. Data for these problems may not fit into cache, physical memory, or even into the virtual address space. When these limits are reached, current maximum flow algorithms hit a performance wall. Just as importantly, good algorithms for vision have not been parallelised [4].

For example, banded [17] and hierarchical [19] graph-cut approximations are directly motivated by the limited scalability of [4] and, where applicable, heuristics such as "touch-expand" [16] are currently needed to generate high quality global solutions.

By definition, global graph-cut methods must consider the entire problem simultaneously. Inside each maximum flow algorithm, however, are basic steps to propagate flow towards the sink. For this paper we call a flow propagation scheme *scalable* in a practical sense if its associated steps

a) allow control of *locality* so that many algorithm steps can be made within memory constraints, and
b) are *parallelisable* so that multiple steps can be carried out asynchronously.

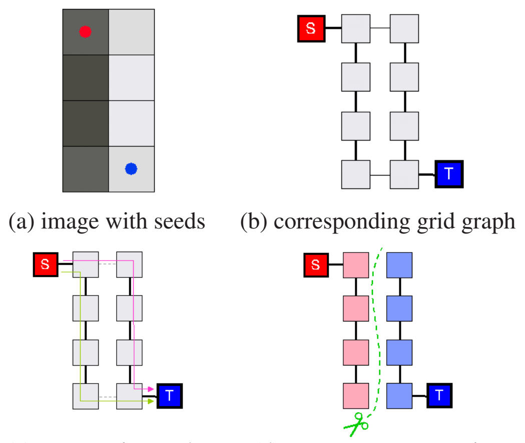
The popular maximum flow algorithm by Boykov and Kolmogorov [4] satisfies neither of these criteria, and none of the other practical algorithms we surveyed satisfy both simultaneously [1, 2, 8, 14, 6].

Our new maximum flow algorithm is of the well-known "push-relabel" variety [14] and is

1. scalable in the above sense,
2. practical on commodity shared-memory multiprocessor workstations, and
3. strongly polynomial, but *only efficient on grid graphs*.

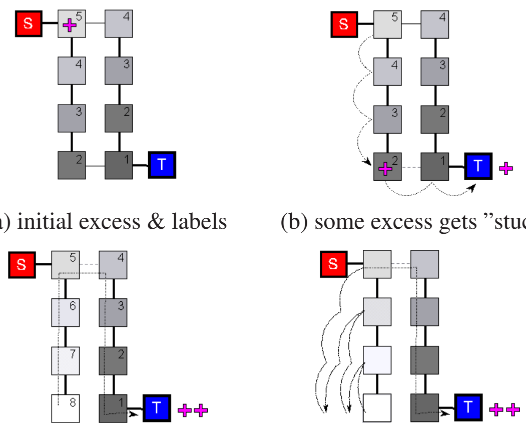
An inherent benefit of our approach is that, at the expense of some parallelisability, synchronisation overhead can be coarsened arbitrarily. This is crucial for systems with 2–8 processors because other parallel methods [1, 2] require up to 8 processors just to overtake a sequential implementation and they cite fine-grained locking as the culprit.

In Section 2 we review push-relabel methods, paying special attention to two effective heuristics: the *global relabel* and *gap relabel* operations. Our algorithm in Section 3 is distinct primarily because it relies on generalised versions of these operations which we term *region relabel* and *region gap relabel*. Our 3D segmentation experiments in Section 4 show that our algorithm is orders of magnitute faster than [4] and [14] when the graph is too large for physical memory (Figure 13). These 3D graphs are also representative of those needed for multiview reconstruction. Section 5 discusses our C++ library and future work.



(a) image with seeds (b) corresponding grid graph (c) augmenting paths (d) s-t cut = segmentation

Figure 1. Simple example of graph-cut image segmentation. (a) Image with seeds. (b) Corresponding capacitated network. (c) An *augmenting paths* algorithm finds two paths and saturates [13]. (d) Max-flow implies minimum cut, thus segmenting image in (a).



(a) initial excess & labels (b) some excess gets "stuck" (c) new maximal labels (d) via repeated PUSH/RELABEL

Figure 2. How *push-relabel* [14] might solve the problem in Fig. 1. (a) Initialise to maximal labels. When excess gets stuck (b), new maximal labels are ideal (c), but RELABEL works eventually (d).

2. Background

The maximum flow problem is to maximise the amount of some scalar quantity transported from a designated source s to a designated sink t, where there exists some capacitated network connecting s and t indirectly. The network is a digraph $\mathcal{G} = (V, A, c)$ of vertices V, directed arcs A, and capacities $c(v, w)$. Each capacity suggests a limit on how much quantity (flow) can be transported across arc $(v, w) \in A$. In computer vision, a simple application [3] is illustrated in Figure 1.

The running time of a maximum flow algorithm is at least proportional to the number of arcs across which flow is sent. So, maximum flow algorithms must balance two competing factors in performance: flow should reach its destination via short paths, but finding short paths is expensive. Strategies for finding such paths fall into either *augmenting path* or *preflow-push* categories.

Augmenting path algorithms [4, 11, 12, 13] are a conservative approach in that they first search for a complete $s \to t$ path and then 'augment' the entire path by sending enough flow to saturate it, effectively removing the path from future consideration. The augmentation step operates on complete $s \to t$ paths which do not meet our locality goals nor is the augmentation step easy to parallelise. The effort by [4] at 'tree recycling' improves performance on many useful graphs, but does not scale in the sense we defined earlier.

Preflow-push algorithms do not operate on paths directly. Instead, flow is optimistically 'pushed' across a single arc at a time. The particular arc chosen is based on purely local information, making this class of algorithms a good starting point for scalability. Our algorithm is an extension of one called *push-relabel*.

2.1. Push-relabel algorithms

The first push-relabel algorithms proposed in 1988 by Goldberg and Tarjan [14] offer the most robust performance for general graphs [1, 8, 10] but are surpassed by [4] for many graphs used in vision, particularly 2D grids. The more recent pseudoflow algorithm shares much in common with preflow-push approaches and was recently shown to be the fastest on many general graphs [6].

The term 'relabel' refers to the fact that pushes (and thereby paths) are determined by labels $d(v) \in \mathbb{Z}_+$ associated with each $v \in V$. The idea is that $d(v)$ should roughly indicate v's distance to the final destination of flow t, where $d(t) = 0$. Flow is only pushed 'downhill' across an arc (v, w) where $d(v) > d(w)$, since w has a shorter expected path to t than v does. If flow gets stuck at a dead-end $u \neq t$ then clearly $d(u)$ was wrong and u needs to be 'relabeled'. Relabeling can be thought of as a backtracking mechanism to correct for overly optimistic local decisions.

Push-relabel tracks the propagation of flow by maintaining a list of all 'active' vertices—those that have received flow, but have not yet passed it downhill. The accumulated flow at vertex v is called its *excess* $e(v)$, and v is active when $e(v) > 0$. Every time δ units of flow are pushed across arc (v, w), any residual (unused) capacity of $c(v, w)$ is remembered as $c'(v, w)$. If $d(v) > d(w)$ and $c'(v, w) > 0$ then flow can be pushed from v to w and arc (v, w) is called *admissible*. If v is active but has no admissible arcs, then v is a dead-end and needs to be relabeled by increasing $d(v)$. Notice that $d(v)$ is always a lower bound on the true minimum distance from v to t. Figure 2 illustrates this process.

Figure 3 lists the PUSH and RELABEL operations, and the basic algorithm from [14] upon which ours will be based. We use the conventional $n = |V|$ and $m = |A|$.

```
PUSH($v, w$)   $\triangleright$ $v$ active
    $\delta \leftarrow \min\{e(v), c'(v, w)\}$
    $e(v) \leftarrow e(v) - \delta$,     $c'(v, w) \leftarrow c'(v, w) - \delta$
    $e(w) \leftarrow e(w) + \delta$,     $c'(w, v) \leftarrow c'(w, v) + \delta$

RELABEL($v$)   $\triangleright$ $v$ active, all $(v, w)$ inadmissible
    $d(v) \leftarrow \min_{(v,w)}\{d(w) + 1 \mid c'(v, w) > 0\}$

GENERIC-PUSH-RELABEL
    $c'(v, w) \leftarrow c(v, w)$
    $d(s) \leftarrow n$,  $d(V - \{s\}) \leftarrow 0$
    $e(s) \leftarrow \infty$,  $e(V - \{s\}) \leftarrow 0$
    PUSH($s, v$) for all $c'(s, v) > 0$
    while exists active $v \in V - \{s, t\}$
        do if exists $(v, w)$ admissible
            then PUSH($v, w$)
            else RELABEL($v$)
    return $e(t)$
```

Figure 3. GENERIC-PUSH-RELABEL runs in $O(n^2m)$ time.

The generic algorithm outlined above has sequential variants that run in $O(n^2m)$, $O(n^3)$ [14], $O(nm \log \frac{n^2}{m})$ [20], and a parallel $O(n^2 \log n)$ version exists assuming $O(n)$ processors [14]. Unfortunately, none of these algorithms are of practical interest unless global heuristics are used [7, 8, 10, 14], and these heuristics inhibit scalability.

2.2. Relabeling heuristics

Basic push-relabel is defined in terms of purely local steps. This has a catastrophic affect on practical performance because, as arcs saturate (i.e. $c'(v, w) = 0$), long paths develop where the labels are totally misleading to the local decisions. Repeated RELABEL operations are a terribly slow (but correct) way to fix large labeling problems.

The two relabeling heuristics described in this section are critical for practical implementations of push-relabel. However, they are strictly *global* heuristics, and their behaviour is bound to the global problem size.

GLOBAL-RELABEL [14] throws away the current labels $d(v)$ and computes a *maximal* labeling for the current c'. That is, it sets $d(v) = \min\{d_G(v, t), d_G(v, s) + n\}$ for all $v \in V$ where geodesic distance $d_G(v, w)$ measures the shortest path (number of non-saturated arcs) from v to w. For a short time thereafter, flow is guaranteed to push towards t if possible, or towards s when t is unreachable. Global relabel is implemented via backwards breadth-first search from t and then from s, taking $O(n + m)$ time. For best performance (and to ammortise the cost), global relabel should be done after roughly hn RELABEL operations [8]. Figure 4 illustrates the affect of h on total running time for a 3D segmentation problem.

GAP-RELABEL(g) [7, 10] is based on the observation that if no vertex has label $g < n$ then any v where $g < d(v)$ can never reach t (i.e. g is a 'gap' in the global labeling).

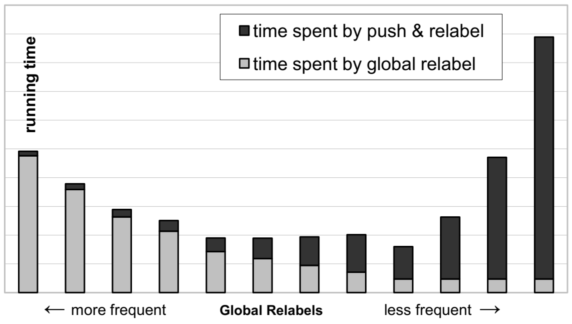


Figure 4. Total running time of sequential push-relabel as frequency of global relabeling (h) varies. Dark and light bars convey the proportion of time spent in push-relabel and global relabel phases respectively. Better labels mean less pushing, but more overhead (left side). Optimal h depends on the specific problem.

Thus, when such g is detected, all labels $g < d(v) < n$ are increased to $n + 1$, which is a conservative lower bound on their maximal label (i.e. what global relabel would compute). Gaps are detected by maintaining a bucket for every label $0 < \ell < n$ containing a global linked list of all v with $d(v) = \ell$. When a bucket becomes empty, a gap relabel is triggered.

Interestingly, [14] notes that generic push-relabel can be modified to maintain maximal labels at all times, without affecting complexity. However, they found that maximal labelings are not worth the computational effort and the aforementioned heuristics result in much better performance.

2.3. Practical parallelism, memory locality

Anderson and Setubal [1] introduced the first practical parallel version of push-relabel. Each thread maintains a local queue of active vertices, and locks v before every RELABEL(v) and also w before each PUSH(v, w). Their main innovation, however, is to interleave a parallel version of global relabeling into the generic computation. They call this *wave relabeling*, and it involves extra bookkeeping and careful locking of vertices during a breadth first search.

More recently Bader and Sachdeva [2] found that cache hierarchies fail to accommodate the memory access pattern of push-relabel and (global) wave relabel, hindering performance. They propose a "cache-aware" memory layout for the graph, and combine wave relabeling with gap relabel and the highest-first ordering suggested in [8, 14].

Table 1 contrasts the most relevant maximum flow algorithms for computer vision. The two parallel versions [1, 2] give good relative speedup as processors are added but absolute speedup over sequential push-relabel is difficult to obtain. They cite fine-grained synchronisation overhead added to each PUSH and RELABEL as the reason, meaning parallel implementations start at a disadvantage.

	practical	**parallel**	**local**
Dinitz AP [11]			
generic PR [14]		√	√
global heuristic PR [8, 14]	√		
tree recycling AP [4]	√		
pseudoflow [6]	√		
wave relabel PR [1]	√	√	
cache-aware PR [2]	√	√	
region PR (this paper)	√	√	√

Table 1. Review of max-flow scalability traits in computer vision (PR = push-relabel, AP = augmenting path).

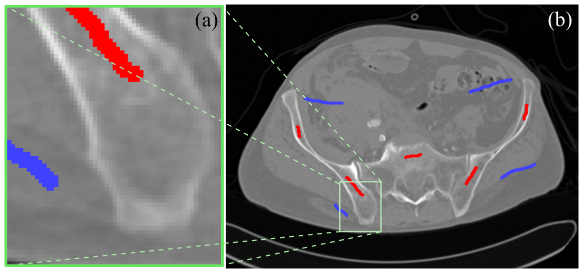


Figure 5. When segmentation problem (a) is solved independently, optimal running time involves many global relabels (i.e. many global relabels were effective). If (a) is part of a larger problem (b) then, at this scale, global relabel is rarely worth the overhead.

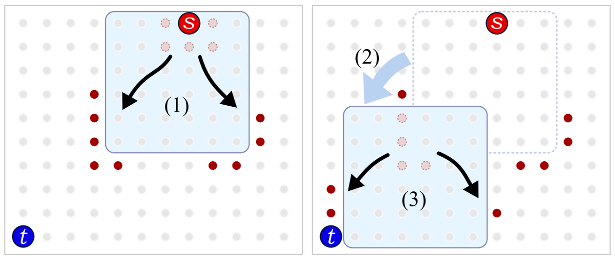


Figure 6. Region push-relabel focuses on a region until it has been purged, or *discharged*, of excess flow (1). Excess accumulates on the boundary. The region becomes inactive, so select a new active region (2) and repeat the process (3).

3. Region push-relabel

We were motivated by situations such as Figure 5 where, at a moderate problem scale, frequent global relabeling was most effective, but at large scales it was rarely worth the effort beyond initialising d. Poor labelings develop at each scale regardless of total problem size, but global relabel, as a tool, becomes inappropriate for dealing with intermediate labeling issues. The best performance on immense graphs is thus to rely on many, many RELABEL operations until global gaps occur. We could update labels better at intermediate scales if only we had the right tool for the job.

We call our algorithm *region push-relabel*. At a high level, ours follows Goldberg and Tarjan's 'discharge' variant of push-relabel [14]. Their idea is to select an active vertex $v \in V$, and repeatedly call PUSH(v, w) and RELABEL(v) until all excess $e(v)$ has been pushed to v's neighbours. In our variant, we instead select an active *region* $R \subseteq V - \{s, t\}$ and try to push all excess to neighbours outside R while only modifying labels inside R. (A region is 'active' if it contains at least one active vertex.) Figure 6 illustrates this scheme on a 2D grid.

To understand how our algorithm relates to generic and heuristic push-relabel, consider the following two extremes. If we only allow one region, $R = V - \{s, t\}$, our algorithm reduces to heuristic push-relabel since we can just use global/gap relabeling inside R to speed things up. If we only allow regions $R_i = \{v_i\}$, then our algorithm is equivalent to the discharge variant of push-relabel without any global heuristics.

Figure 7 illustrates how our algorithm fills the range between these two cases. For all region sizes less than the full graph, global heuristics do not apply. Repeated PUSH/RELABEL inside the current region is a terribly slow but perfectly correct approach. The overall algorithm would then just be generic push-relabel with a particular ordering on processing vertices, and thus have $O(n^2 m)$ complexity (or $O(n^3)$ on nearest-neighbour grid graphs).

The novelty of our approach is in how we discharge regions more efficiently than just PUSH/RELABEL. We introduce two non-global heuristics we call *region relabel* and *region gap relabel* to speed up region discharge in the same manner that global/gap relabel speed up generic push-relabel. Our framework is also trivial to parallelise when non-intersecting active regions are readily available, as suggested by Figure 7.

3.1. Discharging regions efficiently

We redefine global relabel and gap relabel to operate on an arbitrary subset of vertices. At two extremes, our new relabeling operations reduce to the three standard ones:

GLOBAL-RELABEL	$\Leftrightarrow$	REGION-RELABEL$(V - \{s, t\})$
GAP-RELABEL(g)	$\Leftrightarrow$	REGION-GAP-RELABEL$(V - \{s, t\}, g)$
RELABEL(v)	$\Leftrightarrow$	REGION-RELABEL$(\{v\})$ or REGION-GAP-RELABEL$(\{v\}, d(v) - 1)$

When called on some region $R \subseteq V - \{s, t\}$, our operations try to increase labels in R as much as they can while keeping the labeling consistent. This means that, for correctness, each $d(v)$ inside R should be no greater than the (maximal) labeling that a global relabel would have computed. Large regions can 'see' farther than small ones and thereby have potential to increase labels more dramatically.

The easiest way to think of how this works is by analogy to how RELABEL(v) makes local decisions: to increase $d(v)$, each $c'(v, w_i)$ is inspected and, if residual capacity remains, then from a purely local vantage there is *possibly* a

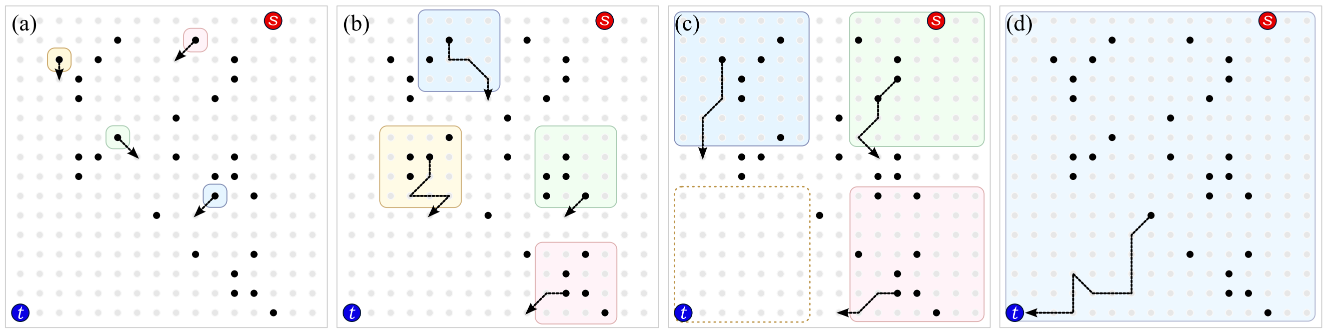


Figure 7. Our *region push-relabel* fills the range between variants of generic push-relabel (a) and global heuristic push-relabel (d) [14]. Region size impacts parallelism, memory locality, and propagation of flow (see Fig.8 for theoretical running times corresponding to a, b, c, d above). Above also conveys the effect on parallelism. Supposing there are 4 CPUs: (a) per-vertex regions, poor labelings, heavy locking overhead; (b) good balance; (c) too large, idle CPU; (d) entire graph, good labelings, but not parallel/local.

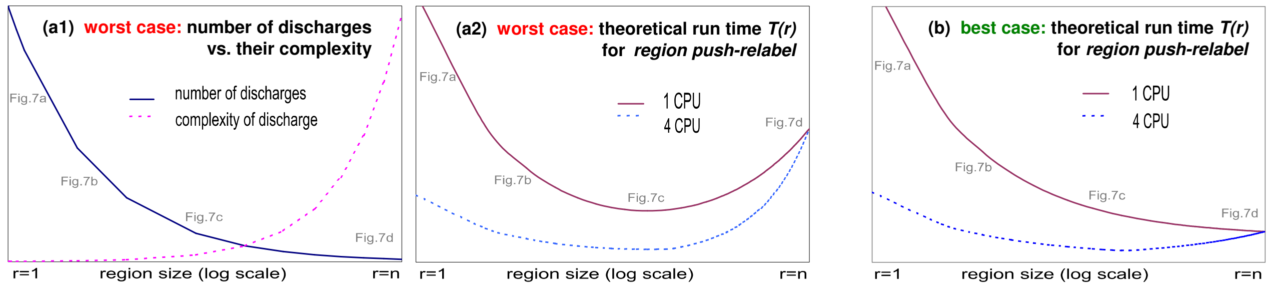


Figure 8. Region size r affects running time $T(r)$ of *region push-relabel*. In the worst case (a1) we may need $(\frac{4n^3}{r^3} + \frac{4n^2}{r^2})$ region discharge operations on a grid of size n. As we show in [9], this bound holds when selected regions are $\frac{2 \cdot n}{r}$ overlapping intervals uniformly distributed over a 1D grid. We conjecture that similar bounds hold for ND grids. Each region discharge involves at most $(a + br + cr^2 + dr^3)$ push-relabel operations internally. Then, the worst case run time (a2) is $T(r) = (\frac{4n^3}{r^3} + \frac{4n^2}{r^2})(a + br + cr^2 + dr^3)$. In the best case (b) we have $T(r) = \frac{n}{r}(a + br)$. The dotted lines in (a2) and (b) suggest ideal parallelisation opportunities $\min\{4, \frac{n}{r}\}$ for 4 CPUs.

path $vw_i \ldots t$ and so v is at least $d(w_i) + 1$ arcs away from t. The only safe move is then to increase $d(v)$ based on minimum $d(w_i)$.

These neighbouring w_i form a separator between v and the rest of the graph because any path leaving v must pass through one of the w_i. Call $\{w_i\}$ the *boundary* of $\{v\}$. Now consider the same idea for some $R \subseteq V - \{s, t\}$.

$$boundary(R) = \left\{\ w \notin R \ \middle|\ c'(v, w) > 0,\ v \in R\ \right\} \quad (1)$$

Our operations assume labels on $boundary(R)$ remain constant, and try to maximise the labels inside R under that constraint.

Figures 9 and 10 list code for our operations in terms of (1). Again, notice that both of our operations reduce to standard RELABEL when applied to a single vertex. For larger regions, their complexity and behaviour resembles that of their global counterparts. In particular, REGION-RELABEL becomes $O(n + m)$.

3.2. Regions & nearest-neighbour grid graphs

An important outstanding issue is our specific choice of regions. Figure 7 suggests that our regions on grids are

```
REGION-RELABEL($R$)   ▷ $R \subseteq V - \{s, t\}$
1  $d(R) \leftarrow \infty$
2  $Q \leftarrow R \cup boundary(R)$
3  while $Q \neq \{\}$
4      do ▷ Choose $w$ with lowest label $d(w)$
5         $w \leftarrow extract\_min(Q)$
6         for $(v, w)$ where $v \in R$
7             do if $c'(v, w) > 0$ and $d(v) = \infty$
8                   then $d(v) \leftarrow d(w) + 1$
```

Figure 9. REGION-RELABEL is simply a special case of Dijkstra initialised with multiple sources $b_i \in B = boundary(R)$ and biases $d(b_i)$. Above runs in $O(\hat{m} + \hat{n} \log \hat{n})$ where $\hat{n} = |R \cup B|$ and $\hat{m}$ is the number of arcs with at least one end in $R \cup B$. See [9] for $O(\hat{m} + \hat{n} + n_b \log n_b)$ version ($n_b = |B|$) that becomes $O(\hat{m} + \hat{n})$ with $O(n_b)$ storage for caching.

```
REGION-GAP-RELABEL($R, g$)   ▷ $g \neq d(v)\ \forall v \in R$
1  $d_b \leftarrow \min\{\ d(b) \mid g < d(b),\ b \in boundary(R)\ \}$
2  for $v \in R$
3      do $d(v) \leftarrow \max\{d(v), d_b + 1\}$
```

Figure 10. Above runs in $O(\hat{n})$ once a new gap g is detected.

roughly square, all of the same size, and have a margin between them when being discharged in parallel. Indeed this is the case for the version we analyse and evaluate in this paper. It is easy to imagine other possibilities, and this section examines tradeoffs to consider.

First, consider the effect of the boundary. When discharging a region R, our heuristics from Section 3.1 only perform useful relabeling work on $d(R)$ even though their locality and complexity also depends on $boundary(R)$. This suggests that our algorithm is only suitable when $boundary(R)$ is small enough relative to R. Take for example a first-degree nearest-neighbour grid graph of dimension N and suppose region R corresponds to an N-cube with sides of length z. The proportion of useful work done by our operations is limited by

$$\frac{|R|}{|R + boundary(R)|} \approx \left(1 - \frac{2}{z+2}\right)^N \tag{2}$$

which approaches zero as N increases, and would do so faster for higher degree neighbourhoods. To counteract this effect, larger regions could be chosen for highly connected graphs, but doing so can inhibit parallelism and locality.

Second, consider the internal 'shape' of the region. For our REGION-RELABEL heuristic to be effective in region R, the average distance of interior vertices to $boundary(R)$ should be as large as possible. On a 2D grid, for example, diamond and square shaped regions are ideal for 4- and 8-connected neighbourhoods respectively since they correspond to a circle under the distance metric by which labels d are computed.

To predetermine a set of equally sized regions, our algorithm thus depends on a good partitioning of the graph via either arc or vertex separator decomposition. The resulting groups of vertices determine which subsets $R \subseteq V - \{s, t\}$ can be selected for discharging, and how boundaries (1) are associated with these regions in practice. See [9] for details. Good quality graph partitions and separators are NP-hard to even approximate on general graphs [5], whereas on nearest-neighbour grid graphs they are trivial, bypassing any preprocessing issues. Nearest-neighbour grid graphs are ubiquitous in computer vision, medical imaging and graphics applications, but are not of particular importance to combinatorial optimisation communities. Our algorithm is still polynomial on arbitrary (i.e. poor quality) partitions of general graphs, but we have no performance ambitions in this case.

3.3. Parallel implementation

For parallel region push-relabel we maintain a global shared list of active regions. Each thread acquires a mutex on this global list, 'reserves' an active region and its boundary, and releases the mutex. Regions themselves each

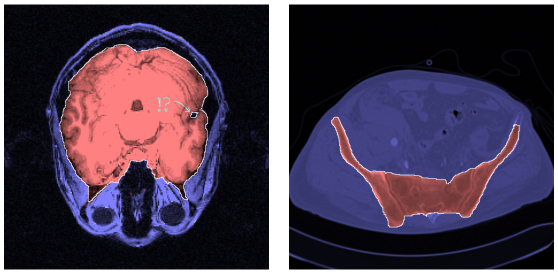


Figure 11. The $256 \times 256 \times 192$ brain MRI (left) and $512 \times 512 \times 256$ abdomen CT (right) data used to construct 6-connected graphs in our tests. Output is 3D object/background labeling (red/blue).

maintain a list of active vertices. The effect is similar to the implementation of [1] in that small 'batches' of work are passed between threads.

Since only non-intersecting regions/boundaries are processed in parallel, all vertices in the region satisfy the mutual exclusion conditions outlined by [1]. Synchronisation is only required when accessing the global queue.

We do not implement REGION-GAP-RELABEL as described in this paper. Instead we rely on REGION-RELABEL to raise labels fast, and a modified gap relabel to detect global labeling gaps. Our parallel gap relabel implementation is similar to [2] except we update label counts in small batches to factor out synchronisation overhead.

4. Experiments on 3D Segmentation

Our proof-of-concept experiments are on the binary 3D segmentation problem outlined in [4] (graph-cuts). Figure 11 shows an example 2D slice of our test output. Our performance results suggest that our algorithm is effective for immense, sparse graphs[1] typical of segmentation, multiview reconstruction and other vision applications.

Our own region push-relabel (RPR) and global heuristic push-relabel (PR) implementations are both optimised specifically for grid graphs, whereas the codes of Goldberg (HIPR-3.6) and Boykov & Kolmogorov (BK-3.0) assume arbitrary graphs. HIPR employs global relabel, gap relabel, and highest-label ordering on discharges. Vertex discharges for both RPR and PR are processed in approximate FILO order; likewise for region discharges in RPR. We also parameterise RPR tests by *region diameter*. In 3D this means our regions are cube shaped and of the same size. (i.e. diameter 64 implies 64^3 vertices, except largest diameter since our full test volumes are not perfect cubes).

Results shown are for graphs 2–3 orders of magnitude larger than what is typically tested in general papers on maximum flow [2]. For small problems in vision, such as 2D segmentation, BK has superior performance to both RPR and PR consistent with [4].

[1] A database of max-flow problem instances, for testing algorithms in vision, is available at `http://vision.csd.uwo.ca/maxflow-data/`.

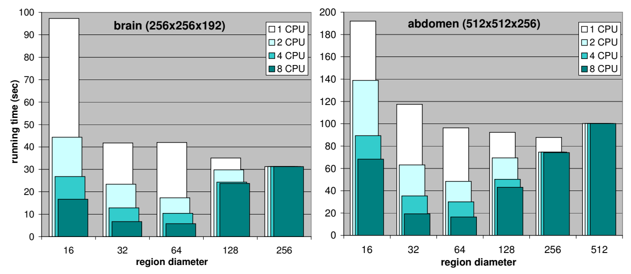


Figure 12. Running time versus region diameter for 3D segmentation. Between smallest and largest regions lies a performance "sweet spot" where parallelism opportunities are best exploited. Compare to the estimates in Figure 8.

4.1. Parallelism performance results

Our parallelism experiments in Figure 12 show that for sufficiently large problems our algorithm achieves near-linear speedup not only in a relative sense, but in an *absolute* sense. In other words, synchronisation is sufficiently coarse that our parallel algorithm beats optimised sequential push-relabel, even with few processors. This is particularly appealing for multi-core systems so prevalent today.

In Figure 12 the PR performance is represented in the rightmost column in each plot. The BK and HIPR codes ran out of virtual addresses on these graphs. On smaller 3D problems they were at least 3 times slower than our PR, but mainly because these codes are not optimised for grids.

Parallelism performance tests were run on an $8\times$ 1.8GHz Xeon E5320 workstation under Windows XP 32-bit.

4.2. Limited physical memory results

Our memory experiments show that PR, HIPR and BK are all impractical when there is not enough physical memory to contain the graph. (This is not to be confused with the virtual address space, which is typically 2–3GB for 32-bit address models and can exceed 256TB for 64-bit.)

Grid graphs can be stored without explicit adjacency information at each arc. Comparing absolute running times is not particularly informative because the HIPR and BK codes do not take advantage of this. Instead we focus on the relative impact of memory scarcity on performance.

For each implementation tested, we chose the largest 3D segmentation problem that could fit into the virtual address space of the test system. BK-3.0 managed to construct a $256\times256\times160$ grid graph needing 1.3GB. HIPR-3.6 managed $250\times250\times81$ needing 1.3GB. Our PR and RPR managed $512\times512\times256$ needing 1.2GB. (Note that PR and RPR are solving the same problem in this test, but BK and HIPR are not.) We repeatedly solved the same segmentation problem while varying the amount of total physical RAM available on the test system. RPR was configured to use one thread and a region diameter of 32 throughout. To

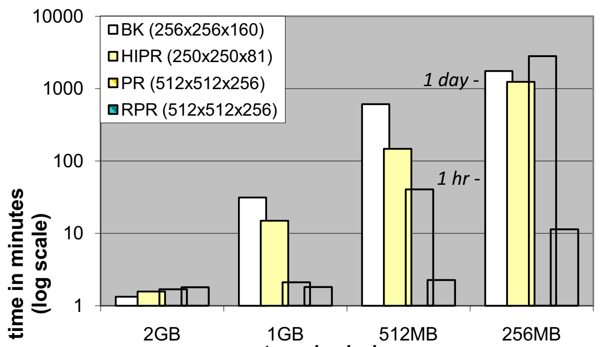


Figure 13. When a graph is larger than RAM, algorithms with poor locality will 'thrash' and become impractical. Our current RPR implementation is still affected, but at a more reasonable rate. (Note that BK and HIPR are solving smaller problems in this plot.)

improve HIPR and PR performance, in cases below 2GB the global relabel frequency h was reduced by a factor of 20 from the default. Results are shown in Figure 13.

To double 3D output resolution requires an $8\times$ increase in the size of the corresponding graph. Moving to a 64-bit architecture helps a little, but is no substitute for a scalable algorithm—physical memory perhaps goes up from 2GB to 16GB (typical for high-end 64-bit workstations), but the "performance wall" in Figure 13 is then merely two steps away instead of one.

Memory performance tests were run on a 2.8GHz Xeon workstation under Windows XP 32-bit.

5. Future work

We intend to further demonstrate our algorithm on important applications in computer vision, particularly 3D multiview reconstruction where current global methods are impractical for high-resolution output. A C++ library of RPR and BK optimised for grid-like graphs is available at `http://vision.csd.uwo.ca/code/`.

Region push-relabel affords many strategies that may affect practical performance, and we intend to explore them. Because regions are represented at a much coarser scale than vertices it pays off to invest in more sophisticated strategies at runtime when choosing active regions to discharge. For example, one could prioritise regions by:

- a schedule to minimise disk-swapping,
- density of excess (to encourage parallelism),
- min-cut estimates computed on coarse data, etc.

Perhaps most interesting is that, within an active region R, excess at any $v\in R$ can be thought of as capacity $c'(s,v)$ from the source and $boundary(R)$ as destinations (sinks) prioritised by lowest-label. An approach to maximum flow that does not scale to immense graphs, such as BK [4] or pseudoflow [6], may still be more effective on intermediate subgraphs than the region-based heuristics we have proposed. This is analogous to why quicksort implementations use insertion sort internally.

One final note is that our approach is a good starting point for computing maximum flows on non-shared/non-uniform memory architectures such as IBM's CELL. To scale to many processors, these architectures *explicitly* distinguish between high-bandwidth (but high-latency) shared memory and extremely fast local memory addresses.

Acknowledgements We wish to thank Olivier Juan for his extremely helpful investigations and discussions concerning performance of max-flow algorithms. We also thank Siemens Corporate Research, Daphne Yu in particular, for the abdomen dataset and for supporting our early research into parallel push-relabel. This work was supported by NSERC (Canada) Discovery Grant R3584A02 and in part by a NSERC PGS-D.

References

[1] R. Anderson and J. Setubal. A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem. *J. of Parallel and Dist. Comp. (JPDC)*, 29(1):17–26, 1995. 1, 2, 3, 4, 6

[2] D. Bader and V. Sachdeva. A Cache-Aware Parallel Implementation of the Push-Relabel Network Flow Algorithm and Experimental Evaluation of the Gap Relabeling Heuristic. In *ISCA Int. Conf. on Parallel and Dist. Comp. Sys. (PDCS)*, 2005. 1, 3, 4, 6

[3] Y. Boykov and G. Funka-Lea. Graph Cuts and Efficient N-D Image Segmentation. *Int. J. of Computer Vision (IJCV)*, 70(2):109–131, 2006. 2

[4] Y. Boykov and V. Kolmogorov. An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. *IEEE Trans. on Pattern Anal. and Mach. Intelligence (PAMI)*, 29(9):1124–1137, 2004. 1, 2, 4, 6, 7

[5] T. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Inf. Process. Lett.*, 42:153–159, 1992. 6

[6] B. Chandran and D. Hochbaum. A Computational Study of the Pseudoflow and Push-relabel Algorithms for the Maximum Flow Problem. *Operations Research*, (to appear). 1, 2, 4, 8

[7] B. Cherkassky. A Fast Algorithm for Computing Maximum Flow in a Network. *AMS Transactions*, 158:23–30, 1994. 3

[8] B. Cherkassky and A. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19:390–410, 1997. 1, 2, 3, 4

[9] A. Delong. A Scalable Max-Flow/Min-Cut Algorithm for Sparse Graphs. Master's thesis, University of Western Ontario, August 2006. 5, 6

[10] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989. 2, 3

[11] E. Dinitz. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970. 2, 4

[12] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM (JACM)*, 19:248–264, 1972. 2

[13] L. Ford and D. Fulkerson. Maximum Flow Through a Network. *Canadian J. of Math.*, 8:399–404, 1956. 2

[14] A. Goldberg and R. Tarjan. A New Approach to the Maximum Flow Problem. *Journal of ACM (JACM)*, 35(4):921–940, 1988. 1, 2, 3, 4, 5

[15] V. Kolmogorov and C. Rother. Minimizing nonsubmodular functions with graph cuts - a review. *IEEE Trans. on Pattern Anal. and Mach. Intelligence (PAMI)*, 29(7), 2007. 1

[16] V. Lempitsky and Y. Boykov. Global Optimization for Shape Fitting. In *Comp. Vision and Pattern Recognition (CVPR)*, 2007. 1

[17] H. Lombaert, Y. Sun, L. Grady, and C. Xu. A Multilevel Banded Graph Cuts Method for Fast Image Segmentation. In *10th IEEE Int. Conf. on Comp. Vision (ICCV)*, pages 259–265, 2005. 1

[18] C. Rother, V. Kolmogorov, V. Lempitsky, and M. Szummer. Optimizing Binary MRFs via Extended Roof Duality. In *Comp. Vision and Pattern Recognition (CVPR)*, 2007. 1

[19] P. M. S. N. Sinha and M. Pollefeys. Multi-View Stereo via Graph Cuts on the Dual of an Adaptive Tetrahedral Mesh. In *Int. Conf. on Comp. Vision (ICCV)*, 2007. 1

[20] D. Sleator and R. Tarjan. A data structure for dynamic trees. *J. of Comp. and Sys. Sci.*, 24:362–381, 1983. 3

[21] G. Vogiatzis, P. Torr, and R. Cipolla. Multi-view stereo via Volumetric Graph-cuts. In *Comp. Vision and Pattern Recognition (CVPR)*, pages 391–398, 2005. 1