

# Parallel Graph-cuts by Adaptive Bottom-up Merging

Jiangyu Liu

Jian Sun

University of Science and  
Technology of China

Microsoft Research Asia

## Abstract

*Graph-cuts optimization is prevalent in vision and graphics problems. It is thus of great practical importance to parallelize the graph-cuts optimization using today's ubiquitous multi-core machines. However, the current best serial algorithm by Boykov and Kolmogorov [4] (called the BK algorithm) still has the superior empirical performance. It is non-trivial to parallelize as expensive synchronization overhead easily offsets the advantage of parallelism.*

*In this paper, we propose a novel adaptive bottom-up approach to parallelize the BK algorithm. We first uniformly partition the graph into a number of regularly-shaped disjoint subgraphs and process them in parallel, then we incrementally merge the subgraphs in an adaptive way to obtain the global optimum. The new algorithm has three benefits: 1) it is more cache-friendly within smaller subgraphs; 2) it keeps balanced workloads among computing cores; 3) it causes little overhead and is adaptable to the number of available cores. Extensive experiments in common applications such as 2D/3D image segmentations and 3D surface fitting demonstrate the effectiveness of our approach.*

## 1. Introduction

The solution to many important vision problems, such as image segmentation [3], stereo matching [21] and multi-view reconstruction [23], involves a minimization of the following energy function (derived from a Maximum A Posteriori estimation of a Markov Random Field [16]):

$$E(X) = \sum_p E_d(x_p) + \sum_{p,q} E_c(x_p, x_q) \quad (1)$$

where  $X = \{x_p\}$  represents the labeling of image pixels,  $E_d(x_p)$  is a data penalty function, and  $E_c(x_p, x_q)$  is the interaction potential.

Minimizing the above energy function can be transformed into solving a *graph-cuts optimization* on a digraph  $\mathcal{G}(\mathcal{V}, \mathcal{E})$  where the nodes  $\mathcal{V}$  are the image pixels plus two

specially added terminals, namely the *source* and the *sink*, and the capacities of edges  $\mathcal{E}$  are set corresponding to the energy terms in Equation (1). The goal is to find a *minimum cost cut* that separates the nodes into two disjoint sets  $\mathcal{S}$  and  $\mathcal{T}$ , so that the source is in  $\mathcal{S}$  and the sink is in  $\mathcal{T}$ . The optimization is equivalent to solving the *maxflow* problem, which is to find the maximum amount of flow that can be sent from source to sink via the capacitated edges  $\mathcal{E}$ .

Generally, maxflow algorithms are categorized as augmenting-path based [4, 11, 12, 13, 28] and push-relabel based [5, 6, 9, 15, 17], and most practical parallel maxflow algorithms are of the push-relabel variety [1, 2, 8] due to the memory locality feature of push-relabel operations. However, on a commodity multi-core platform, the current best serial algorithm by Boykov and Kolmogorov [4] (called the BK algorithm), which iteratively searches for augmenting paths to send flow from source to sink, is still faster for common vision and graphics problems. Therefore, effectively parallelizing the BK algorithm would further boost the performance of many graph-cuts based vision applications<sup>1</sup>.

In this paper, we propose a parallel version of the BK algorithm. We observe that the searched augmenting paths often vary largely in range, with some paths containing only several nodes while some spreading on the entire graph. While the serial BK algorithm searches for all the paths in an out-of-order fashion, we explicitly divide the computation into two phases to facilitate parallelization. In the first phase, we uniformly partition the graph into a number of disjoint subgraphs, so that we can concurrently search for short-range paths within each subgraph, without synchronization; moreover, the partitioning promotes memory locality and hence improves cache-friendliness for computations within the subgraphs. In the second phase, we gradually merge the subgraphs in an adaptive way to enlarge the search range and consequently augment flow on longer paths, still in parallel, until the maximum flow is reached.

---

<sup>1</sup>GPU-based CUDA Cuts [29] might be superior when the data transferring between main memory and graphics card is not the bottleneck. In this paper, we focus on the more general CPU-based approaches.

It is important to note that, during the adaptive merging, we can keep reusing intermediate results already obtained to search for new paths, without ever having to start from scratch. Extensive experiments on 2D image segmentation, 3D volume segmentation, and 3D surface fitting show the good speedup factors of our proposed algorithm.

## 2. Related Work

To find the maximum flow, the push-relabel algorithm [15] takes a local approach by working on one node at a time and looking only at the node’s neighbors. The algorithm maintains an *excess* and a *height* for each node to guide the push and relabel operations. The algorithm starts by sending an excessive amount of flow into the graph from the source, and then it keeps adjusting the heights of nodes and pushing flow between nodes until all possible flow reaches the sink and the rest of flow that was sent into the graph goes back to the source.

The augmenting path algorithm [13] does not preserve memory locality in that it directly finds an *augmenting path* from source to sink for sending flow, which could involve any number of nodes. The algorithm iterates the path finding until all augmenting paths are exhausted. To speed up the process, one should consider augmenting flow on paths as short as possible (since striving for shortest augmenting paths only leads to a polynomial time complexity), while minimizing the costs of finding such short paths [4, 11, 12, 28]. For example, the BK algorithm [4], based on which our parallel algorithm is developed, well achieves such balance in practice.

### 2.1. The BK algorithm

The BK algorithm [4] maintains two non-overlapping dynamic trees with non-saturated edges (capable of transporting flow), originating from source and sink respectively; it constantly sends flow on augmenting paths that are formed when the two trees touch each other. The algorithm iterates through three stages, namely, *growth*, *augmentation* and *adoption*, until the maximum flow is achieved.

At growth stage, the two dynamic trees expand in a breadth-first manner by attaching *free* nodes onto the trees, using a set of *active* nodes. The growth stage terminates as soon as an active node touches a node on the opposite tree, forming an augmenting path. Then at augmentation stage, the edge capacities along the path are decremented as flow is sent from source to sink; as a result, some nodes become *orphans* as their connecting edges to their *parents* get saturated. Finally, at adoption stage, orphan nodes either are assigned new parents or become free nodes again. The iteration of the three stages terminates when no more augmenting paths can be found.

### 2.2. Parallel BK algorithms

Although not optimal in theory, the BK algorithm demonstrates the best empirical performance among most maxflow algorithms for typical vision and graphics problems [4], it even beats the top parallel push-relabel algorithm for applications like 2D segmentation [8]. However, we cannot trivially parallelize this algorithm as locking individual nodes would cause an overhead that is even larger than the gain of parallelism.

Juan and Boykov first pointed out the possibility of parallelizing the *pushing* and *pulling* operations for *Active Graph Cuts* [19], which is an extension to the BK algorithm. However, such parallelization is confined to dual processors and there is little guarantee on balanced workloads. Recently, a parallel BK algorithm was proposed to coarsen the locking with a graph partitioning method [25]. In this algorithm, the graph was partitioned into several disjoint subgraphs, so that each core can process a single subgraph without synchronization on individual nodes. To keep balanced workloads, the algorithm alternatively partitions the graph in different directions, based on the location of active nodes. However, the drawback is that alternative graph partitioning is likely to destroy dynamic trees already constructed and reusing the trees requires extra operations of orphan adoption [4].

## 3. A New Parallel BK Algorithm

To avoid synchronization on individual nodes, our proposed algorithm adopts the graph partitioning strategy similar to [25]. However, different from the iterative dynamic partitioning approach, our algorithm distributes the computational workloads into a *uniform partitioning* phase and an *adaptive merging* phase, to facilitate parallelism by discriminately searching for augmenting paths of different ranges. We next describe the two phases in detail, taking the example of computing maxflow on a 2D grid graph.

### 3.1. Phase One: uniform partitioning

To concurrently search for short-range paths in the graph without locking, in the first phase we start by uniformly partitioning the graph into a number of disjoint rectangular subgraphs denoted as *blocks*, with a list of *boundary segments*, as illustrated in Figure 1. Partitioning means to set all the edge capacities between two adjacent blocks to zero (these edge capacities are stored in an auxiliary buffer for the adaptive merging phase). Each block is assigned a unique ID, and a boundary segment records the locations of boundary nodes of two adjacent blocks, as well as the IDs of the two blocks.

Next we spawn a number of threads (equal to the number of available cores) to process the partitioned blocks in parallel. Each thread works as follows:

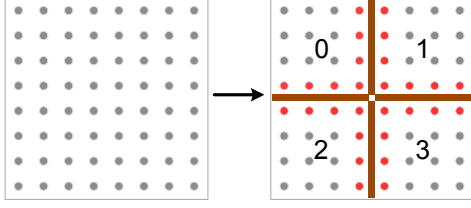


Figure 1. A partition example. Here an  $8 \times 8$  grid graph is partitioned into four  $4 \times 4$  disjoint blocks, with four boundary segments. Each boundary segment records the locations of boundary nodes (red) on its both sides (up to 8 nodes in this case), as well as the IDs of the two adjacent blocks.

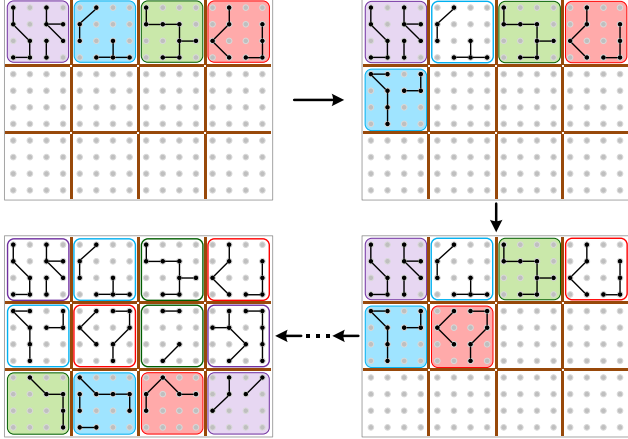


Figure 2. Parallel processing based on uniform partitioning (with four cores). The grid graph is first partitioned into disjoint blocks with boundary segments; each thread runs the BK algorithm within one block at a time, and moves on to the next unprocessed one upon finishing, until all the blocks are processed.

```

1: while true do
2:   block_id = InterlockedIncrement(&g_block_id);
3:   if block_id exceeds total number of blocks then
4:     return;
5:   end if
6:   Run the BK algorithm within the block indexed by
     block_id;
7: end while

```

“*g\_block\_id*” is a global variable initialized as 0 before the threads are spawned; the “InterlockedIncrement()” function increments the integer passed in by one as an atomic operation and returns the incremented value (here we assume that the blocks are indexed starting from 1). The scenario is that each thread processes one block at a time, and upon finishing, it moves on to handle the next unprocessed block, until all the blocks are processed, as shown in Figure 2.

Within each block, dynamic trees are constructed and naturally restrained at block boundaries from forming augmenting paths across adjacent blocks because of the reset

edge capacities, thus we cannot obtain the global optimum in this phase. But the benefit is that we do not need any locking as all threads are independent of each other, and the workloads are adaptively distributed as each thread is able to process a flexible number of blocks. Besides, it is more cache-friendly to run the BK algorithm on smaller subgraphs than on the entire graph, and speed may be further boosted as shorter paths are forced to exhaustion in the smaller blocks. (The BK algorithm does not guarantee shortest augmenting paths.)

### 3.2. Phase Two: adaptive merging

Because obtaining the global optimum was obstructed by the reset edge capacities, in the second phase, we adaptively merge the blocks to search for the remaining longer augmenting paths in parallel, until all edge capacities at block boundaries are restored and the global optimum is achieved. In this context, a *block* is extended to represent not only the rectangular subgraph defined in the first phase, but any subgraph merged from smaller subgraphs. We maintain the blocks as a disjoint-set forest with simple union-find operations [7]. *Merging* then refers to uniting two isolated blocks into one and exhausting all augmenting paths within the new block.

We perform the adaptive merging by incrementally removing the boundary segments to form larger blocks; removing a boundary segment means to restore the edge capacities between nodes on its both sides. Once a boundary segment is removed, we can reuse dynamic trees already constructed to form new augmenting paths for sending flow. Rather than locking individual graph nodes, we only need to impose a coarse lock on the block in which the BK algorithm is running. Since a block is still disjoint from its adjacent blocks, the dynamic trees will not extend beyond the block boundary to cause hazardous race conditions.

Specifically, our algorithm is to let each thread keep scanning the list of boundary segments; once a thread has found one boundary segment connecting two *unlocked* blocks, it applies a union operation [7] to unite the two blocks into one and marks it as *locked* (a block structure has a 1-bit field for the locking status), next it finds the rest of boundary segments also residing in the new block and removes all found boundary segments. To start the BK algorithm in the block, the thread “activates” [4] nodes on both sides of the removed boundary segments, so that the flow can be sent across previously isolated graph regions. After finishing the BK algorithm, the thread marks the block as *unlocked* and returns to scanning the list. A thread exits when it can no longer find a boundary segment to merge blocks; eventually the last-exiting thread obtains the global optimum within the only block left, which is the entire graph, as illustrated in Figure 4. The process is referred to as *adaptive* because the order for merging the blocks is

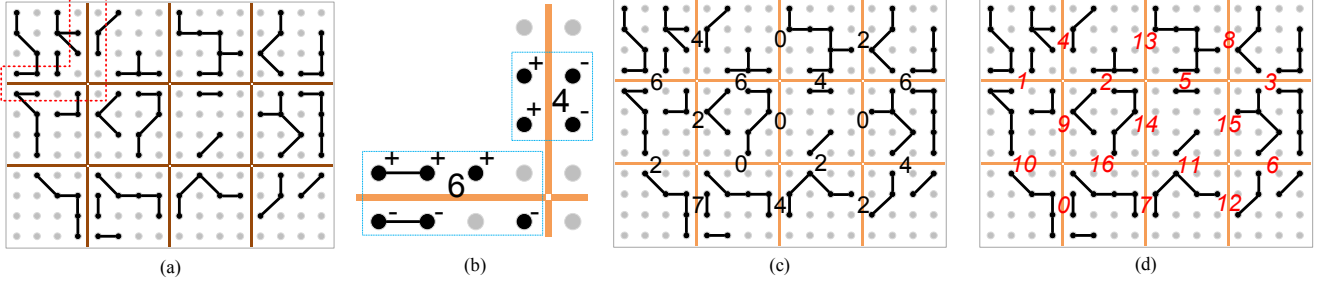


Figure 3. Sorting the boundary segments. (a) Maxflow result produced in the first phase. (b) Close-up view on counting potential active nodes (8-connected) for the first two segments. (c) The counting result for all segments. (d) The order of segments in the list after sorting.

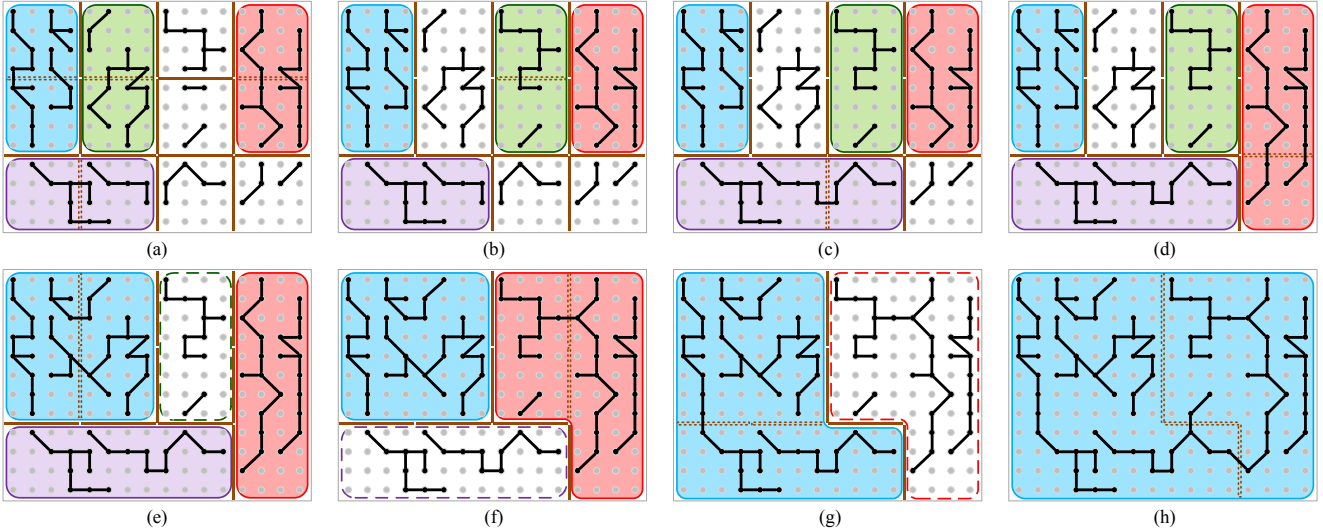


Figure 4. Adaptive merging (with four cores) based on the sorting order in Figure 3. In (a), each thread scans the sorted list to find a boundary segment connecting two unlocked blocks, then removes the boundary segment (denoted with dashed lines) and continues running the BK algorithm to merge the blocks. In (b), when a thread finishes merging, it keeps scanning the list to locate the next eligible boundary segment(s), and continues to perform the merging. From (c) to (h), such procedure goes on until all boundary segments are removed and the global optimum is achieved. Starting from (e), a thread exits (denoted with dashed bounding box) when it cannot find a proper boundary segment for further merging. So do another two threads in (f) and (g), respectively.

solely determined by the workload dynamics at run-time.

The scanning order of the boundary segment list is important. As the merging progresses, parallelism is likely to decrease because there will be fewer blocks available than computing cores. To avoid imbalanced workloads, we should prioritize the removal of boundary segments through which dense augmenting operations would occur. Notice that when activating nodes beside boundary segments, we need not activate a node unless its neighbor node(s) on the other side of the boundary segment has a different label, indicating an augmenting path will be formed; the workload associated with one boundary segment can then be roughly measured by the number of *potential* active nodes (nodes to be activated later) beside it. Thus prior to the merging, we count the number of potential active nodes for each boundary segment and sort them in a non-ascending order, as shown in Figure 3. This strategy leads to an average of

15% speed increase compared to a random ordering.

After the boundary segment list is sorted, each thread carries out the adaptive merging as follows:

```

1: while true do
2:   Lock synchronization object;
3:    $S = \text{Get\_Boundary\_Segment\_Set}()$ ;
4:   if  $S = \emptyset$  then
5:     Unlock synchronization object;
6:     return;
7:   else
8:     Mark the block in which  $S$  exists as locked;
9:   end if
10:  Unlock synchronization object;
11:  for each boundary segment  $B$  in  $S$  do
12:    Restore edge capacities between nodes on both sides of  $B$ ;
13:    Activate nodes on both sides of  $B$  if their neigh-

```

bors on the other side have a different label;

```

14: end for
15: Run the BK algorithm with newly activated nodes;
16: Lock synchronization object;
17: Mark the block in which  $S$  exists as unlocked;
18: Unlock synchronization object;
19: end while

```

We maintain only one synchronization object for all threads to prohibit simultaneous retrievals of boundary segments as well as modifications to the locking status of the blocks. The pseudocode for the “Get\_Boundary\_Segment\_Set()” function, which unites two unlocked blocks and retrieves the corresponding unremoved boundary segments, is shown as follows:

```

1: Get_Boundary_Segment_Set()
2: Initialize boundary segment set  $S$  as  $\emptyset$ ;
3: for each element  $B$  in the boundary segment list  $L$  do
4:   Get the two blocks on both sides of  $B$ ;
5:   if neither of the two blocks is locked then
6:     Unite the two blocks into one;
7:     Add  $B$  into  $S$ ;
8:     Scan the rest of  $L$  to add other boundary segments residing in the united block into  $S$ ;
9:     Erase from  $L$  every boundary segment in  $S$ ;
10:  end if
11: end for
12: return  $S$ ;

```

Because the boundary segment list is a linear structure containing a much smaller number of elements compared to the number of graph nodes, the overhead of the locked operations is almost negligible compared to maxflow computations. Furthermore, it takes less and less time to retrieve the boundary segment set as the list keeps shrinking.

The correctness of the parallel algorithm can be established by observing that for each block throughout the two phases, we have essentially exhausted every augmenting path within it, using the serial BK algorithm. This observation is self-evident for the first phase. For the second phase, however, it is not so obvious as we only activate the “potential active nodes” for merging two blocks. Actually, what we do is equivalent to activating every node in the two blocks, and using nodes other than the potential active nodes to search for paths first, which, however, would result in no augmentation at all. Because the output of the BK algorithm does not depend on the search order of active nodes, we guarantee optimality in any merged block till the end of the merging process.

### 3.3. Implementation details

**“Marking” heuristic.** In [20], Kolmogorov introduced a “marking” heuristic that intuitively shortens the augmenting paths to speed up the serial BK algorithm. In this heuristic,

a global counter  $TIME$  is maintained that is incremented at the beginning of each augmentation; each graph node is attached with a *time-stamp* label and a *distance* label. The *distance* label is an estimate on the distance between the node and the terminal (source or sink); the *time-stamp* label records when such an estimate was made based on the  $TIME$  counter. These labels are used to help a tree node to get a “better” parent that is closer to the terminal than its existing parent (at growth stage), and an orphan node to get a new parent that is closest to the terminal among all possible parents (at adoption stage).

To incorporate the “marking” heuristic into our parallel implementation, we maintain a  $TIME$  counter for each individual block throughout the two phases. An additional but simple operation is required in the second phase: after uniting two blocks into one (and before starting the BK algorithm), we initialize the  $TIME$  counter of the new block as  $TIME_{new} = \max(TIME_0, TIME_1)$ , where  $TIME_0$  and  $TIME_1$  are counters for the two blocks to be merged. This initialization invalidates all *time-stamp* labels in the new block and enables us to continue running the heuristic-aided BK algorithm correctly. Overall, the “marking” heuristic provides the same speedup factor to our parallel algorithm as it does to the serial BK algorithm.

**Partitioned block size.** The size of the partitioned block in the first phase also impacts the performance. If the size is too small, the workload of the first phase decreases (consider a  $1 \times 1$  block, in which no flow can be sent) while the number of boundary segments increases, leading to larger overhead in the second phase; if the size is too large, load imbalance is likely to occur among the small number of blocks (consider the block as the entire graph, in which no parallelism can be exploited).

In our implementation, assuming the width and height of the grid graph are  $w$  and  $h$  respectively, the width and height of a partitioned block are experimentally set as:  $w_b = \min(\frac{w}{4}, \sqrt{\frac{L2 \text{ cache size}}{2 \times \text{node size}}})$  and  $h_b = \min(\frac{h}{4}, \sqrt{\frac{L2 \text{ cache size}}{2 \times \text{node size}}})$ . For smaller graphs, there are at least 16 blocks so that the possibility of load imbalance is relatively small; for larger graphs, the nodes in the partitioned block can approximately fit in half of the L2 cache (on current commodity *Intel* multi-core platform, one L2 cache ranges from 1MB to 12 MB in size and is shared by two cores [18]), which obtains a good trade-off between block size and load balance.

## 4. Applications and Results

To verify the effectiveness of the proposed algorithm, we test it in common vision applications such as 2D/3D image segmentation and 3D surface fitting (shown in Figure 5),

and demonstrate the results. Our experiments were conducted on a  $4 \times 2.8\text{GHz}$  Xeon E5440 CPU with 4GB RAM; the code is optimized for grid graphs<sup>2</sup> and compiled under the 32-bit Windows system.

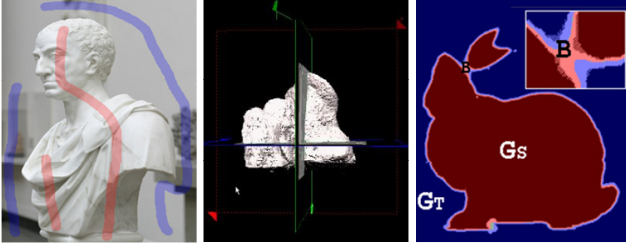


Figure 5. Graph-cuts applications. Left to right: 2D image segmentation, 3D image segmentation, 3D surface fitting. The right two images are courtesy of [26].

#### 4.1. 2D image segmentation

Figure 6 shows the performance of our algorithm for 2D image segmentation, on a 3.1 Mega-pixel and a 6.3 Mega-pixel image, respectively. The segmentations were made by marking strokes on both foreground and background [3], and the graphs were constructed as described in [27]. As can be seen, our algorithm provides near-linear speedup with respect to the number of cores.

The overhead of our algorithm is small. In the instance of segmenting the 6.3 Mega-pixel image, the parallel algorithm using one core (spawning one thread) is even a bit faster than the serial BK algorithm. The reasons are twofold: first, the costs of extra operations such as graph partitioning and boundary segments retrieving are indeed trivial compared to the BK computations; second, such costs may be negated by the gain of cache-friendliness plus the possibility of augmenting even shorter paths, thanks to the uniform partitioning.

**Comparison with Alternative Graph Partitioning.** Figure 7 compares our algorithm with the Alternative Graph Partitioning algorithm (AGP) [25] on varying numbers of cores, using the segmentation instances above (both algorithms implement the “marking” heuristic as described in Section 3.3). Our algorithm is clearly more efficient in all cases. The main reason is that the AGP has to spend a significant amount of time restoring the tree structures destroyed by each partitioning, in order to reuse the dynamic trees; while our algorithm is free from such overhead as we partition the graph only at the beginning, before any dy-

<sup>2</sup>The main difference between our code and the popular *BK-3.0* code [14] is that, instead of explicitly maintaining an “arc” structure, we store a fixed-sized array in the graph node itself as the weights of outgoing edges for regular grids, which facilitates memory access and speeds up the runtime execution.

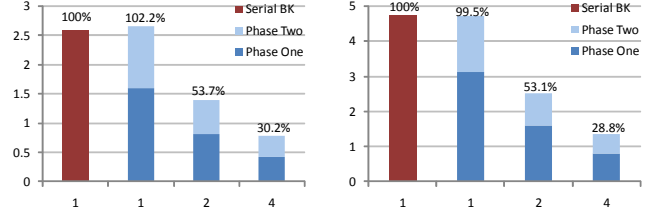


Figure 6. Performance in 2D image segmentation. Left: on a 3.1 Mega-pixel image ( $1448 \times 2172$ , 8-connected, maximum capacity 1600). Right: on a 6.3 Mega-pixel image ( $2048 \times 3072$ , 8-connected, maximum capacity 1600). X-axis denotes the number of cores and Y-axis denotes the computing time (in seconds).

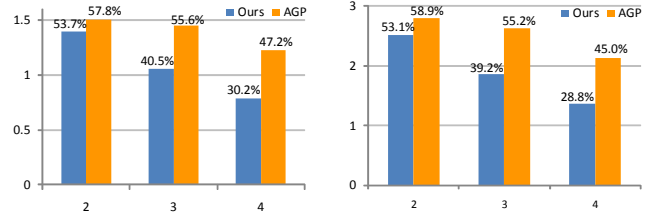


Figure 7. Performance comparisons between our algorithm and the Alternative Graph Partitioning algorithm (AGP) [25], against varying numbers of cores (using the same instances in Figure 6). (Serial BK = 100%)

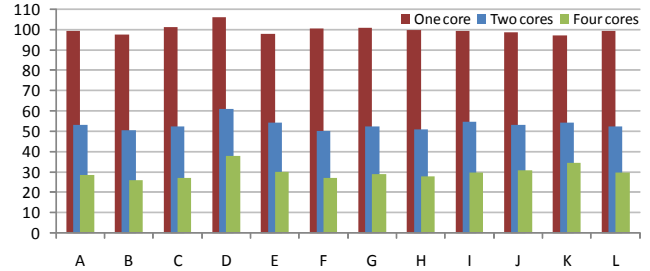


Figure 8. Average performance for segmenting twelve randomly chosen images. X-axis denotes different images and Y-axis denotes the percentage compared against the serial BK algorithm (100%).

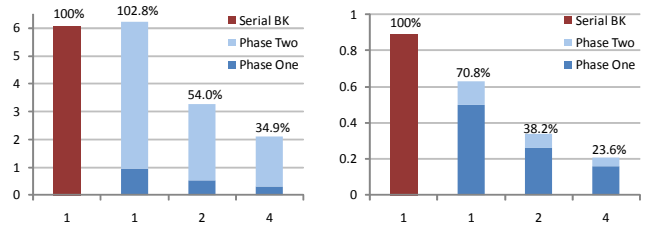


Figure 9. Performance in 3D applications. Left: image segmentation ( $170 \times 170 \times 144$ , 6-connected, maximum capacity 100). Right: surface fitting ( $202 \times 199 \times 157$ , 6-connected, maximum capacity 30000)

dynamic tree is constructed, and later in both phases, the dynamic trees only keep growing without getting disturbed.

Another advantage of our algorithm is that it is adaptable to the number of cores, while the AGP demands it to be a power of two [25]. For instance, when only 3 cores are idle (a situation common in multi-tasking environments), with

the AGP we still have to dispatch the quad-core version to fully utilize computing resources. However, more threads running on less cores would inevitably cause excessive thread switching, and the performance is only slightly better than the dual-core version, as shown in Figure 7. In contrast, our algorithm can spawn the same number of threads as the available cores and keep providing scalable performances.

**Robustness across multiple images.** Figure 8 shows our algorithm’s average performance in segmenting twelve randomly selected images (with identifiable foreground) of varying resolutions (2.6 – 10.0 Mega-pixels) from the web [10]. For each image, we first mark several strokes (4 – 10) until the foreground is correctly segmented (using the serial BK algorithm), in the process, we record every stroke that triggers the maxflow computation. Then, altering the numbers of cores, we replay the strokes to measure the average computing time of the parallel algorithm and compare it with the serial one. On average, with two cores, our algorithm takes 53.3% of the time consumed by the serial BK algorithm; with four cores, the ratio is 29.8%. In some cases (such as Image D in Figure 8) where the average length of augmenting paths is relatively long with respect to the image size, the computing time does not scale so well with the number of cores, the analysis on the relationship between path lengths and parallelism will be presented in the next section.

#### 4.2. 3D image segmentation and surface fitting

The algorithm for 2D graphs can be naturally extended to solve 3D graph-cuts optimizations. The only modification is that, in the 3D scenario, the graph is partitioned into *cubes* instead of planar blocks and the boundary segments described in Section 3 become *boundary surfaces*.

Figure 9 shows the algorithm’s performance for 3D image segmentation and surface fitting. The testing instances are taken from the benchmarking website for maxflow problems in vision [26]. The segmentation instance is the liver volume, and the surface fitting instance is based on the laser scans of the Stanford Bunny [22].

From Figure 9, we can see that the time proportions of uniform partitioning and adaptive merging phases vary largely with particular graph instances. For example, in surface fitting, the augmenting paths from source to sink are mostly short (as the graph only involves a narrow band in the 3D space [22]), so the first phase takes the majority of computing time. On the other hand, in 3D segmentation, the paths are relatively long (possibly spanning the entire 3D volume), such that most augmenting operations cannot take place until blocking boundary surfaces are removed, therefore the computing time is concentrated on the second phase.

Such variation has a direct impact on the extent of parallelism. In the surface fitting case where the uniform partitioning is dominant, cache-friendliness and reduced workloads from augmenting shorter paths even enable the parallel algorithm to achieve super-linear speedup compared to the serial BK algorithm. While in the 3D segmentation case where adaptive merging is dominant, parallelism drops to sub-linear, because the merits of cache can hardly be enjoyed with larger ranges of nodes, and a thread has to frequently fetch data from the main memory in both growth and adoption stages. With the current commodity shared-memory multi-core architecture, fetching data from the main memory is *serial* as all cores share the memory bus, which is a common bottleneck that prevents most parallel algorithms from achieving scalable performances [24]. This is also the main reason why our algorithm’s speed increase is not strictly linear with respect to the number of cores.

## 5. Conclusion

In this paper, we have presented a new parallel BK algorithm for graph-cuts optimizations that outperforms the state of the art. Through novel uniform partitioning and adaptive merging operations, our proposed algorithm achieves near-linear speedup for common vision problems. Our algorithm is cache-friendly, keeps balanced workloads, causes little overhead and is adaptable to the number of available cores. These features make our algorithm a practical and attractive replacement for the serial BK algorithm on commodity multi-core platforms that are prevailing today.

We need to point out that currently our algorithm is only suitable for graphs that can fit in the system memory. For massive graphs that exceed memory limits, the parallel algorithm implementing “region push-relabel” [8] still provides the best solution. In the future, we would like to explore the possibility of adapting our parallel BK algorithm to memory-insufficient conditions; another important aspect of the future work is to design practical partitioning strategies so as to apply our algorithm to general graphs.

**Acknowledgements.** We thank the anonymous reviewers and area chairs for the insightful and constructive comments to help improve the paper. This work was done when the first author was an intern at Microsoft Research Asia.

## 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.

- [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.
- [3] Y. Boykov and M. P. Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in n-d images. In *Proceedings of ICCV*, 2001.
- [4] 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 (PAMI)*, 26(9):1124–1137, 2004.
- [5] B. Chandran and D. Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations Research*, 57(2):358–376, 2009.
- [6] B. Cherkassky and A. Goldberg. On implementing push-relabel method for the maximum flow problem. In *Proceedings of the 4th International IPCO Conference on Integer Programming and Combinatorial Optimization*, 1995.
- [7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.
- [8] A. DeLong and Y. Boykov. A scalable graph-cut algorithm for n-d grids. In *Proceedings of CVPR*, 2008.
- [9] 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.
- [10] Digital Camera Reviews and News. <http://www.dpreview.com/gallery/>.
- [11] E. Dinits. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.
- [12] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM (JACM)*, 19:248–264, 1972.
- [13] L. Ford and D. Fulkerson. Maximum flow through a network. *Canadian J. of Math.*, 8:399–404, 1956.
- [14] Generic max-flow/min-cut library. <http://vision.csd.uwo.ca/code/>.
- [15] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. 35(4):921–940, 1988.
- [16] D. Greig, B. Porteous, and A. Seheult. Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society*, 51(2):271–279, 1989.
- [17] D. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum flow problem. *Operations Research*, 56(4):992–1009, 2008.
- [18] Intel Corporation. <http://www.intel.com/multi-core/>.
- [19] O. Juan and Y. Boykov. Active graph cuts. In *Proceedings of CVPR*, 2006.
- [20] V. Kolmogorov. *Graph Based Algorithms for Scene Reconstruction from Two or More Views*. PhD thesis, Cornell University, 2003.
- [21] V. Kolmogorov and R. Zabih. Computing visual correspondence with occlusions using graph cuts. In *Proceedings of ICCV*, 2001.
- [22] V. Lempitsky and Y. Boykov. Global optimization for shape fitting. In *Proceedings of CVPR*, 2007.
- [23] V. Lempitsky, Y. Boykov, and D. Ivanov. Oriented visibility for multiview reconstruction. In *Proceedings of ECCV*, 2006.
- [24] C. Lin and L. Snyder. *Principles of Parallel Programming*. Addison-Wesley, 2008.
- [25] J. Liu, J. Sun, and H.-Y. Shum. Paint selection. *ACM Trans. Graph.*, 28(3):1–7, 2009.
- [26] Max-flow problem instances in vision. <http://vision.csd.uwo.ca/maxflow-data/>.
- [27] C. Rother, A. Blake, and V. Kolmogorov. Grabcut - interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph.*, 24(3):309–314, 2004.
- [28] D. Sleator and R. Tarjan. A data structure for dynamic trees. *J. of Comp. and Sys. Sci.*, 24:362–381, 1983.
- [29] V. Vineet and P. J. Narayanan. Cuda cuts: Fast graph cuts on the gpu. In *Proceedings of CVPR Workshops*, 2008.