Proceedings of Parallel Scientific Computing
May 26, 2021, IIT, Madras

FASTER IMAGE SEGMENTATION ON THE MULTI-CORE PLATFORMS USING THE GRAPH CUTS

Jash Khatri
PACE Laboratory
Department of Compute Science and Engineering
Indian Institute of Technology, Madras, 600036
Email: cs19s018@smail.iitm.ac.in

ABSTRACT

Graph cuts find their applications in the various number of computer vision and image processing problems. Max-flow/Min-cut algorithms are the well-known algorithms to compute graph cuts. However, determining the graph cuts using the sequential Max-flow/Min-cut algorithms is computationally expensive. As part of this work, we present the parallel implementation of the Max-flow/Min-cut algorithm for computing the graph cuts. The presented parallel implementation of the Max-flow/Min-cut algorithm uses multi-core CPUs to compute the graph cuts faster than its sequential implementation. We then apply our presented parallel implementation of the Max-flow/Min-cut algorithm to the problem of image segmentation. We use our stated parallel algorithm to divide an image into background and foreground segments through the following steps. In the first step, we generate the input flow network from the given input square grayscale image. In the second step, we use the parallel Max-flow/Min-cut algorithm to compute the graph cut faster in the input flow network, which gives us the precise segments of the input image. We then compare the segmented images obtained using sequential and parallel image segmentation applications to analyze the presented parallel algorithm's correctness. We then finally demonstrate the parallel image segmentation application's speed-up over the sequential image segmentation application on the several input square grayscale images.

NOMENCLATURE

G Flow network or Graph.

I_p Denotes intensity of the pixel vertex p in the input square grayscale image.

V Set of vertices in the flow network.

E Set of edges in the flow network.

c Function($V \times V \rightarrow R$) denoting edge capacities one corresponding to each edge in the flow network.

f Denotes the flow function($V \times V \rightarrow R$) which is defined on each edge of the flow network.

h Function($V \rightarrow I$) denoting height associated with the vertex present in the flow network.

e Function($V \rightarrow R$) denoting excess flow associated with the vertex present in the flow network.

s Source vertex in the flow network.

t Sink vertex in the flow network.

c_f Function($V \times V \rightarrow R$) denoting residual capacity of each edge of the flow network.

G_f The residual network of a given input flow network induced by the residual edges.

INTRODUCTION

In Computer Vision, Graph cut optimization solves various problems like image smoothing, the stereo correspondence problem, image segmentation [1, 2, 3, 4, 5, 6]. Computing the graph cut can be modeled as an energy minimization problem which can be solved using the maximum flow algorithms. This work focuses on segmenting the square grayscale image into the foreground and background components using the maximum flow algorithm.

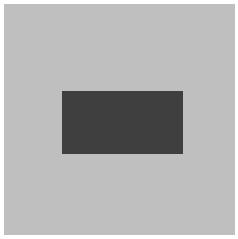


FIGURE 1: Square grayscale Image

Our implementation consists of two phases. First, we build the network flow graph from the input square grayscale image. Second, the max-flow algorithm is run on the graph to find the min-cut, which produces the optimal segments of the input image. The state-of-the-art Push-Relabel algorithm [7] is used for computing the min-cut on the generated network flow graph. Push-Relabel algorithm computes the maximum flow values for the given flow network in time $O(V^2E)$. Due to its high time-complexity, computation of maximum flow is quite time-consuming even on moderately-sized graphs. Multi-core platforms can be a promising avenue towards improving its execution time.

As part of this work, we present the parallel implementation of the push-relabel algorithm for computing the graph cuts using OpenMP. The presented parallel implementation of the push-relabel algorithm is up to 2.17× faster than the serial push-relabel algorithm and performs consistently better than the serial push-relabel algorithm for various input square grayscale images.

PROBLEM DESCRIPTION

Given the square grayscale image and six seed pixels, five indicating the background and one indicating the foreground, our task is to divide the given square grayscale image into two segments. One segment includes all the foreground pixels, and another segment includes all the background pixels. The sample square grayscale image is shown in the figure 1. The pixels forming the rectangle in the figure 1 forms the foreground, while the rest of the pixels forms the background.

FLOW NETWORK AND MAXIMUM FLOW

A flow network is a directed graph G(V, E, c) where V and E denote the sets of vertices and edges in the graph, respectively, and each edge (u, v) ∈ E has a capacity c(u,v). In addition to this, the graph has source vertex s ∈ V and sink vertex t ∈ V. The flow function, or merely the flow which is defined on each edge of the flow network, is denoted by f: V × V → R. The flow must satisfy the following constraints:

1. Anti-symmetry constraint: The net-flow across any edge (u,v) is zero, i.e.,f(u, v) = − f(v, u), ∀ (u, v) ∈ E.
2. Capacity constraint: The flow along an edge can not exceed the capacity of an edge, i.e., $0 \leq f(u, v) \leq c(u, v)$, ∀ (u, v) ∈ E
3. Conservation constraint: The flow entering into some particular vertex v is the same as the flow leaving out of that particular vertex v, except at the source s and the sink t vertex, i.e., $\sum_{(v,u)\in E} f(v, u) - \sum_{(u,v)\in E} f(u, v) = 0$, ∀ v ∈ V − {s,t}.

The val(f) denotes the value of a flow f and is defined as, val(f) = $\sum_{j \in V} f(s, j) = \sum_{j \in V} f(j, t)$. The flow $f^{'}$ is said to be the maximum flow for the given flow network G if there does not exist any flow $f^{"}$ in the same flow network G such that val($f^{"}$) > val($f^{'}$).

According to the max-flow min-cut theorem, the maximum amount of flow passing from the source to the sink is equal to the total weight of the edges in a minimum cut. Hence, by solving the max-flow problem, we directly solve the min-cut problem as well.

For our problem, the computed min-cut provides the distinct foreground/background segments of the input image where vertices on one side of min-cut denote the foreground pixels while the vertices on another side of min-cut denote the background pixels.

GOVERNING EQUATIONS FOR IMAGE TO GRAPH CONVERSION

We use the approach discussed by Boykov and Funka-Lea [3] to convert the input square grayscale image to the flow network(G). We have one vertex corresponding to each pixel of the image in the flow network. These vertices are known as pixel vertices. In addition to these vertices, we have two extra vertices that act as the source(s) and sink(t) in the flow network.

Two distinct types of edges then connect the above-discussed vertices. First is the N-link edges that connect each vertex with its four neighboring vertices. Vertices at the boundary are also connected with two or three neighboring vertices with the same N-link edges. Second is the t-link edges which connect the source and sink vertex with the vertices that correspond to the pixels that are marked as seeds.

The weight to the N-link edges is assigned carefully. They reflect the inter-pixel similarity; that is, when the two pixels are similar, then the edge connecting them has a high weight assigned to it and vice-versa. In order to achieve this, the weight is computed using the boundary penalty, a function that maps two pixel's intensities to a positive integer. If I_p denotes the intensity of the pixel p, then for edge (p,q) ∈ E the boundary penalty $B(I_p, I_q)$ is computed using the equation 1.

$$B(I_p, I_q) = 100 * exp(\frac{-(I_p - I_q)^2}{2 * \sigma^2}) \quad (1)$$

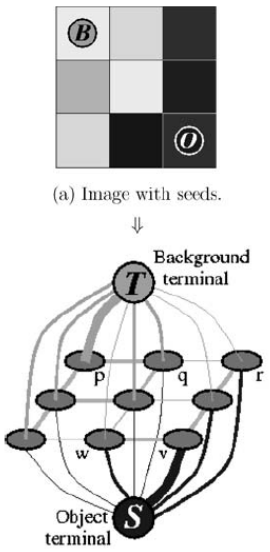


FIGURE 2: An example network defined on a simple 3x3 image with a background seed and a foreground seed

For the experimentation purpose, the value for σ is set to 30. The equation 1 give high values for edge weights when $|I_p - I_q| < \sigma$ and vice-versa. The final edge weight computed is multiplied by 100 and is cast to an integer in the equation 1.

For generating t-links, the user has to specify five background pixels and one foreground pixel. These pixels are known as seeds. For each background seed, the edge is added from the vertex corresponding to that seed pixel to the source node(s) with capacity κ. In a similar manner, for each foreground seed, the edge is added from the vertex corresponding to that seed pixel to the destination node(t) with capacity κ. The value κ is determined using the equation 2.

$$\kappa = 100 * max(\{B(I_p, I_q) | (p,q) \in E\}) \quad (2)$$

Once the flow network is generated as described above from the input square grayscale image, we can apply the parallel push-relabel algorithm discussed in the later sections to compute the min-cut and generate the square grayscale image's precise foreground and background segments.

SERIAL PUSH RELABEL ALGORITHM

The Push-Relabel algorithm [7] is one of the well-known algorithms for computing the maximum flow in the given flow network. Its time complexity is $O(n^2m)$, where n denotes the number of vertices and m denotes the number of edges in the input flow network. It is presented in Algorithm 1.

Algorithm 1: Serial Push-Relabel algorithm

Input: A Flow Network G(V, E, c), source *s*, sink *t*
Output: Maximum flow value

```
1  let e(v) = 0; ∀v ∈ V
2  let h[s] = n ;        // Initialize the heights
3  let h[v] = 0; ∀v ∈ V - {s}
4  create a pre-flow f that saturates all out-going edges of s

5  while ∃ v ≠ s, t with e(v) > 0 do
6  |  choose such a vertex v with the largest height.
7  |  if ∃ (v, w) ∈ E with h(v) = h(w) + 1 then
8  |  |  Push the excess flow from v to w.
9  |  else
10 |  |  Relabel the vertex v.
11 |  end
12 end
13 return e(t);
```

A crucial property of the Push-Relabel algorithm is that it relaxes the conservation constraints (discussed above) by allowing the difference between the incoming and the out-going flows to be non-negative during the execution of the algorithm instead of being strictly zero. This difference between the incoming and the out-going flows associated with each vertex is also termed as excess flow and is denoted as e(v) in Algorithm 1. The Push-Relabel algorithm also associates the height with each vertex, denoted as h[v] in algorithm 1. The higher height vertices can push their excess flow to their neighbor vertices with lower height in the Push-Relabel algorithm. The height for sources vertex is initialized to the number of nodes in the input flow network, while the rest of the vertices' height is initialized to zero. The pre-flow is computed using the above concept by saturating all the out-going edges from the source vertex during the initialization step, as shown by lines 1-4 in Algorithm 1. Hence, pre-flow contains the vertices in the flow network, which will have the positive excess values. When the conservation constraint is again satisfied for all the vertices, pre-flow becomes the maximum flow of the flow network, and the algorithm terminates. At this stage, the excess flow accumulated at the sink vertex, i.e., e(t), denotes the maximum flow value.

GOVERNING PARALLEL PUSH RELABEL ALGORITHM

We use the approach discussed by Zhengyu He and Bo Hong [8] for parallelizing the push-relabel algorithm on multi-core setup. The CPU parallel push-relabel algorithm is presented in Algorithm 2. The value $c_f(u,v)$ denotes the residual capacity of each edge of the flow network G(V, E, c), where $u, v \in V$. For the given flow f and the flow network G(V, E, c), it is calcu-

lated as c(u,v) - f(u,v), where $u, v \in V$. The residual network of a given input flow network is denoted as G_f(V, E_f, c_f), where E_f = $\{(u,v)|u \in V, v \in V, c_f(u,v) > 0\}$.

The initialize routine that is being called as part of the algorithm 2 is presented in the algorithm 3. Here we set the initial values for heights(h), excess flow(e) for each vertex and calculate the residual capacity(c_f) for each edge in the input flow network and generate the residual network. Additionally, the *ExcessTotal* variable is used to keep track of the total amount of the excess flow present at all the vertices of the residual flow network. Since the pre-flow is computed by saturating all the out-going edges from the source vertex during the initialization step, the *ExcessTotal* variable is initialized to the total amount of the excess flow that each vertex connected to the source vertex has during the pre-flow.

Algorithm 2: OpenMP Parallel_Push_Relabel Algorithm

Input: A Flow Network G(V, E, c), source s, sink t
Output: Maximum flow value

```
1  Initialize e, h and c_f
   // Loop until convergence
2  while e(s) + e(t) < ExcessTotal do
3  |  #pragma omp parallel num_threads(thread_count)
4  |  {
   |    1. Divide the vertices among the threads.
   |       Each thread takes responsibility for
   |       a chunk of vertices.
   |    2. Each thread finds the vertex u such that
   |       e(u) > 0 in its assigned chunk of vertices.
   |    3. call push_relabel_kernel(u)
   |  }
5  end
6  return e(t)
```

The main `while` loop (Line 2) in the algorithm 2 consists of a parallel region forking thread_count number of threads. Threads initially divide the vertices among themselves such that each thread works on a chunk of vertices where each chuck is of almost similar size. After that, each thread search for the vertex(u) with a positive excess value in its assigned chunk of vertices. Each thread then calls the push_relabel_kernel with vertex(u) as the argument.

Inside the `push_relabel_kernel` shown in algorithm 4, each thread operates on the vertex(u). Initially, each thread searches for the min-height neighbor(v') of the vertex(u), as shown by lines 6-12. If the height of the min-height neighbor(v') is lesser than the height of vertex(u), then the thread performs the push operation to vertex(v') shown by lines 13-18. Else it does relabel operation by increasing the height of vertex(u) shown by line 20. Once the single iteration performing either push or relabel operation is over, each thread again searches for the vertex(u) with a positive excess value in its assigned chunk of vertices. Each thread continues this process for KERNEL_CYCLES number of times. After every thread finishes KERNEL_CYCLES push or relabel operations, the `push_relabel_kernel` terminates.

Algorithm 3: Initialize Routine

```
1  h(s) ← |V| ;
2  e(s) ← 0 ;
3  foreach u ∈ V − {s} do
4  |  h(u) ← 0 ;
5  |  e(u) ← 0 ;
6  end
7  foreach (u,v) ∈ E do
8  |  c_f(u , v) ← c(u,v) ;
9  |  c_f(v , u) ← c(v,u) ;
10 end
11 foreach (s,u) ∈ E do
12 |  c_f(s , u) ← c_f(s , u) - c(s,u) ;
13 |  c_f(u , s) ← c_f(u , s) - c(s,u) ;
14 |  e(u) ← c(s,u) ;
15 |  ExcessTotal ← ExcessTotal + c(s,u) ;
16 end
```

Once the `push_relabel_kernel` terminates, control is returned back to the algorithm 2 where the algorithm checks for its termination condition, which is stated as follows: If the summation of excess flow value on the source (e(s)) and the excess flow value on the sink (e(t)) is equal to the ExcessTotal, then it implies that there are no active vertices in the residual network (except the source and the sink) and hence the main `while` loop terminates. The final excess flow value on the sink vertex (e(t)) provides the maximum flow.

MAIN ROUTINE

The main routine is shown as algorithm 5. The main routine reads the image as shown in 1. It then converts the input sqaure grayscale image to the flow network(G), as shown in line 2. It then calls the parallel push-relabel algorithm, as shown in line 3. As we are interested in the min-cut, we do one additional step after calling the parallel push-relabel algorithm. By definition of the s/t cut, S contains a set of vertices reachable from source vertex(s) in the residual network, and T contains a set of vertices

Algorithm 4: Parallel Push_Relabel_Kernel using OpenMP

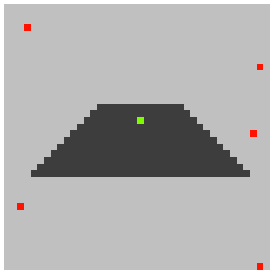
```
1  cycle = KERNEL_CYCLES ;
2  while cycle > 0 do
3      if e(u) > 0 then
4          e' ← e(u) ;
5          h' ← ∞ ;
6          foreach (u,v) ∈ E_f do
7              h" ← h(v) ;
8              if h" < h' then
9                  v' ← v ;
10                 h' ← h";
11             end
12         end
13         if h(u) > h' then
               // Push Operation
14             flow ← min(e' , c_f(u, v'));
15             AtomicAdd(c_f(v' , u), flow);
16             AtomicSub(c_f(u, v'), flow);
17             AtomicAdd(e(v'), flow);
18             AtomicSub(e(u), flow);
19         else
               // Relabel Operation
20             h(u) ← h' + 1
21         end
22     end
23     Find vertex u such that e(u) > 0 in the assigned
         chunk of vertices ;
24     cycle ← cycle - 1 ;
25 end
```

Algorithm 5: Main Routine

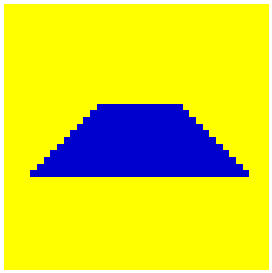
Input: Square grayscale Image(IMG)
Output: Set of foreground, Background pixels

```
1  Read(IMG) ;
2  ImageToGraph(IMG,G) ;
3  Parallel_Push_Relabel(G,s,t) ;
4  BFS(G_f,foreground,background) ;
```

reachable from sink vertex(t) in the residual network. Therefore, we can run a breadth-first search(BFS) from source vertex(s) using the residual network(G_f) and find all the reachable vertices from the source vertex. These vertices correspond to the pixels that are classified as the foreground pixels by parallel_push_relabel algorithm. Pixels corresponding to the rest of the vertices form the set of background pixels. The call to the BFS routine is shown in line 4. After the BFS is finished, we have two sets corresponding to the foreground and background pixels, respectively. Hence the main routine terminates.

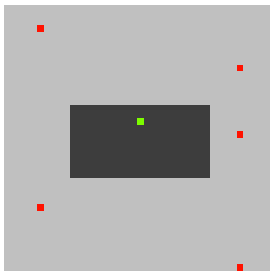


(a) Square grayscale Image of Size 40x40 with Input Seeds from User

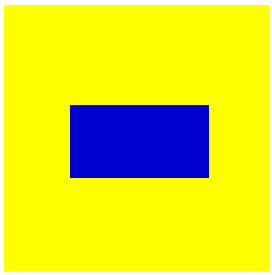


(b) Segmented Image

FIGURE 3: Image Segmentation Result from the Parallel Algorithm



(a) Square grayscale Image of Size 40x40 with Input Seeds from User



(b) Segmented Image

FIGURE 4: Image Segmentation Result from the Parallel Algorithm

RESULTS AND DISCUSSION

In this section we evaluate the performance of the proposed parallel push-relabel algorithm with the exact serial versions of the push-relabel algorithm.

Input Images. We evaluate the performance of our parallel algorithm on the synthetically generated square grayscale images of varying sizes such as 20×20, 30×30, and 40×40. Table 1 lists the various square grayscale images used along with their sizes. Note that the square grayscale images are selected only for the sake of convenience. The same implementation of the parallel algorithm can work on the non-square colored images.

Machine Configuration. We perform experiments on a machine with an Intel Core i5-6200U @ 2.30GHz CPU having 8

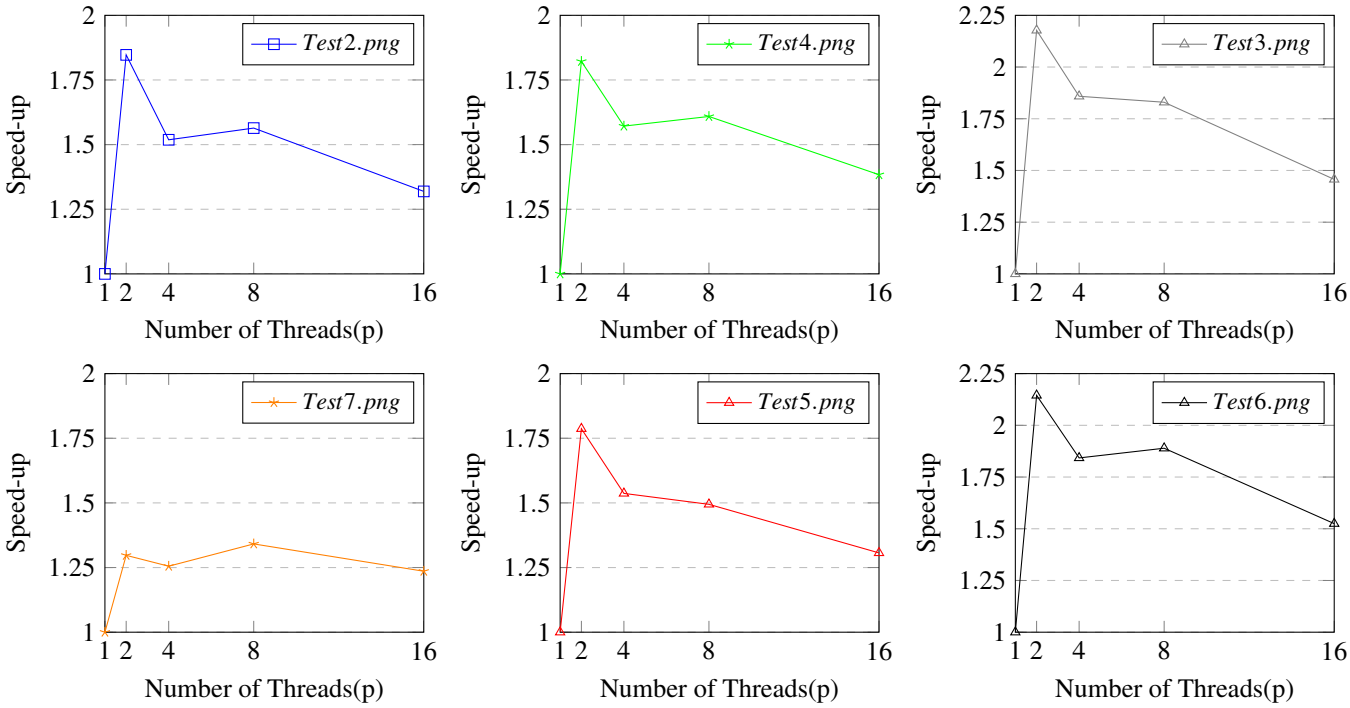
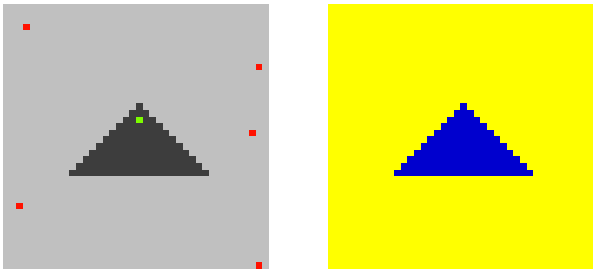


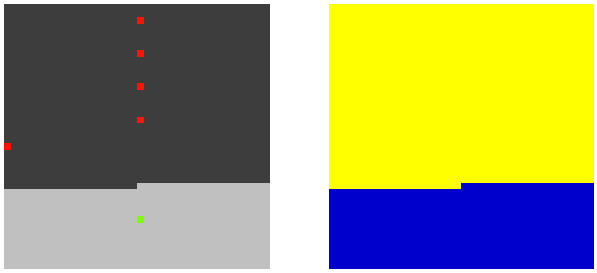
FIGURE 5: Effect of varying the number of threads on the speed-up for various grayscale images



(a) Square grayscale Image of Size 40x40 with Input Seeds from User

(b) Segmented Image

FIGURE 6: Image Segmentation Result from the Parallel Algorithm



(a) Square grayscale Image of Size 40x40 with Input Seeds from User

(b) Segmented Image

FIGURE 7: Image Segmentation Result from the Parallel Algorithm

GB RAM and four processing cores.. The machine runs Ubuntu Debian 18.04.5 LTS (64-bit).

Figures 3, 4, 6, 7 show the resulting segmented images corresponding to the square grayscale images of the size 40×40 provided as input. The **yellow color** indicates the pixels that are classified as background pixels by the parallel algorithm. In contrast, the **blue color** indicates the pixels classified as foreground pixels by the parallel algorithm. In the input square grayscale images shown as part of figures 3, 4, 6, 7 the **red dots** in the image show the pixels that are marked as background seeds by the user. Similarly, **green dots** in the input square grayscale images show the pixels that are marked as foreground seeds by the user.

Based on the results shown as part of the figures 3, 4, 6, 7, we can conclude that the parallel algorithm provides the precise segments of the input images with no loss in the accuracy of the results.

Figure 5 shows the effect of varying the number of threads(p) on the speed-up for various input grayscale images

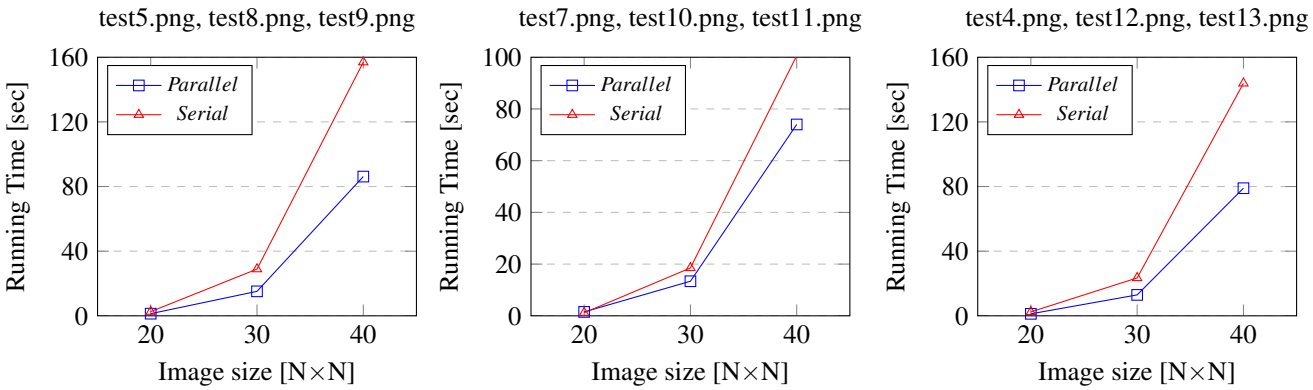


FIGURE 8: Effect of Increasing Input Image Size on the Serial and Parallel Algorithm's performance

Image Name	Image Size	Image Type
test1.png	40×40	Square grayscale
test2.png	40×40	Square grayscale
test3.png	40×40	Square grayscale
test4.png	40×40	Square grayscale
test5.png	40×40	Square grayscale
test6.png	40×40	Square grayscale
test7.png	40×40	Square grayscale
test8.png	30×30	Square grayscale
test9.png	20×20	Square grayscale
test10.png	30×30	Square grayscale
test11.png	20×20	Square grayscale
test12.png	30×30	Square grayscale
test13.png	20×20	Square grayscale

TABLE 1: Image Dataset

of the size 40×40. Based on the result, it is noted that the parallel algorithm achieves the maximum speed-up for p=2 for most of the input images of the size 40×40. However, for a few of the 40×40 sized images, the parallel algorithm achieves its maximum speed-up for p=8.

Figure 8 shows the effect of increasing input image size on the serial and parallel algorithm's performance for various input images. From figure 8, we can conclude that the parallel algorithm performs consistently better than the serial algorithm for various images of size 20×20, 30×30, and 40×40.

CONCLUSIONS

As part of this work, we presented the CPU parallel push-relabel algorithm for computing the graph cuts. We further applied our parallel algorithm to the problem of image segmentation. We observed that the presented parallel algorithm gives the precise segments of the input square grayscale images of various sizes with no loss in the accuracy of the results. The presented parallel algorithm is up to 2.17× faster than the serial algorithm and performs consistently better than the serial algorithm for various input square grayscale images.

.

ACKNOWLEDGMENT

Thanks go to Dr. Kameswararao Anupindi and Dr. Rupesh Nasre for their support.

REFERENCES

[1] Eriksson, A., Barr, O., and Åström, K., 2006. "Image segmentation using minimal graph cuts".

[2] Boykov, Y., and Jolly, M.-P., 2001. "Interactive graph cuts for optimal boundary region segmentation of objects in n-d images". In Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001, Vol. 1, pp. 105–112 vol.1.

[3] Boykov, Y., and Funka-Lea, G., 2006. "Graph cuts and efficient nd image segmentation". *International Journal of Computer Vision - IJCV,* ***70***, 11, pp. 109–131.

[4] Yan, D., and Yongzhuang, M., 2007. "Image restoration using graph cuts with adaptive smoothing". In 2007 International Conference on Information Acquisition, pp. 152–156.

[5] Zureiki, A., Devy, M., and Chatila, R., 2008. *Stereo Matching and Graph Cuts*. 11.

[6] Hong, L., and Chen, G., 2004. "Segment-based stereo matching using graph cuts". In Proceedings of the 2004

IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004., Vol. 1, pp. I–I.

[7] Goldberg, A. V., and Tarjan, R. E., 1988. "A new approach to the maximum-flow problem". *J. ACM,* ***35***(4), Oct., p. 921–940.

[8] He, Z., and Hong, B., 2010. "Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms". In 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–10.