

## Question 1

- Needs Grading

Give two different approaches to parallel implementation of convolution with a finite (and small) kernel assuming distributed memory. Discuss advantages and disadvantages.

Selected - By using simple sequential perblock for small kernels

Answer:

- By moving the inner loops over the kernel outwards we can reuse parallel points operator and shifts

Correct 

Answer:

The most straightforward approach is to use compute the output values in  $P$  disjoint sections of equal size, with  $P$  the number of processors. Each processor is given an area of the input image the corresponding to the desired output area dilated by the support of the filter kernel. In other words, it gets the desired output area, plus all pixels just outside its boundary required to compute the output pixels along the boundary. This is simple, and requires no barriers to compute the final result, although it does incur some memory overhead (pixels in the boundary region are copied to multiple processors).

The second method relies on the ability to perform point operations and image shifts in parallel very quickly. Assuming there is an efficient library for these point and shift operators, we can implement convolution by setting the output to zero, repeatedly shifting the image by a vector corresponding to each position in the support of the kernel, multiplying the input by the corresponding kernel value, and then adding that to the output. The advantage is that we can make use of a pre-existing library, the disadvantage is that there are multiple barriers per point in the support of the kernel

## 1. Question 2

- Needs Grading

Connected-component labelling by flood-filling works by moving through the image, until it finds an unlabelled pixel, assigning it a new, arbitrary label, and then recursively or iteratively filling all unlabelled neighbours of the same grey value or colour with the same label. Give two reasons why this is unsuitable for parallel implementation. Can you suggest a solution that would allow flood filling to be used in parallel in principle?

Selected - FIFO can not be parallelized

Answer:

- Divide the image into arbitrary section does not work because it will cut the components into multiple sections

- It doesn't work very well on binary images
- Processing per grey level has problems with load balancing

Flood filling can be used to build max tree.

Correct 

Answer:

First of all, flood filling is a queue based algorithm, i.e. it uses either a FIFO or priority queue to dictate the order of processing. This is an inherently sequential approach. Pixels cannot be processed simultaneously because I cannot determine before the flood-filling of a component is finished whether or not two pixels should receive the same label.

The second issue is related. Suppose I cut the image into disjoint sections, and perform flood filling on each section. The problem that arises then is that different parts of the same connected component will receive different labels, and different connected components might receive the same label. In either case, these label conflicts need to be resolved, requiring (potentially multiple) passes of relabelling.

One way around these problems is not to use arbitrary labels, but instead using the index of the first pixel of each component you find as the label. This effectively creates Union-Find trees consisting of groups of pixels all pointing to the first element of the connected component. In this case, we can use flood filling on disjoint image sections, and then later apply the union find approach to glue the results together

## 1. Question 3

- Needs Grading

In Union-Find, explain how union-by-rank works, and why it is important. Give a pseudo-code example of how it might be implemented.

Selected union-by-rank works by adding rank to each item.

Answer:

It's important because it keeps the tree balance and the rank maintain the dept of the tree.

Pseudo code:

```
procedure Union(x,y)
```

```

xr <- Find (x)
yr <- Find (y)
if xr != yr then
  if xr.rank > yr.rank then
    yr.parent = xr;
  else
    if yr.rank > xr.rank then
      xr.parent = yr;
    else
      xr.parent = yr;
      xr.rank ++;
    end if
  end if
end if
end procedure

```

Correct 

Answer:

Union-by-rank is a method to keep the depth of trees low, by always linking the root of the least deep tree to that of the deeper one. It needs an extra field "rank" per element, which is initialised at zero in MakeSet, i.e.

MakeSet(x)

x.parent  $\leftarrow$  x;

x.rank  $\leftarrow$  0;

End

In the Union function we check the ranks of the roots xroot and yroot of either tree, and if the rank of xroot is smaller than that of yroot, link xroot to yroot, if the root of yroot is smaller than that of xroot, link yroot to xroot, if they are equal, feel free to choose any order of linking, but increment the rank of the resulting root by 1:

```

Union(x,y)

  xroot ← Find(x);

  yroot ← Find(y);

  if xroot ≠ yroot then

    if yroot.rank > xroot.rank then

      xroot.parent ← yroot;

    else

      yroot.parent ← xroot;

      if xroot.rank = yroot.rank then

        xroot.rank ← xroot.rank + 1;

      endif;

    endif;

  endif;

end;

```

## 1. Question 4


- Needs Grading

The original parallel max-tree algorithm for shared memory parallel machines works by merging max-trees computed on disjoint sets of the image. It is restricted to images with at most 16 bits per pixel. For larger values the performance plummets, and sequential computation is faster. Why is this?

Selecte Because each merge process of tree quite heavy. The omplexity is O  
d (Ainterface min(G,Npaxes/Nblocks))

Answer:

The number of merge processes is  $\log P$ , where  $P$  is the number of processors

Correct   
Answer:

The merger process inspects all pairs of adjacent pixels along the boundary of two sections to be merged. Starting from the top, it uses a merge-sort like process which visits all pixels along both paths from the current grey level down to the root, or (if the first pair of pixels has been processed) until the first point at which the trees have already merged. Worst case, this scales linearly with the number of grey levels in the data. The complexity of a merge is therefore  $O(A \min(N/P, 2b))$  with  $A$  the area of the interface between the two sections (or length of the boundary for 2D images),  $N$  the total number of pixels or voxels,  $P$  the number of processes, and  $b$  the number of bits per pixel. Thus, for large enough data sets, the overhead of merging at some point outweighs the benefits of parallel construction of individual max-trees.


## 1. Question 5

- Needs Grading

Describe four ways in which computation of the maps used for rubble detection are speeded up compared to the repeated application of reconstruction filters using the classic  $O(N^2)$  sequential algorithm.

Selected -

Answer:

Correct 

Answer:

- 1) Replace the quadratic algorithm with computation by max-trees: these are either  $O(GN)$  with  $G$  the number of grey levels, or  $O(N \log N)$ , depending on the algorithm
- 2) Compute the max-tree *just once* and re-use it for all computations of reconstruction filters: building the tree costs most of the time compared to filtering
- 3) Use the parallel algorithm for computation, either using distributed or shared-memory, this can produce near linear speed-up for large data sets
- 4) Do not iteratively compute each output image in parallel, but use a single, more complicated filtering algorithm to output the entire stack of images

Other correct options:

- 5) Use area openings rather than reconstruction filters. This is much faster because no marker images need be calculated by (parallel) erosions.

6) Compute the map directly from the max-tree, rather than first computing all the intermediate images, differencing them, and selecting the scale representing the maximum