university of groningen

Supercomputing for Giga- and Tera-Pixel Image Scales, Part II: Connectivity and Connected Filters

Michael H. F. Wilkinson

Johann Bernoulli Institute
University of Groningen

Overview

- Connected Component Labelling
  - Sequential algorithms
  - Parallel algorithms
- Connected Filters
  - Binary case
  - Grey scale
- Algorithms for connected filters
  - Sequential algorithms
  - Low dynamic range shared-memory
  - High dynamic range shared memory
  - Low dynamic range distributed memory

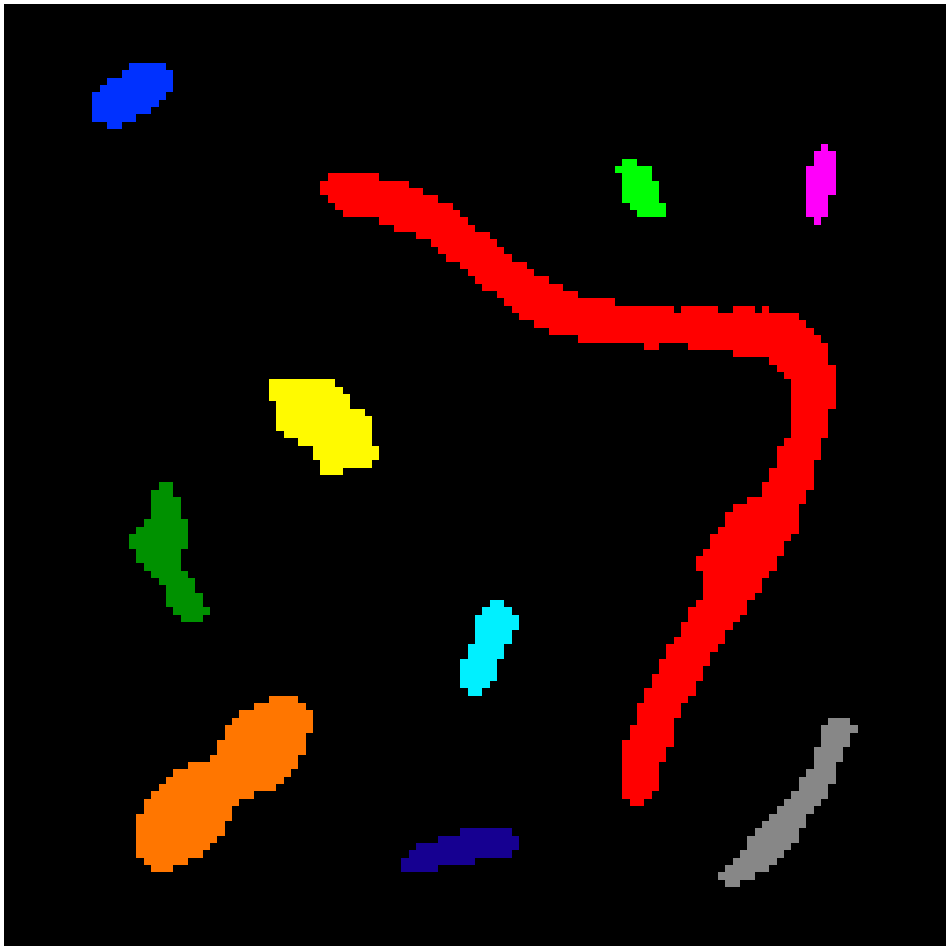
Connected Components

- Connected components are *connected sets* of pixels in the image of a constant value of *maximum extent*.
- This means that for every connected component C of image f
  - For ever pixel $p \in C$ there are no neighbours q such that $q \notin C$ and $f(p) = f(q)$
  - For ever pair of pixels $p, q \in C$ there exists a path $\{x_0, x_1, \ldots, x_n\} \subseteq C$, such that $x_0 = p$, $x_n = q$, for all x_i $f(x_i) = f(p)$, and for all $i \in \{1, 2, \ldots, n\}$ we have x_{i-1} and x_i are neighbours.
- Connected components partition the image domain into different *equivalence classes*

Connected Component Labelling

- Assigns labels to all pixels such that:
  - All the pixels from the *same* connected component have the *same* label
  - All pixels from *different* connected components have *different* labels
- Essential step in many segmentation methods

Example



Types of algorithm

- Equivalence table methods
  - Earliest
  - Memory hungry!
- Flood filling
  - Simple and effective
- Union-Find
  - Efficient in both time and storage
  - Simple, but strange!

The Flood-fill Algorithm

- We assume an input image f of N pixels
- We require a queue Q of size $O(N)$.
- We require a label image L of size N (typically unsigned int or long).
- Convention: label 0 means "unlabelled"

The Flood-fill Algorithm II


```
CurLabel ← 0;                                          ▷ No current label in use
for all pixels p do
    L[p] ← 0;                                          ▷ All pixels unlabelled
end for
for p ← 0 to N − 1 do
    if L[p] = 0 then                                   ▷ Pixels not labelled yet
        CurLabel ++;                                   ▷ Create new label
        PUSH(p, Q);
        while not ISEMPTY(Q) do                        ▷ Start flooding
            curpix ← POP(Q);
            L[curpix] ← CurLabel;
            for all neighbours n of curpix do
                if f[n] = f[curpix] then               ▷ Test for equivalence
                    PUSH(n, Q);                        ▷ Push into queue
                end if
            end for
        end while
    end if
end for
```

The Flood-fill Algorithm III

- A key issue is the use of a FIFO queue: cannot be parallelized
- Dividing the image into arbitrary sections does not work:
  - This will inevitably cut components into multiple sections
  - These will not get the same label
  - Some costly relabelling will be needed to correct for this
- Processing per grey level has problems with load balancing
- It will not work on binary image very well

- Designed to deal efficiently with disjoint sets
- Represents disjoint sets as a forest
- Items sharing the same root are in the same set
- Relies on three basic functions
  - $\textsc{MakeSet}(x)$: Initialises item x as a singleton set
  - $\textsc{Union}(x, y)$: Merges trees containing x and y
  - $\textsc{Find}(x)$: Find the root of the tree containing x

```
procedure MakeSet(x)
    x.parent ← x                    ▷ Convention: the root is its own parent
end procedure
```

```
function Find(x)
    if x.parent = x then                                  ▷ x is the root
        return x;
    else
        return Find(x.parent);                  ▷ Recursively seek the root
    end if
end function
```

procedure UNION(x,y)
    $xr \leftarrow$ FIND(x); ▷ Find the roots
    $yr \leftarrow$ FIND(y);
    if $xr \neq yr$ **then** ▷ If not already in same set
        xr.parent $\leftarrow yr$; ▷ Merge two sets
    end if
end procedure

Path Compression

During find, it pays to flatten the tree, by letting all pointers along the root path point directly to the root:

function FIND(x)

  if x.parent $\neq x$ **then** ▷ x is not the root
    x.parent $\leftarrow$FIND(x.parent); ▷ Recursively seek the root
  end if

  return x.parent

end function

Or, non-recursively


```
function FIND(x)
    r ← x
    while r.parent ≠ r do
        r ← r.parent;                    ▷ Iteratively seek the root
    end while

                                         ▷ r is now root

    while x.parent ≠ r do
        s ← x.parent;                    ▷ Save current parent
        x.parent ← r;                    ▷ Update parent to root
        x ← s
    end while

    return r;

end function
```

Union-by-Rank

A further improvement adds a *rank* to each item, to keep trees balanced:

procedure MAKESET(x)
    x.parent $\leftarrow x$ ▷ Convention: the root is its own parent
    x.rank $\leftarrow 0$; ▷ Rank starts at zero
end procedure

The rank maintains the depth of the tree (unless path-compression is also use)

Adapted Union


```
procedure UNION(x,y)
    xr ← FIND(x);                                  ▷ Find the roots
    yr ← FIND(y);
    if xr ≠ yr then                                ▷ If not already in same set
        if xr.rank > yr.rank then
            yr.parent ← xr;                        ▷ Link y to x
        else
            if yr.rank > xr.rank then
                xr.parent ← yr;                    ▷ Link x to y
            else
                xr.parent ← yr;                    ▷ You can choose any order
                yr.rank + +;
            end if
        end if
    end if
end procedure
```

Union-Find Complexity

- The storage complexity of Union-Find is $O(N)$
- If we use path compression *or* union-by-rank we have $O(N \log N)$ computational complexity
- If we use path compression *and* union-by-rank we have $O(N\alpha(N))$ computational complexity
- α is the *inverse Ackermann function* which grows *extremely slowly*
- For certain processing orders, it becomes $O(N)$ in time complexity, even without union-by-rank

CC-labelling by Union-Find

Phase 1: Union-Find

for $p \leftarrow 0$ **to** $N - 1$ **do**
    MakeSet(p); ▷ Initialize p

    for all neighbours q of p with $q < p$ **do** ▷ Only *previously* ▷ processed neighbours

        if $f(p) = f(q)$ **then** ▷ If same grey scale
            Union(p,q); ▷ Merge
        end if
    end for
end for

Note: The order of processing does not matter

Phase 2: Resolve, assume all $L[p] = 0$, and curlabel$= 0$

for $p \leftarrow 0$ **to** $N - 1$ **do**
  $r \leftarrow p$
  while $L[r] = 0$ **and** $r \neq r$.parent **do** ▷ Find root, or labelled
    ▷ pixel along root path
    $r \leftarrow r$.parent
  end while
  if $L[r] = 0$ **then** ▷ We have an unlabelled root
    curlabel++; ▷ Create new label
    $L[r] \leftarrow$ curlabel; ▷ Assign to root
  end if ▷ $L[r]$ now contains the correct label
  $s \leftarrow p$
  while $s \neq r$ **do** ▷ Assign label to pixels along root path
    $L[s] \leftarrow L[r]$
    $s \leftarrow s$.parent
  end while
end for

- Because the order of the union operations does not matter, we can easily perform them in parallel
- Simply split the image into a number of sections
- Perform phase 1 of union-find CC on each section
- Hierarchically merge the sections, by processing the neighbours along the borders
- Once merged, perform resolving in parallel

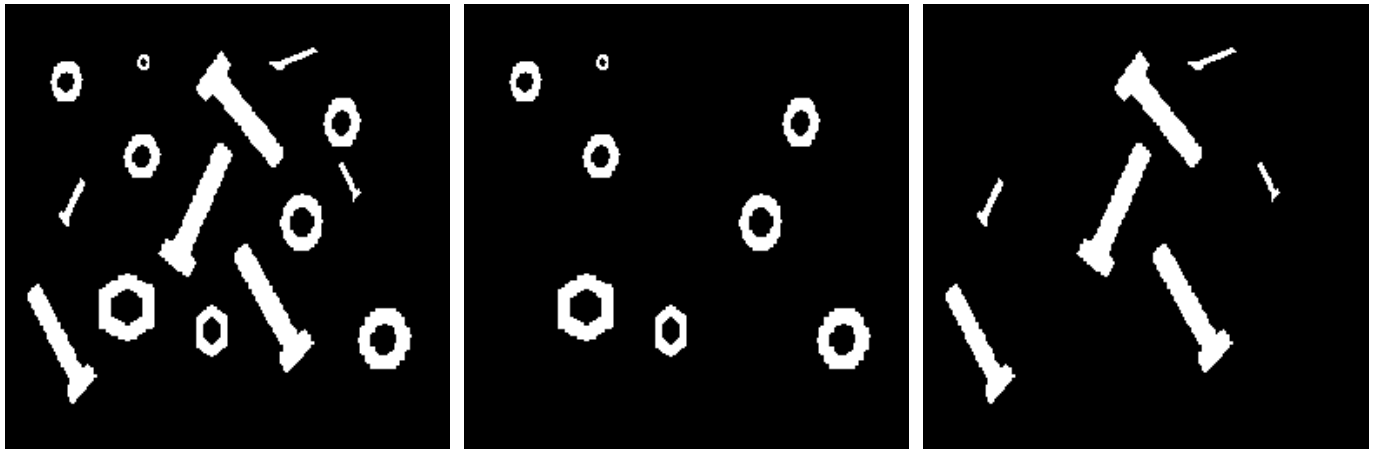
Area Openings: Binary Case

- In the binary case, the area opening removes all connected *foreground* components below a certain area
- We first compute all connected fore ground components
- Compute the area of each
- Assign each pixel of each foreground component a grey level equal to the area
- Threshold the resulting image at some threshold λ

- Introduced by Breen and Jones in 1996.
- Examples: area openings/closings, attribute openings, shape filters
- How do they work?

Binary image :

1. compute attribute for each connected component
2. keep components of which attribute value exceeds some threshold λ



Attribute Openings: Formally

- Let $T : \mathcal{P}(E) \rightarrow \{false, true\}$ be increasing, i.e. $C \subseteq D$ implies $T(C) \Rightarrow T(D)$.
- A binary *trivial opening* $\Gamma_T : \mathcal{P}(E) \rightarrow \mathcal{P}(E)$ using T as above is defined as

$$\Gamma_T(C) = \begin{cases} C & \text{if } T(C), \\ \emptyset & \text{otherwise.} \end{cases} \tag{1}$$

- A typical form of T is

$$T(C) = (\mu(C) \geq \lambda) \tag{2}$$

with μ an increasing scalar (i.e. $C \subseteq D \Rightarrow \mu(C) \leq \mu(D)$), and λ the *attribute threshold*.

- The binary *attribute opening* Γ^T is defined as

$$\Gamma^T(X) = \bigcup_{x \in X} \Gamma_T(\Gamma_x(X)), \tag{3}$$

Attribute Openings: Examples

X

$T = A(C) \geq 11^2$

$T = I(C) \geq 11^4/6$

- An *area opening* is obtained if the criterion $T = A(C) \geq \lambda$, with A the area of the connected set C.
- A *moment-of-inertia opening* is obtained if the criterium is of the form $T = I(C) \geq \lambda$, with I the moment of inertia.

The Grey-scale Case

- In the case of attribute openings, generalization to grey scale is achieved through *threshold decomposition*.
- A threshold set X_h of grey level image (function) f is defined as

$$X_h(f) = \{x \in E | f(x) \geq h\}. \tag{4}$$

- The grey scale attribute opening γ^T based on binary counterpart Γ^T is given by

$$(\gamma^T(f))(x) = \sup\{h \leq f(x) | x \in \Gamma^T(X_h(f))\} \tag{5}$$

- Closings ψ^T are defined by duality:

$$\psi^T(f) = -\gamma^T(-f). \tag{6}$$

- The non-increasing case will be dealt with after discussing the algorithms.

Example Area Filters

Lenna with noise (left) structural open-close with square S.E. (middle) area open-close (right)

Area Opening by Union-Find

- We need to build connected components starting at the highest grey levels
- We grow these components by adding new pixels, potentially at lower grey levels *until* the area is large enough.
- The root of such a component should "contain" the final grey level
- The root of each component should also contain the area of the CC
- Any connected component with a neighbour of larger grey level and area $\geq \lambda$ should remain unchanged!

The Area Opening by Union-Find II

- To achieve these goals, sort pixels in descending grey level order.
- Pixels of the same grey level are stored in lexicographic order
- As we perform unions, we always maintain the root as *the last pixel added*
- We let

$$x.\text{parent} = -\textsc{Area}(CC(x))$$

  A negative parent flags a root!
- During unions, we simply add the parent values to maintain the area
- Merges of roots of differing grey levels only take place if the one with the larger grey level has an area smaller than λ

Area Opening MakeSet and Find


```
procedure MakeSet(x)
    x.parent ← −1                          ▷ Area of singleton = 1
end procedure

function Find(x)

    if x.parent ≥ 0 then                   ▷ x is not the root
        x.parent ←Find(x.parent);          ▷ Recursively seek the root
        return x.parent
    else
        return x                           ▷ x is the root, but x.parent is not!
    end if
end function
```

Area Opening Union

Require: $x \prec y$ ▷ x earlier in processing order
Require: y is a root

procedure UNION(x,y)
  $xr \leftarrow \text{FIND}(x)$;
  if $xr \neq y$ **then** ▷ If not already in same set
    if $f(xr) > f(y) \wedge -xr.\text{parent} \geq \lambda$ **then**
      $y.\text{parent} = -\lambda$; ▷ Block further merges
    else
      $y.\text{parent} \leftarrow y.\text{parent} + xr.\text{parent}$; ▷ Add areas
      $xr.\text{parent} \leftarrow y$; ▷ Always point to last processed
    end if
  end if
end procedure

Area Opening by Union-Find

Sort all pixels in descending grey level order
for All pixels p in sort order **do**
    MakeSet(p);
    for All neighbours q of p such that $q \prec p$ **do**
        Union(q, p);
    end for
end for
for All pixels p in reverse sort order **do** ▷ Resolve
    if p.parent < 0 **then** ▷ We have a root
        Out$[p] \leftarrow f[p]$ ▷ $f[p]$ is the lowest grey level
    else
        Out$[p] \leftarrow$ Out$[p$.parent$]$; ▷ The parent has already been resolved
    end if
end for

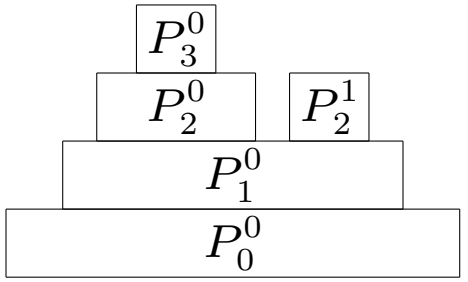
A Parallel Version?

- A spatial division of labour has problems:
  - We cannot compute the areas of CCs correctly per section
  - We would need a barrier per grey level to link different sections first
  - Even if the number of grey levels $G = 256$ this kills performance
- Processing a series of G binary images and stacking them is possible, but:
  - Performing a sequential grey level area opening is as fast as a binary one
  - Overhead from computing thresholds would kill performance

Max-Tree Representation

0	1	2	3	2	1	2	1	0

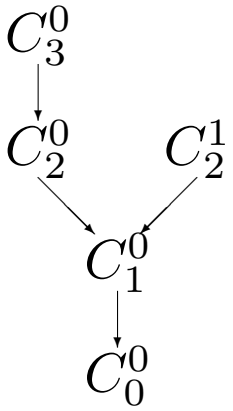
input signal



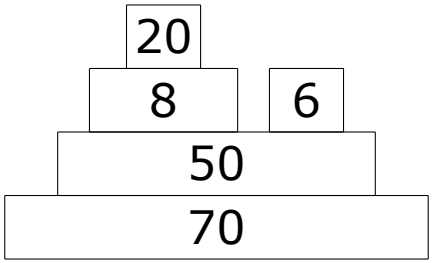
peak components

C_0^0	C_1^0	C_2^0	C_3^0	C_2^0	C_1^0	C_2^1	C_1^0	C_0^0

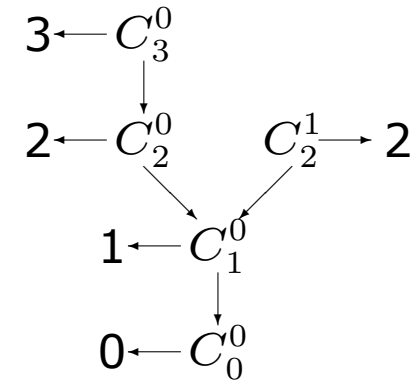
labelling

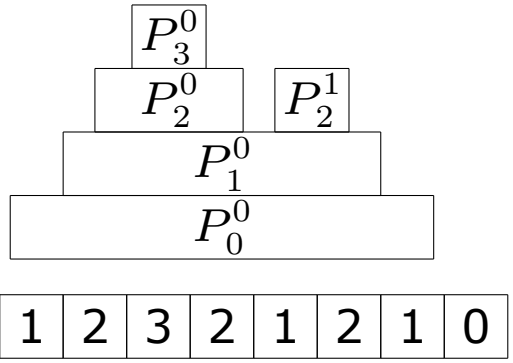


Max-Tree



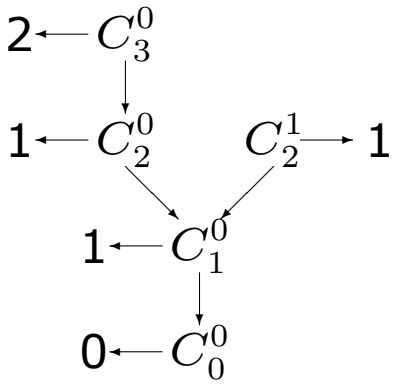
attributes

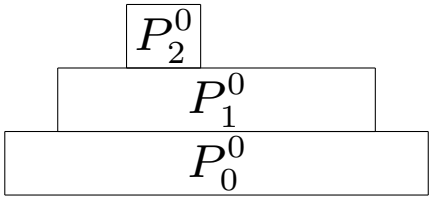




0	1	2	3	2	1	2	1	0

original

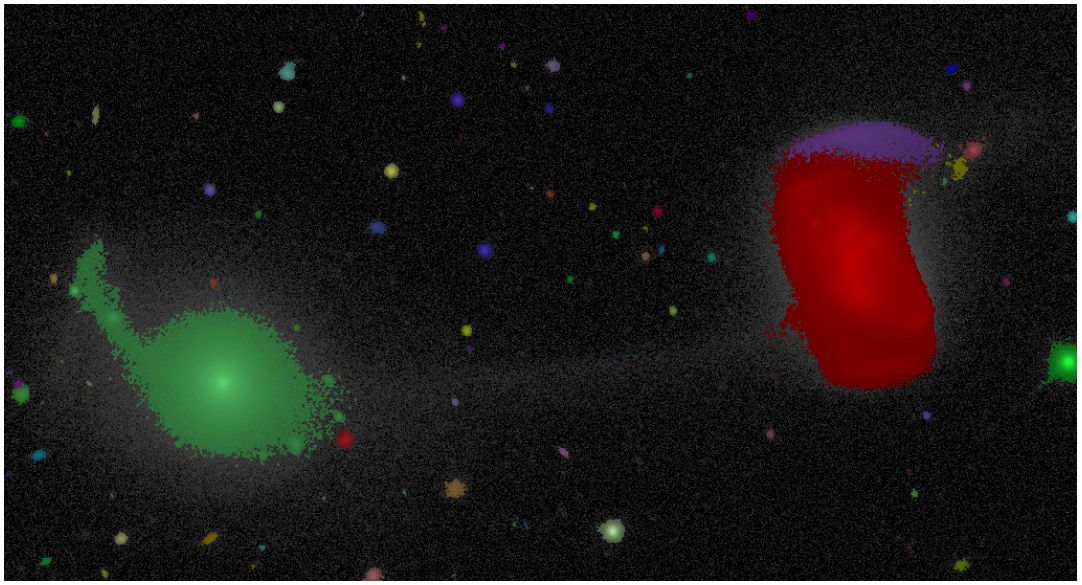




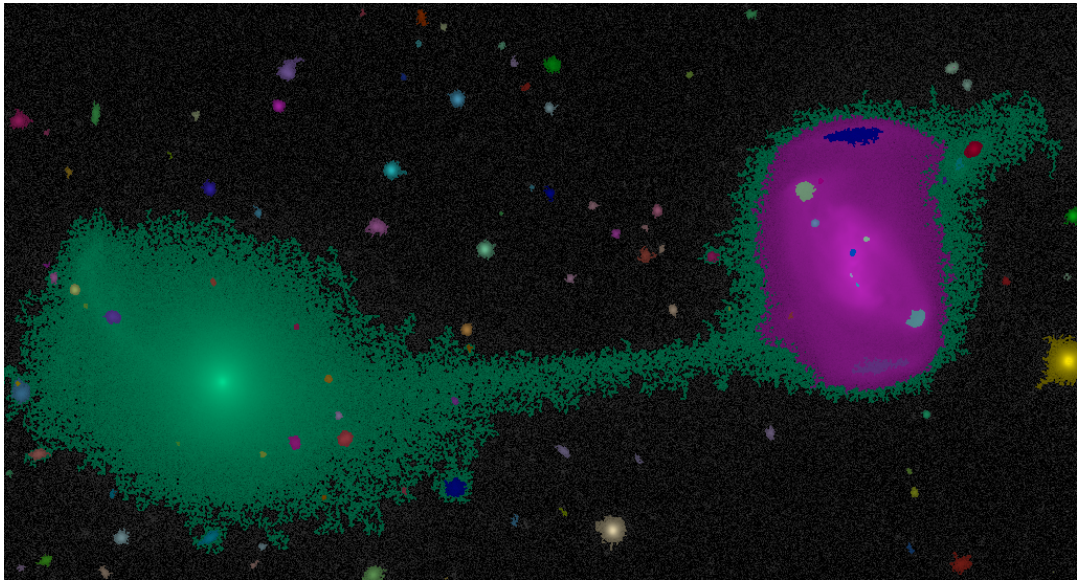
0	1	1	2	1	1	1	1	0

filtered $(\lambda = 10)$

Astronomical Applications

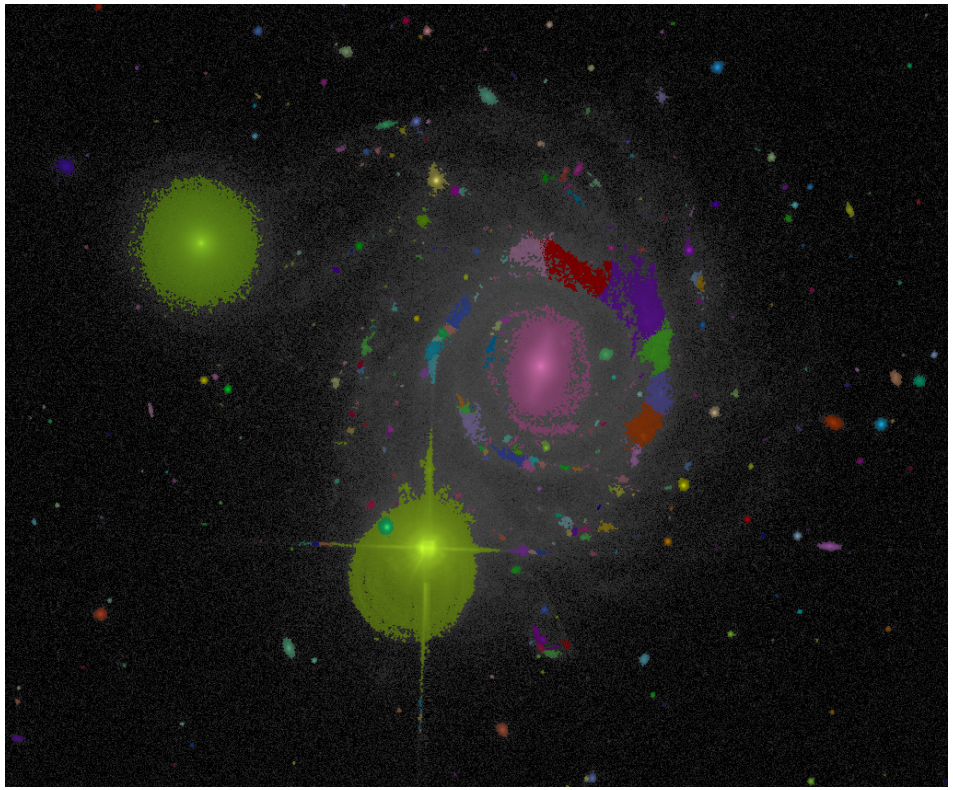


SExtractor

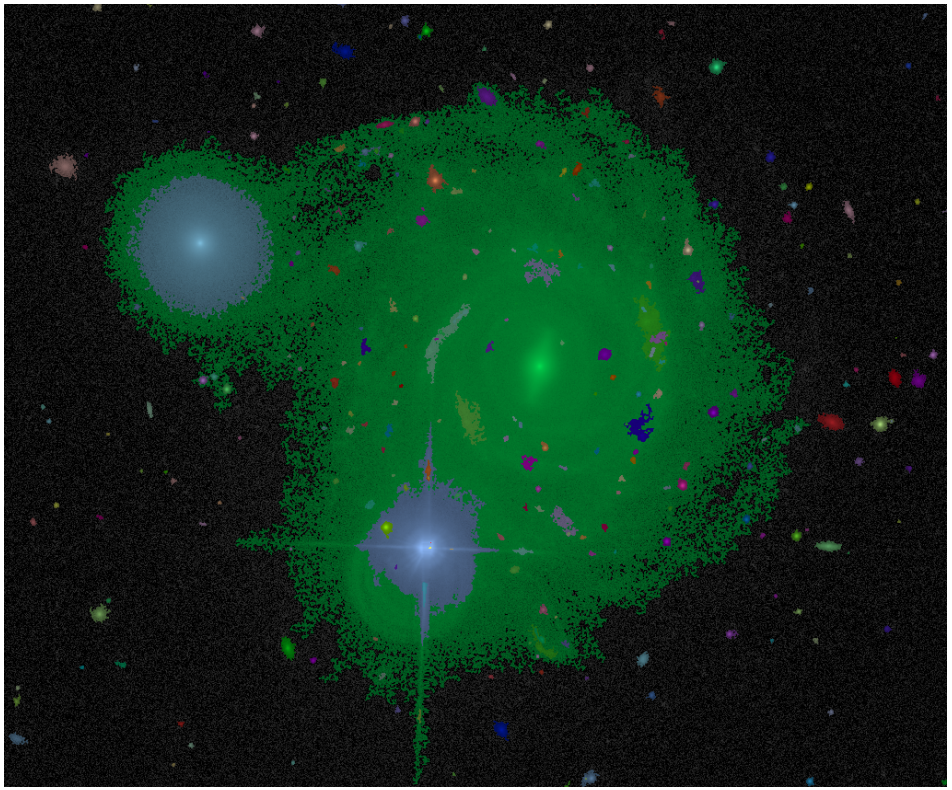


MTObjects

Astronomical Applications

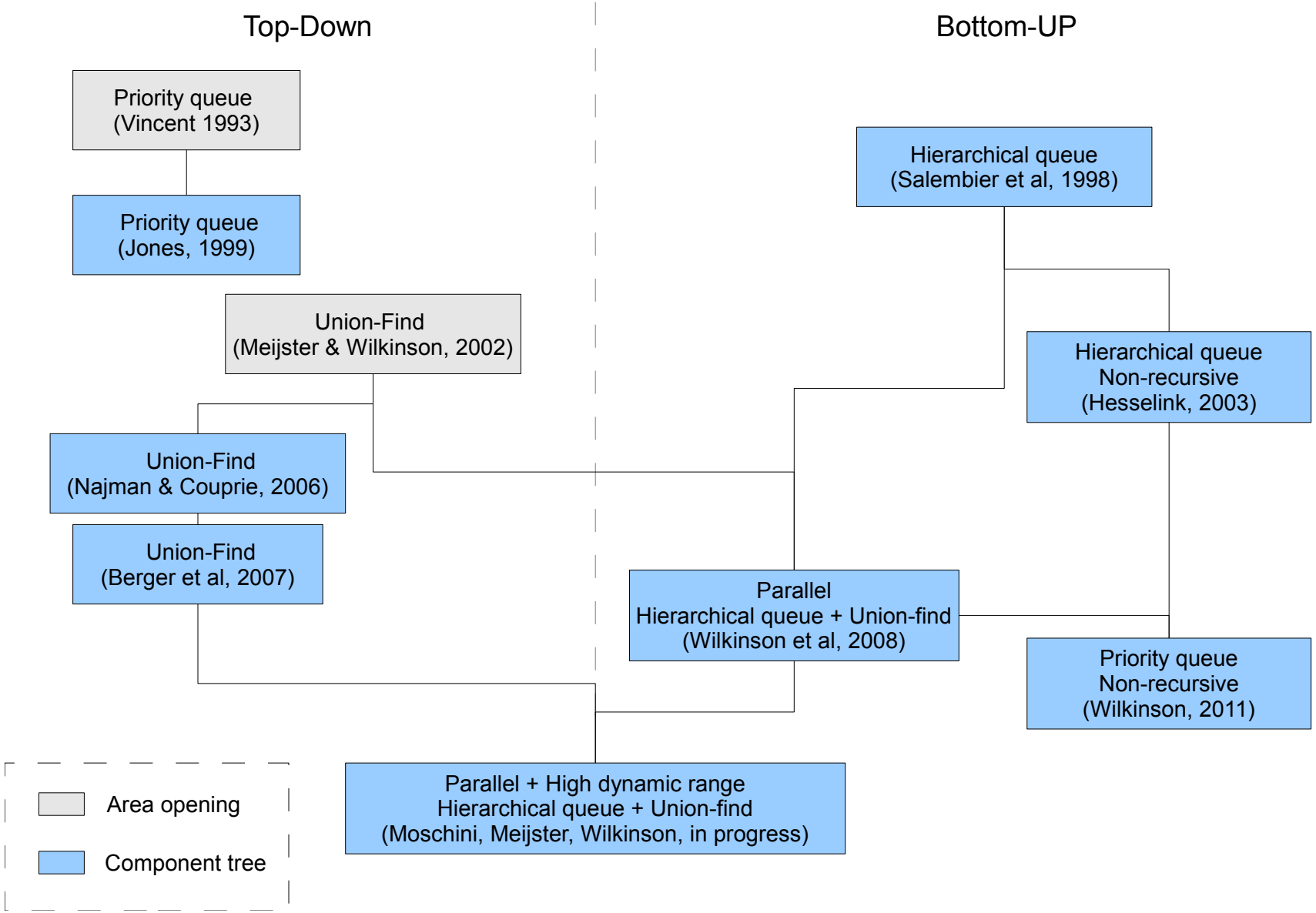


SExtractor

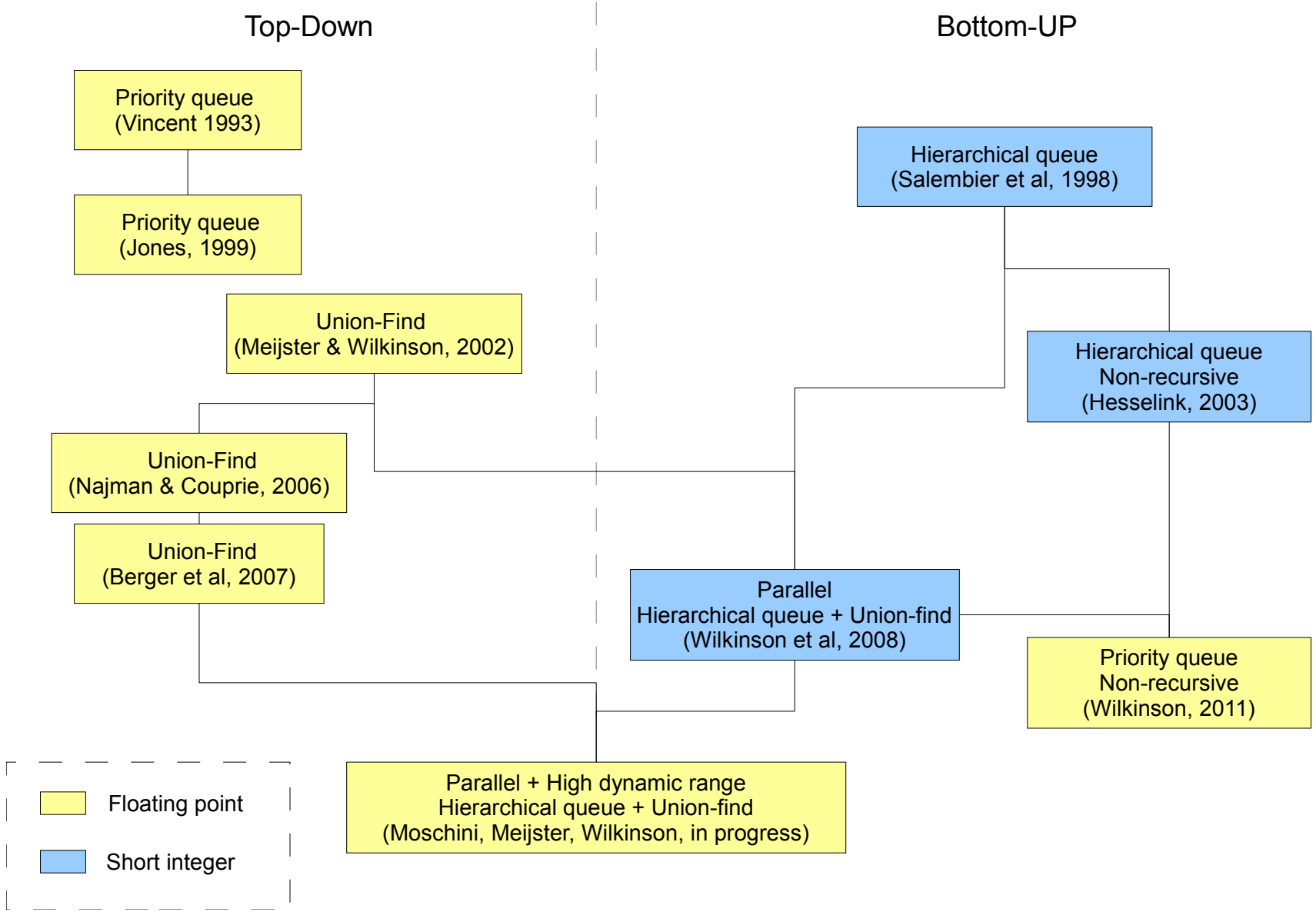


MTObjects

Taxonomy: Top-down vs Bottom-up



Taxonomy: Float vs Short



Salembier's Algorithm

- Salembier et al. developed an efficient algorithm to compute the Max-Tree
- It requires:
  - hierarchical queue: one FIFO queue per grey level
  - a boolean array NodeAtLevel
  - an array NumNodesAtLevel
  - a integer array Status of the same size as the image
- All four structures are cleared before the algorithm starts

Salembier's Algorithm II

- We start by selecting a pixel at the lowest grey level $h_{\min}$ in the image
- This is inserted at $h_{\min}$ in the hierarchical queue
- A recursive, depth first flooding algorithm is called at level $h_{\min}$
- Attributes are computed as the tree is built
- Filtering is implemented as a simple look-up, where the combination of Status$[p]$ and $f[p]$ determine to which node pixel p belongs

Salembier's Algorithm III

- The algorithm is the fastest for building Max-Trees for low dynamic range images sequentially
- The fact that it is queue-based makes it hard to parallellize
- More importantly, the use of arbitrary labels means merging Max-Trees of image sections is costly
- The fact that a FIFO-queue is needed per grey level (and two arrays of length G means the algorithm will not work on extreme dynamic range images ($>$16 bits per pixel)
- A non-recursive alternative using a stack and a priority queue exists to solve the latter problem

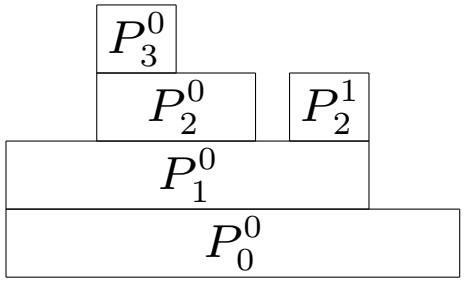
Parallel Computation

- We have developed a parallel algorithm:
  - divide the image (or volume) into strips
  - compute a Max-tree for each strip
  - merge the local Max-trees into a single tree
  - performing the filtering strip-wise
- To do this efficiently, we need to merge the Max-tree and union-find approaches.
- The big problem is keeping the attributes correct

Including Union-find

1	1	3	2	2	1	2	0	0

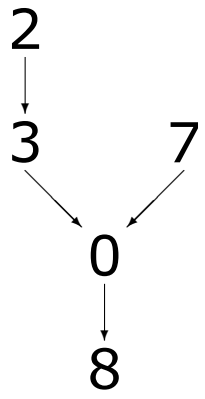
input signal



peak components

0	0	2	3	3	0	7	8	8

labelling



Max-Tree

Salembier with Union-Find

- We store the Max-Tree as an array of nodes of the same size as the image
- Each node has a parent field which contains the index of its parent
- Use the first pixel p in each node found as *canonical element*
- Fill the parent field of all pixels of that node with the index p
- The parent pointer of p is set to the canonical element of its parent in the Max-Tree
- Such pixels are distinguished by the fact that $f[p] > f[p.\text{parent}]$
- We call these pixels *level roots*
- The root of an entire Max-Tree has a parent with special index $\perp$

Divide and Conquer!

procedure FLOOD(Level lev, Partition P, Attribute a)
  $at \leftarrow a$;
  while $lev \neq 0$ **do**
    Extract pixel p from the queue; Update attribute in $levelroot[p]$;
    for all neighbours $q \in P$ of p **do**
      if $isVisited[q] = false$ **then**
        $isVisited[q] \leftarrow true$; $atq \leftarrow 0$;
        if $levelroot[f(q)] =$ not set **then**
          $levelroot[f(q)] \leftarrow q$;
        else
          q.parent $\leftarrow levelroot[f(q)]$;
        end if
        Add q to the queue;
        while $f(q) > lev$ **do**
          $flood(f(q), P, atq)$; $at \leftarrow a + atq$;
        end while
      end if
    end for
  end while

Flooding (cont.)

$m \leftarrow m - 1$;
while $m \geq 0 \wedge levelroot[m] =$ not set **do**
$m \leftarrow m - 1$;
end while
if $m \geq 0$ **then**
$par(node\_qu[levelroot[lev]]) \leftarrow levelroot[m]$;
end if
$node\_qu[levelroot[lev]].attribute = at$;
$levelroot[lev] \leftarrow$ not set;
$a \leftarrow at$;
end procedure

procedure CONNECT(th, i, (x,y))

$area \leftarrow 0$; $areat \leftarrow 0$;

$x \leftarrow$ GetLevelRootOf(x); $y \leftarrow$ GetLevelRootOf(y);

if $f(x) < f(y)$ **then**

Swap(x,y);

end if

while $x \neq y \wedge x \neq \perp$ **do**

$z \leftarrow$GetLevelRootOf($node[x].parent$);

if $f(z) >= f(y) \wedge z \neq \perp$ **then**

$node[x].Area \leftarrow node[x].Area + area$;

$x \leftarrow z$;

else

$areat \leftarrow node[x].Area + area$;

$area \leftarrow node[x].Area$;

$node[x].Area \leftarrow areat$;

$node[x].parent \leftarrow y$;

$x \leftarrow y$; $y \leftarrow z$;

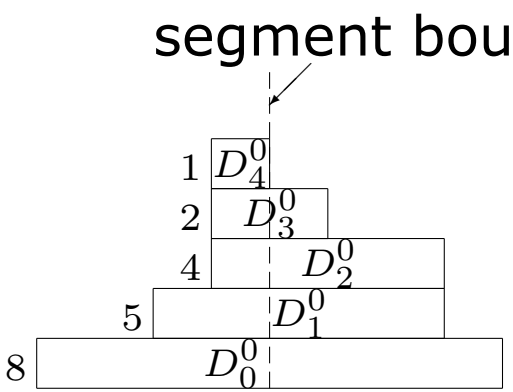
end if

end while

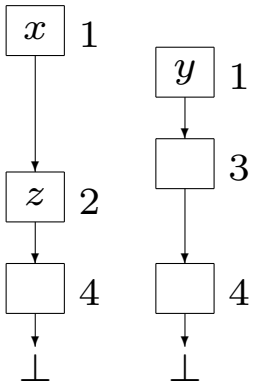
Connect (cont.)


```
    if y = ⊥ then
        while y ≠ ⊥ do
            node[x].Area ← node[x].Area + area;
            x ← GetLevelRootOf(node[x].parent);
        end while
    end if
end procedure
```

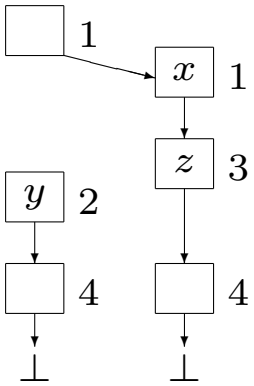
Fusing Two Trees



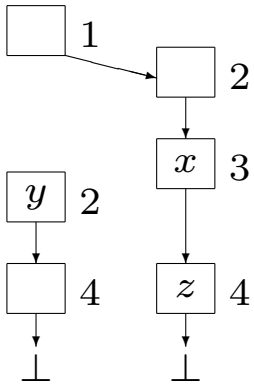
(a) signal



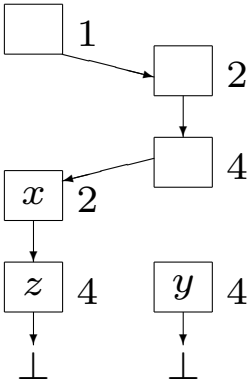
(b) start; `cor`=0



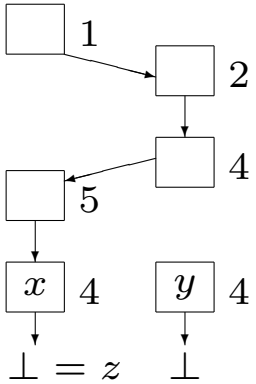
(c) step 1; `cor`=1



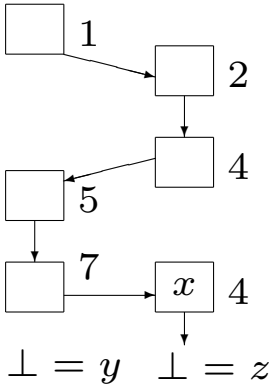
(d) step 2; `cor`=1



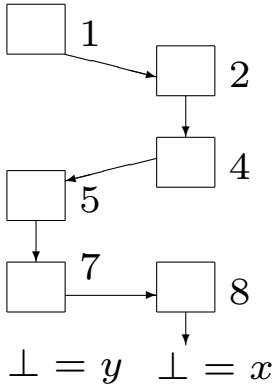
(e) step 3; `cor`= 3



(f) step 4; `cor`= 3



(g) step 5; `cor`=4

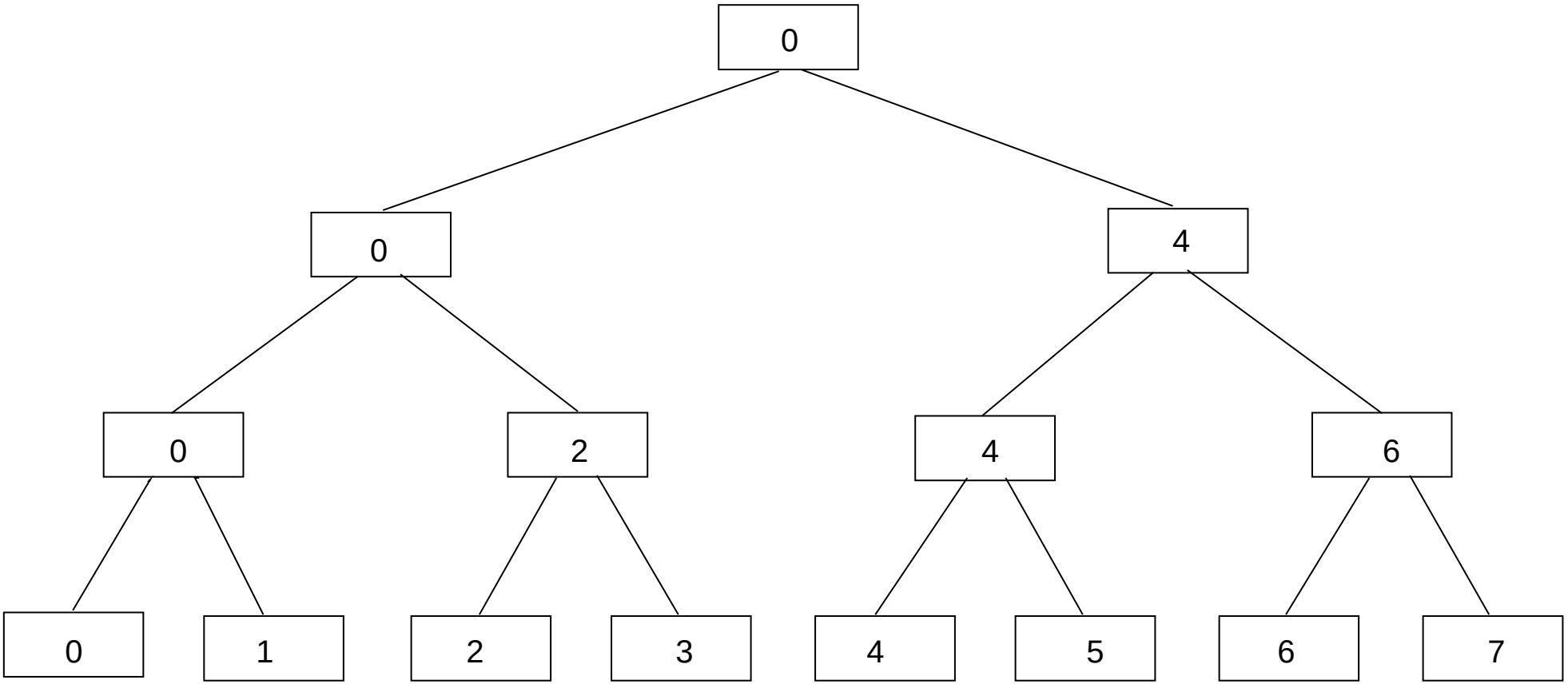


(h) final tree

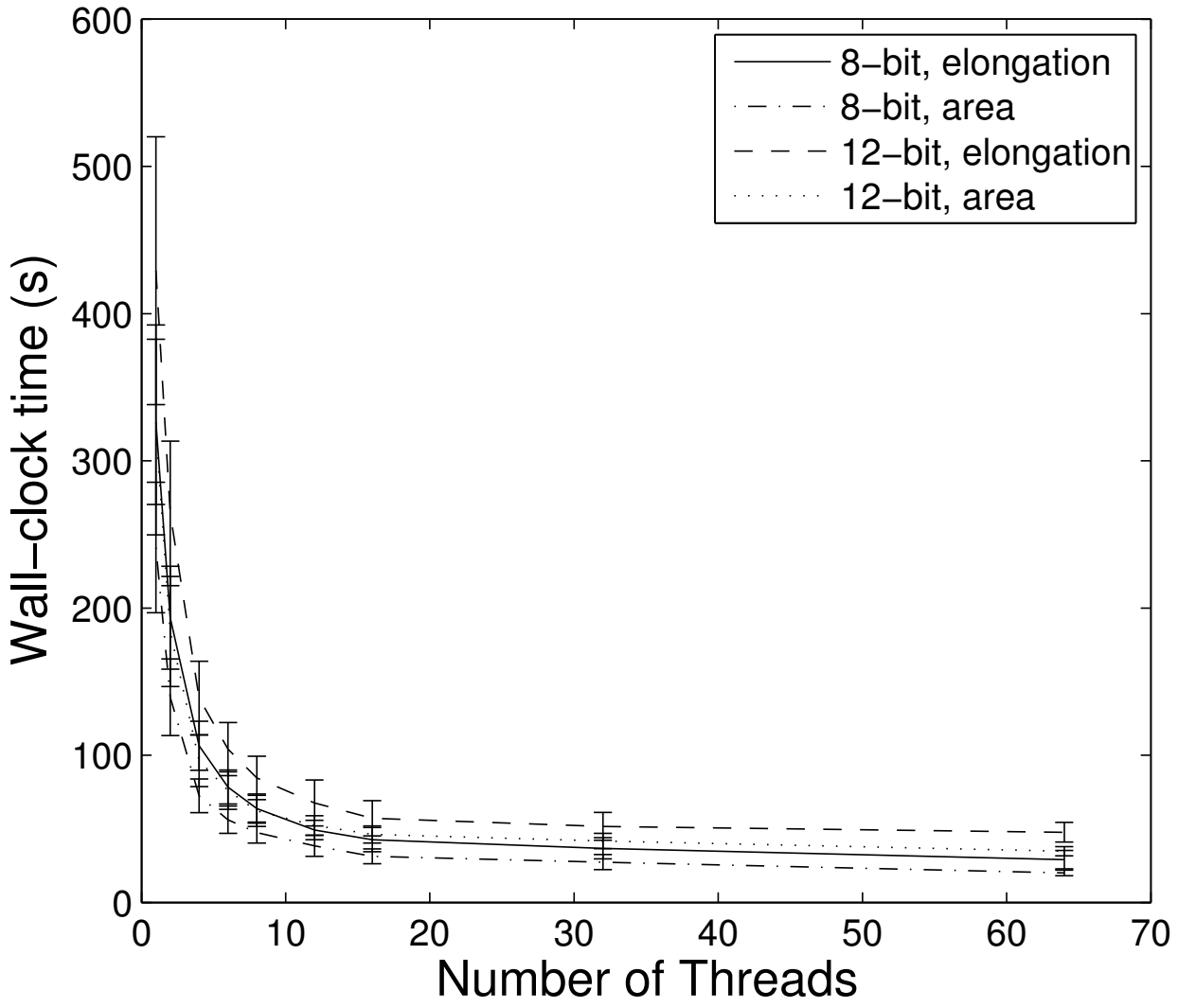
The full algorithm

- Build local Max-Trees using FLOOD
- When finished, signal appropriate neighbour
- Receiving thread merges two max trees by calling CONNECT for each pair of pixels spanning the border
- When pairs of section have merged, repeat merging heirarchically

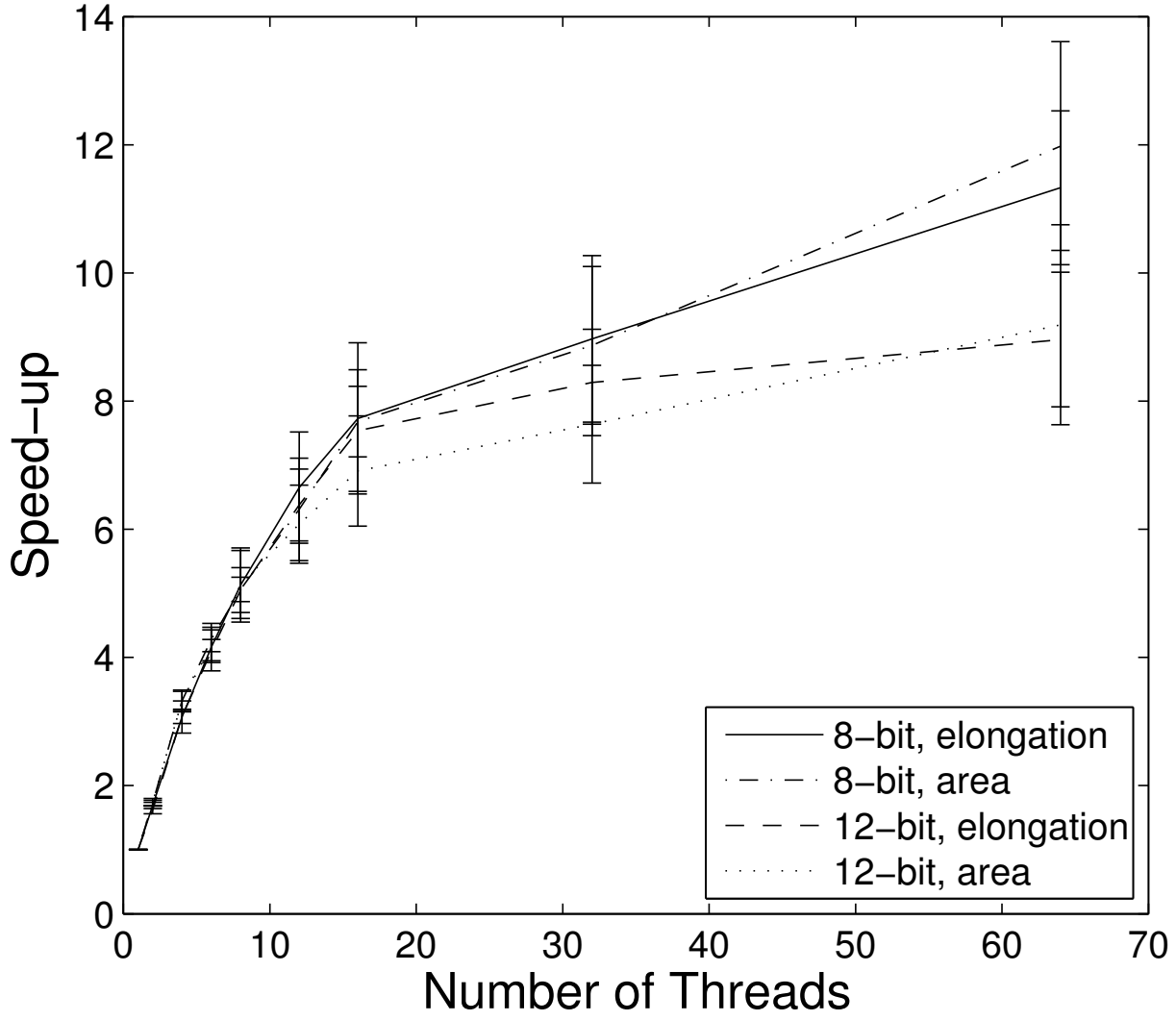
Another Tree



Timings (Onyx 16-core)



Speed-up (Onyx 16-core)



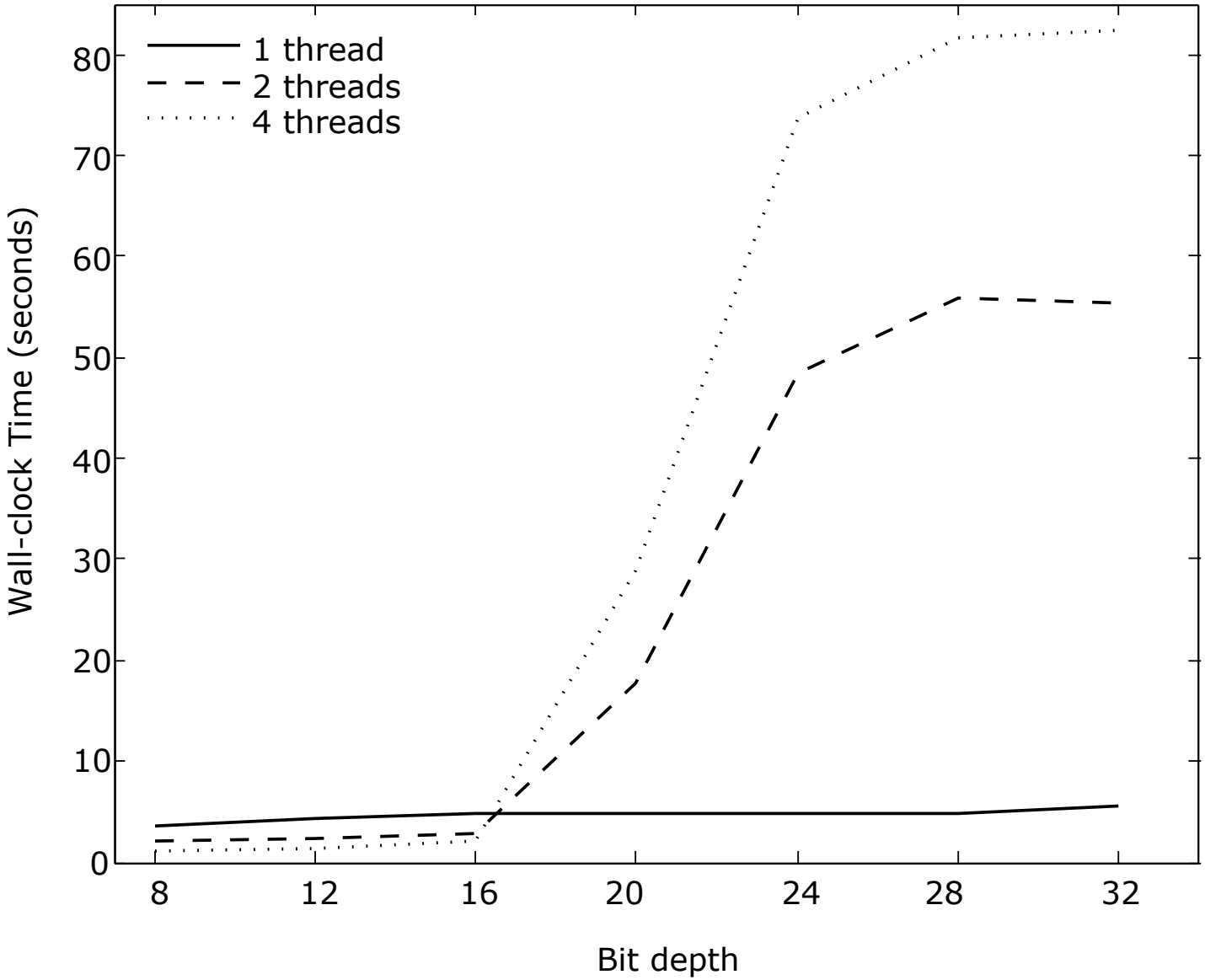
Complexity of Merges

- Each merge process has complexity

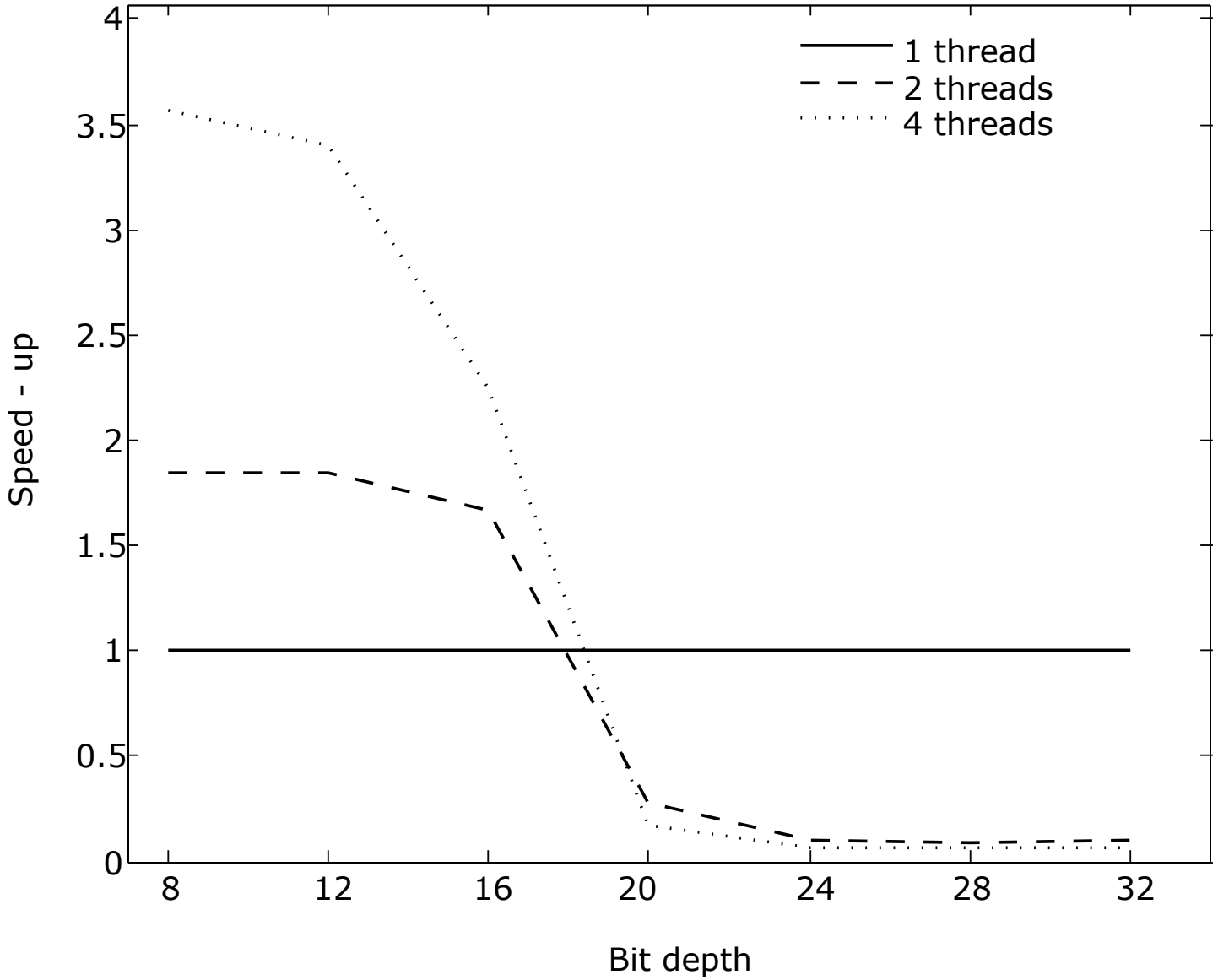
$$O(A_{interface} \min(G, N_{pixels}/N_{blocks})) \tag{7}$$

- N_{blocks} ranges from 2 to P, with P the number of processors
- $A_{interface}$ is the area of the interface:
  - $O(\sqrt{N_{pixels}})$ in 2D
  - $O(N_{pixels}^{2/3})$ in 3D.
- The number of merge processes is $\log P$

The Impact of Bit-Depth



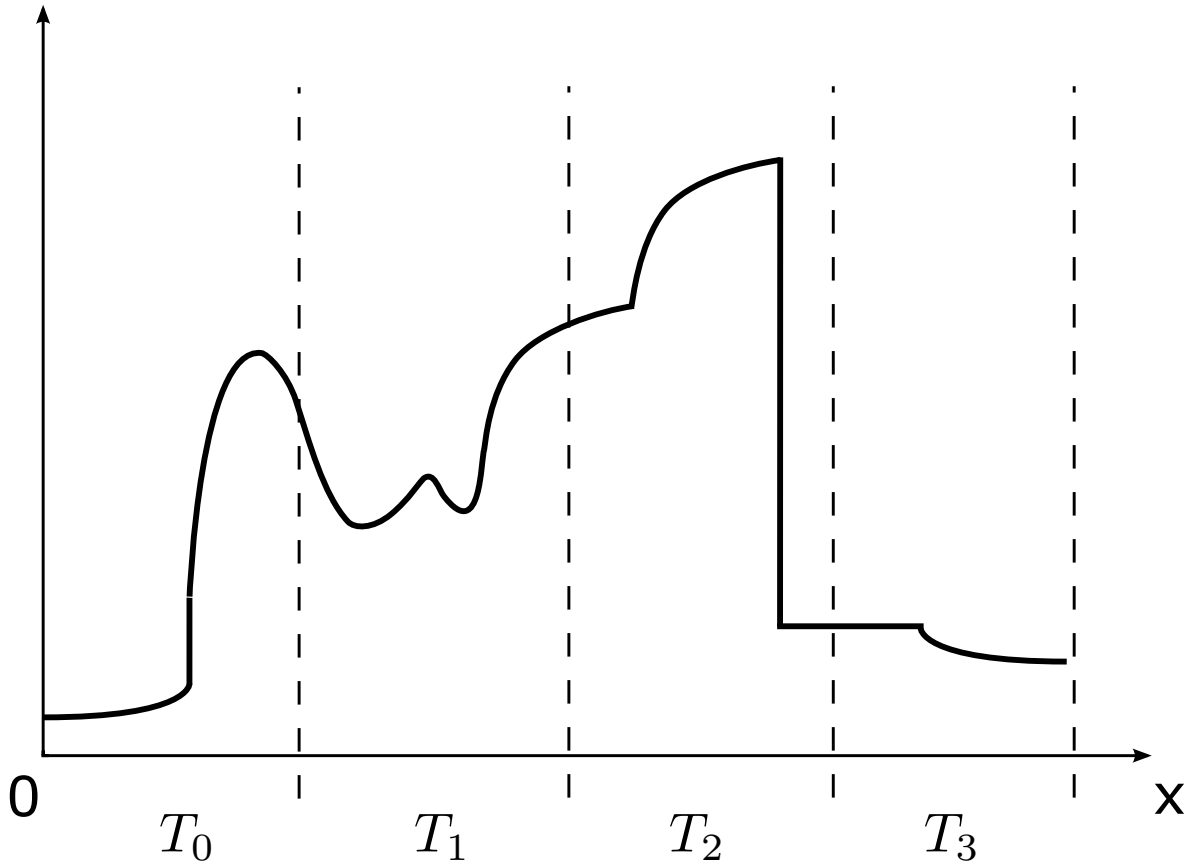
The Impact of Bit-Depth



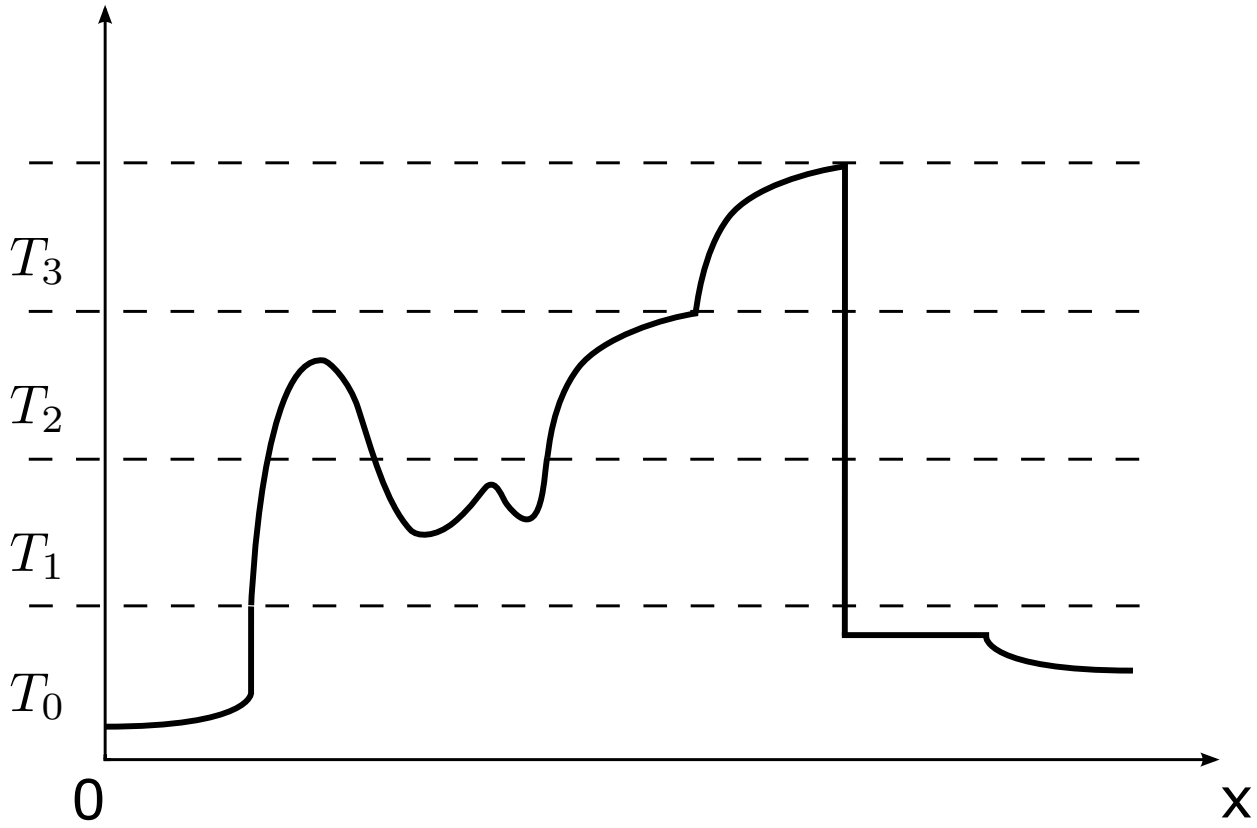
Alternative Approach: Union-Find?

- Two options:
  1. Start out with one thread per maximum
     - Flood using priority queue.
     - Flooding waits when next pixel has lower grey level than current
       - Problem: mutual exclusion
       - Semaphore per pixel may be needed
  2. Classic spatial division per grey level
     - Region growing waits when new grey level reached
       - Problem: One barrier per grey level needed.
       - Worst case: one barrier per pixel
- Conclusion: does not look promising

The Old Cut



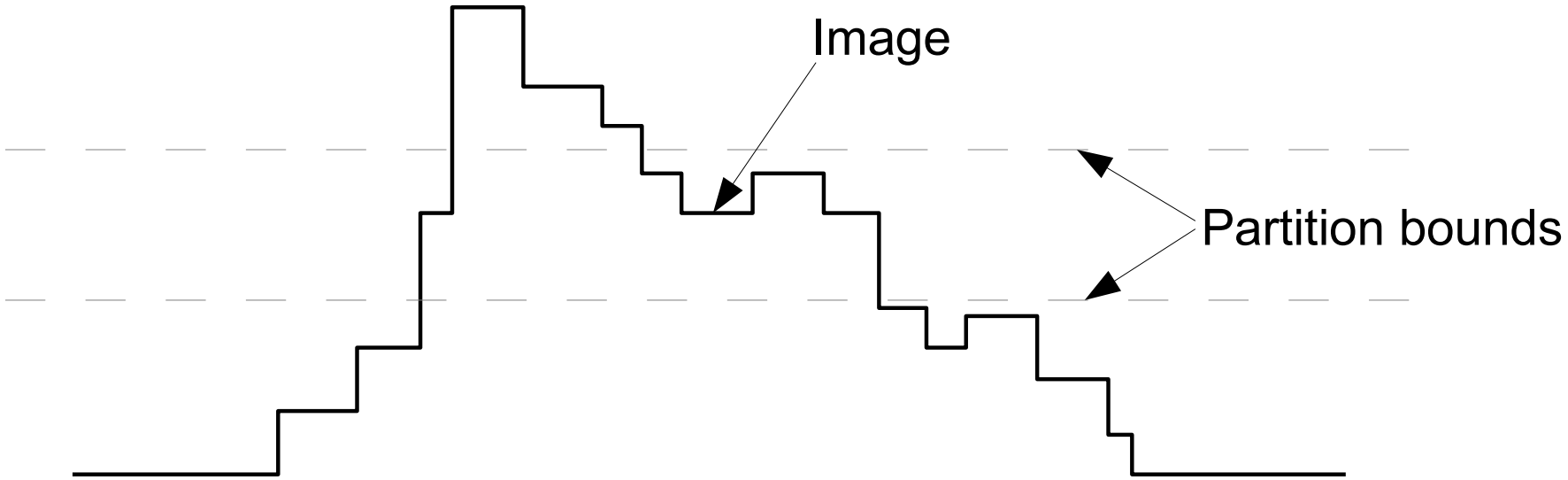
A Different Cut



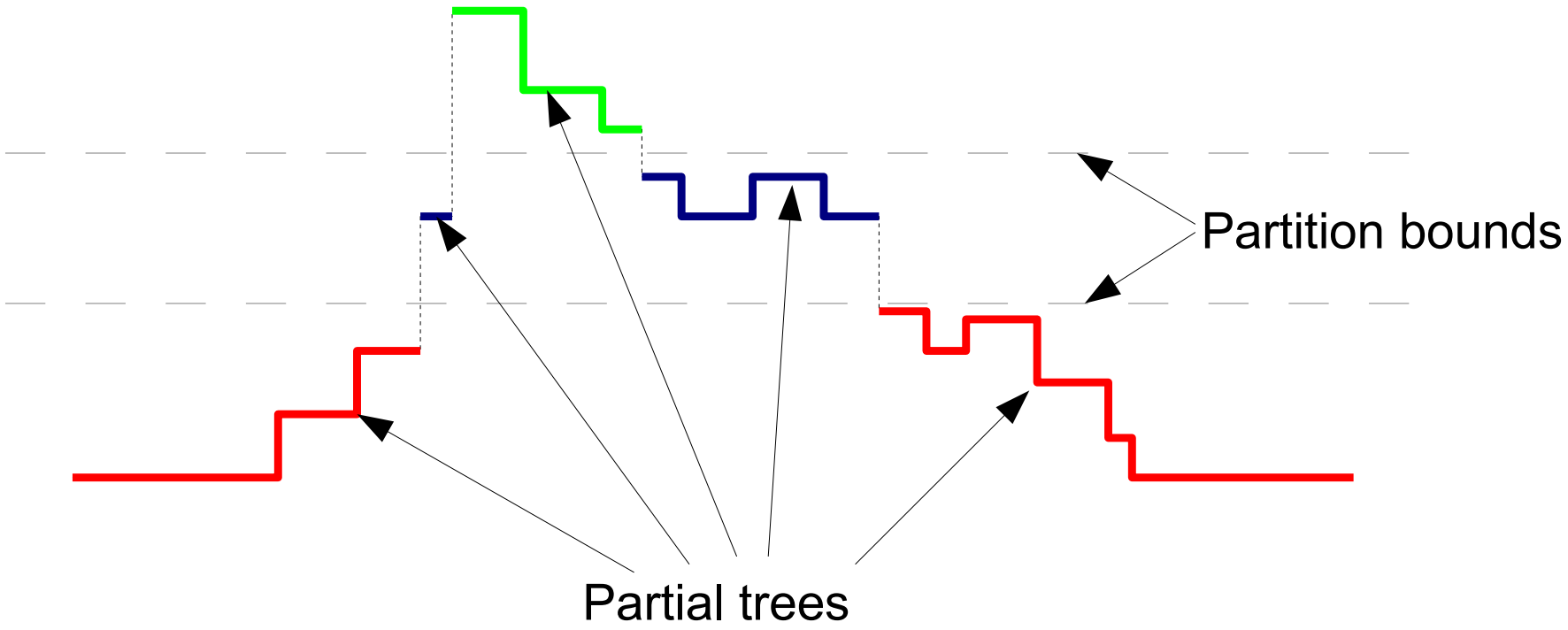
Partitions by grey level

- Divide image in data chunks *by grey level*
- Compute forest of Max-Trees per grey level
- Merge these into full tree
  - Pro: Most single merges are $O(1)$
  - Con: some "zipping" still required
- These costly merges arise because the topology of the upper neighbours unknown
- Does not look like whole answer

The "Zipping" Problem



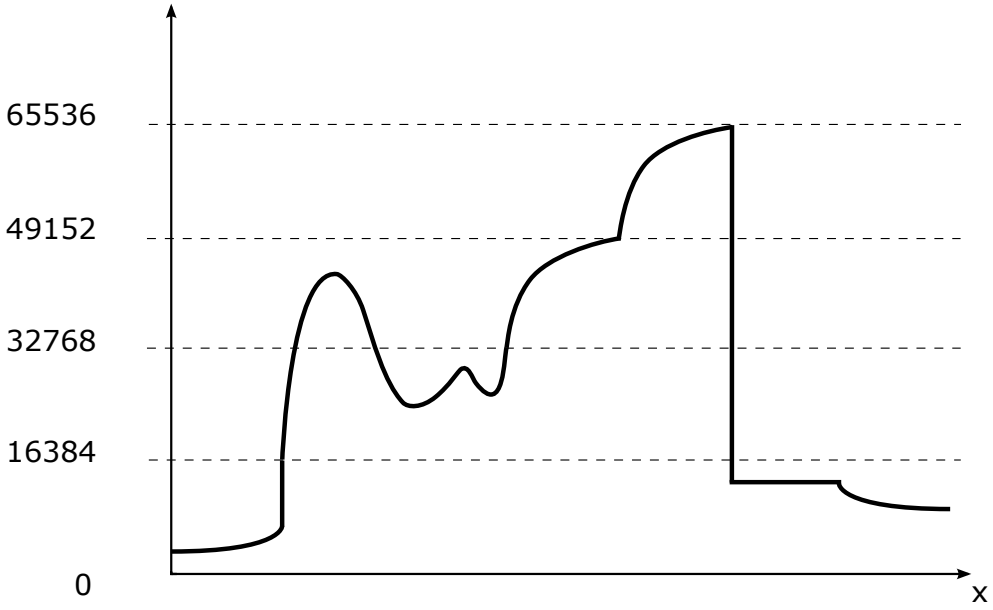
The "Zipping" Problem



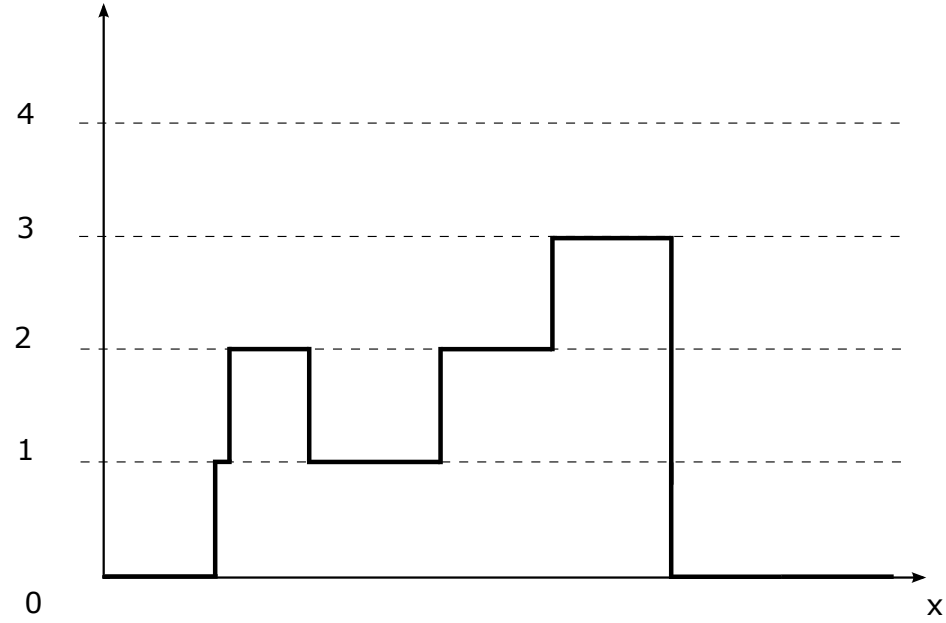
A Diplomatic Algorithm

- Sort data according to grey level (Parallel Radix Sort)
- Compute image g quantised to P grey levels with uniform histogram (in parallel)
- Build "pilot" Max-Tree of using parallel low-dynamic-range algorithm
- Use modification of Berger et al per quantisation bin to refine this pilot tree
- filter in parallel as usual

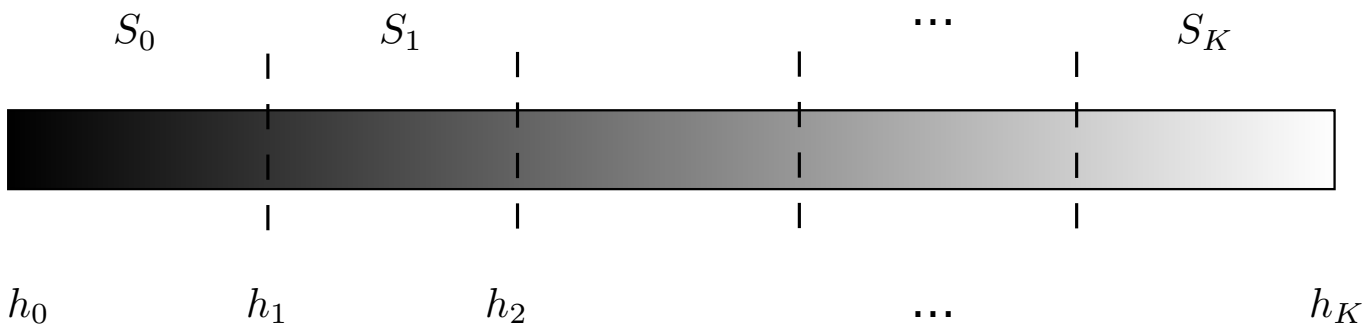
Quantisation



original



quantised



Sorted array

- At the moment our quantisation is uniform
- It is not clear that this gives optimal load balancing
- Lower quantisation bins are expected to have larger numbers of nested maxima
- This means that the refinement phase of lower bins has to consult the quantised tree more often
- The depth of the branch above them also plays a role
- Inspecting a nested maximum is $O(P - i)$ worst case, with P the number of processors, and i the thread id (quantisation bin).

procedure FLOOD(Level lev, Partition P, Attribute a)
  $at \leftarrow a$;
  while $lev \neq 0$ **do**
    Extract pixel p from the queue; Update attribute in $levelroot[p]$;
    for all neighbours $q \in P$ of p **do**
      if $isVisited[q] = false$ **then**
        $isVisited[q] \leftarrow true$; $atq \leftarrow 0$;
        if $levelroot[$**g**$(q)] =$ not set **then**
          $levelroot[$**g**$(q)] \leftarrow q$;
        else
          KeepLowestLevelRoot();
        end if
        Add q to the queue;
        while **g**$(q) > lev$ **do**
          $flood($**g**$(q), P, atq)$; $at \leftarrow a + atq$;
        end while
      end if
    end for
  end while

Flooding (cont.)

$m \leftarrow m - 1$;
while $m \geq 0 \wedge levelroot[m] =$ not set **do**
$m \leftarrow m - 1$;
end while
if $m \geq 0$ **then**
$par(node\_qu[levelroot[lev]]) \leftarrow levelroot[m]$;
end if
$node\_qu[levelroot[lev]].attribute = at$;
$levelroot[lev] \leftarrow$ not set;
$a \leftarrow at$;
end procedure

procedure KEEPLOWESTLEVELROOT( )
  if $q \prec levelroot[g(q)]$ **then** ▷ exchange level roots
    $par(node\_qu[levelroot[g(q)]]) \leftarrow q$;
    $levelroot[g(q)] \leftarrow q$;
  end if
  $par(node\_qu[q]) \leftarrow levelroot[g(q)]$;
end procedure

With

$$q \prec levelroot[g(q)] \equiv (f(q) < f(levelroot[g(q)])) \vee ((f(q) = f(levelroot[g(q)])) \wedge (q < levelroot[g(q)])) \quad (8)$$

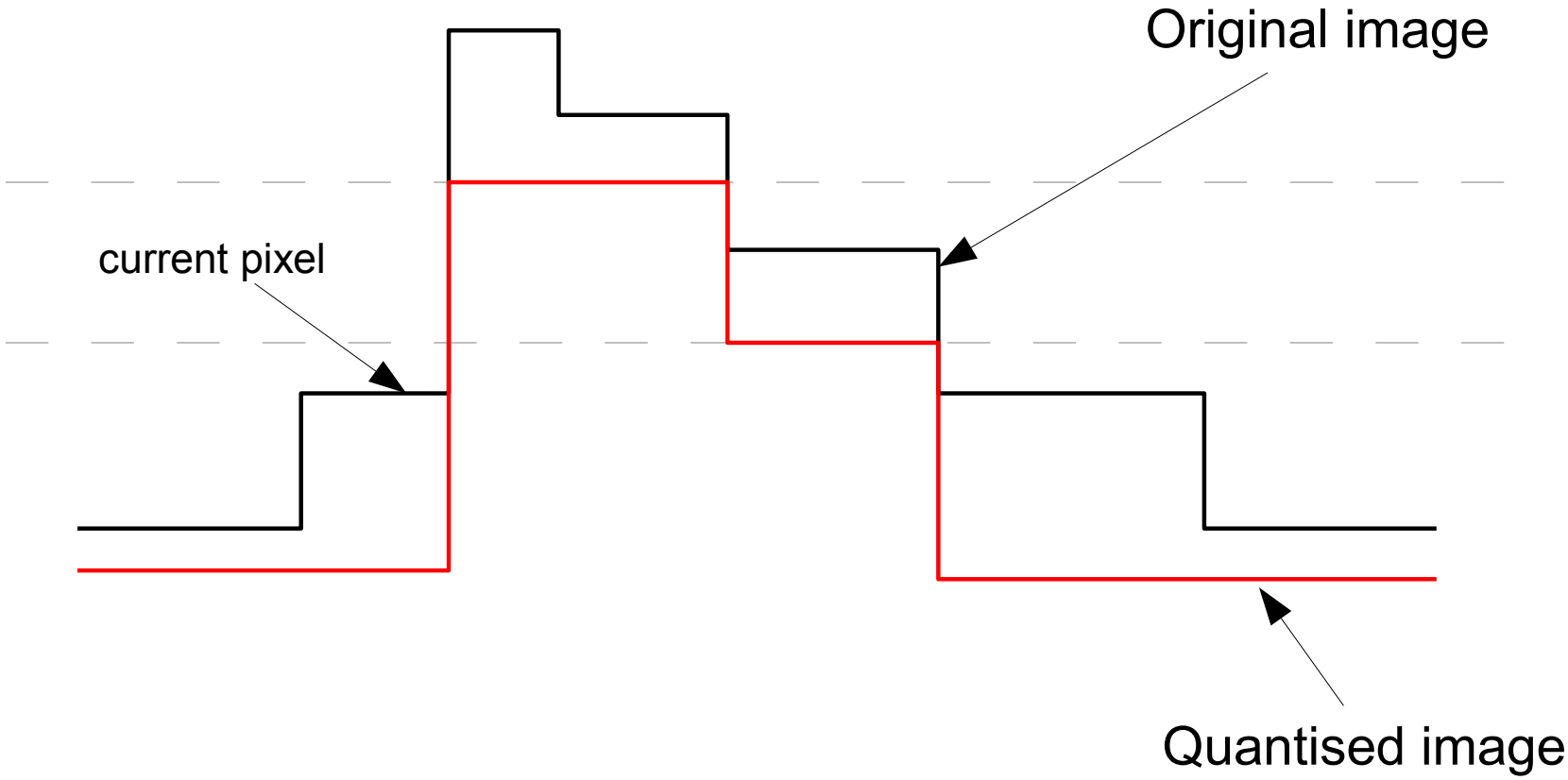
with $g(.)$ the quantised image and $f(.)$ the original image value.

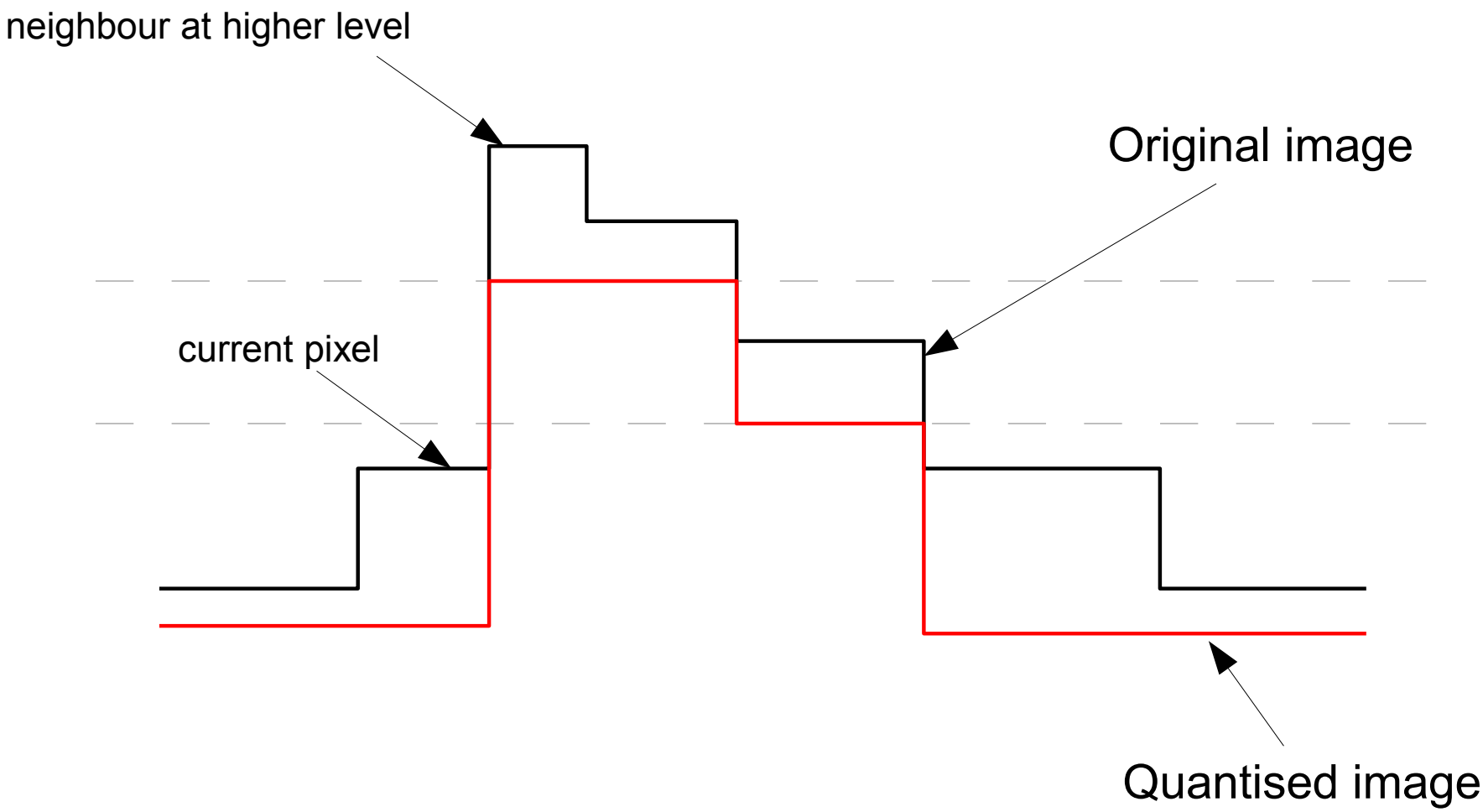
procedure REFINEMENT(i, S_i, H_i)
  $lwb \leftarrow min(S_i)$; $upb \leftarrow max(S_i)$;
  for $j = upb$ to lwb **do**
    $p \leftarrow SORTED[j]$;
    $zpar[p] = p$;
    for all neighbours q of p **do**
      if $g(q) > i$ **then**
        $anc \leftarrow$ DESCENDROOTS(q, i);
        if par($node\_ref[desc]$) = not set **then**
          $par(node\_ref[desc]) \leftarrow p$;
          merge attribute of $desc$ with p;
        else
          $z \leftarrow$ FINDROOT($par(node\_ref[desc])$);
          if $z \neq p$ **then**
            $par(node\_ref[z]) \leftarrow p$;
            $zpar[z] \leftarrow p$;
            merge attribute of z with p;
          end if
        end if

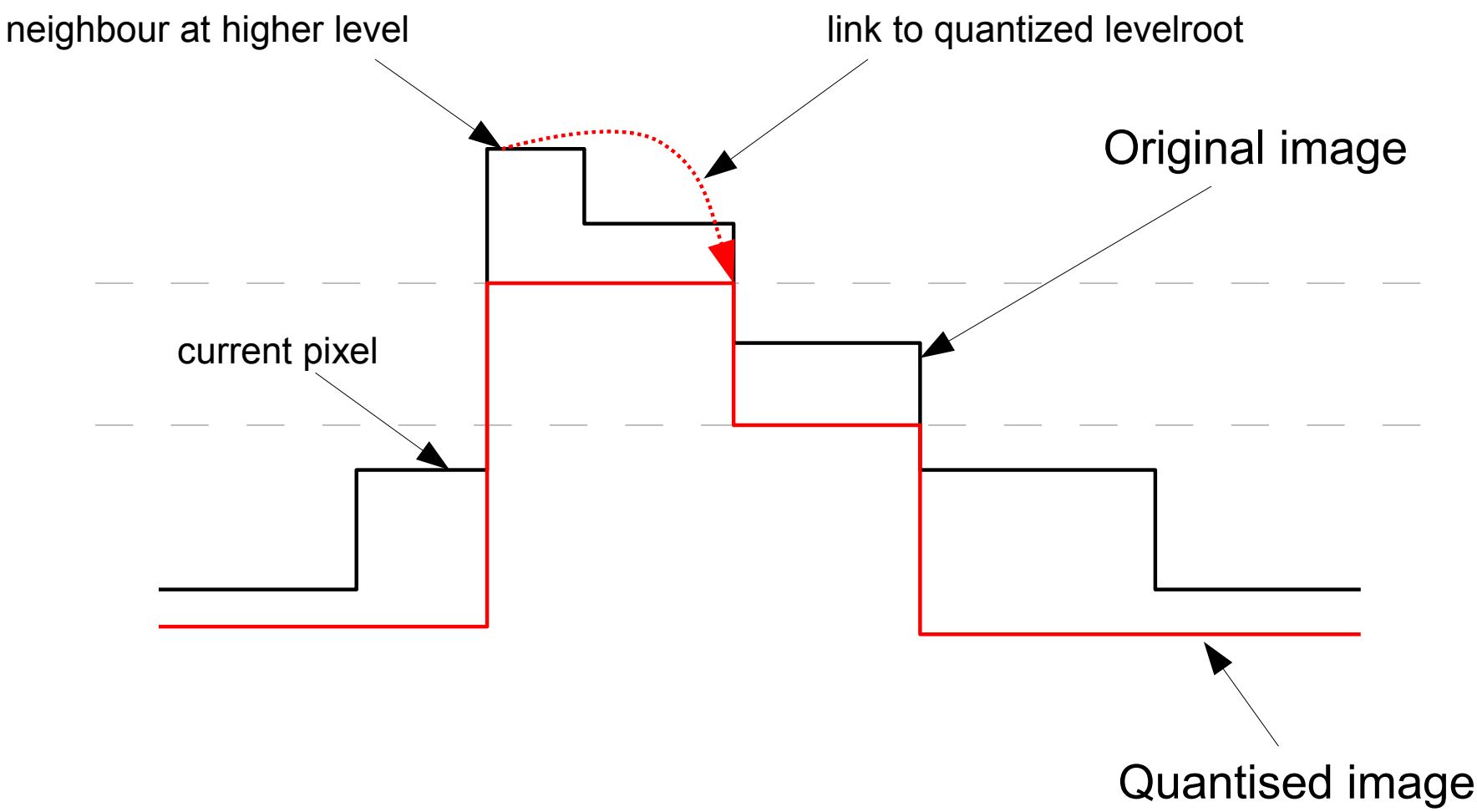
```
            else if g(q) = i then
                if zpar[q] ≠ −1 then
                    r ← FindRoot(q);
                    if r ≠ p then
                        par(node_ref[r]) ← p;
                        zpar[r] ← p;
                        merge attribute of r with p;
                    end if
                end if
            end if
        end for
    end for
end procedure
```

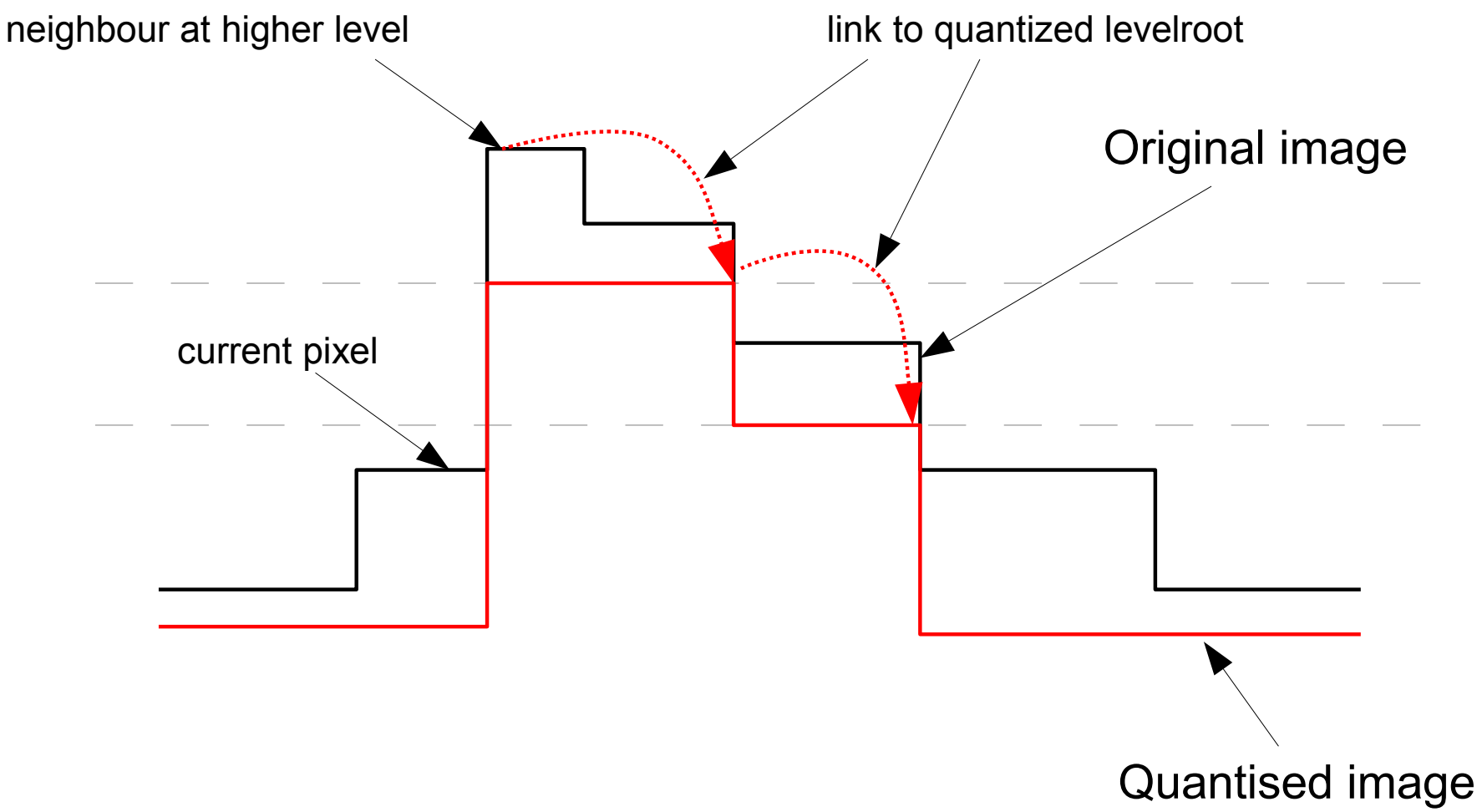
procedure DESCENDROOTS(Pixel q, int i)
    $c \leftarrow q$;
    while $g(par(node\_qu[c])) > i$ **do**
        $c \leftarrow par(node\_qu[c])$;
    end while
    return c;
end procedure

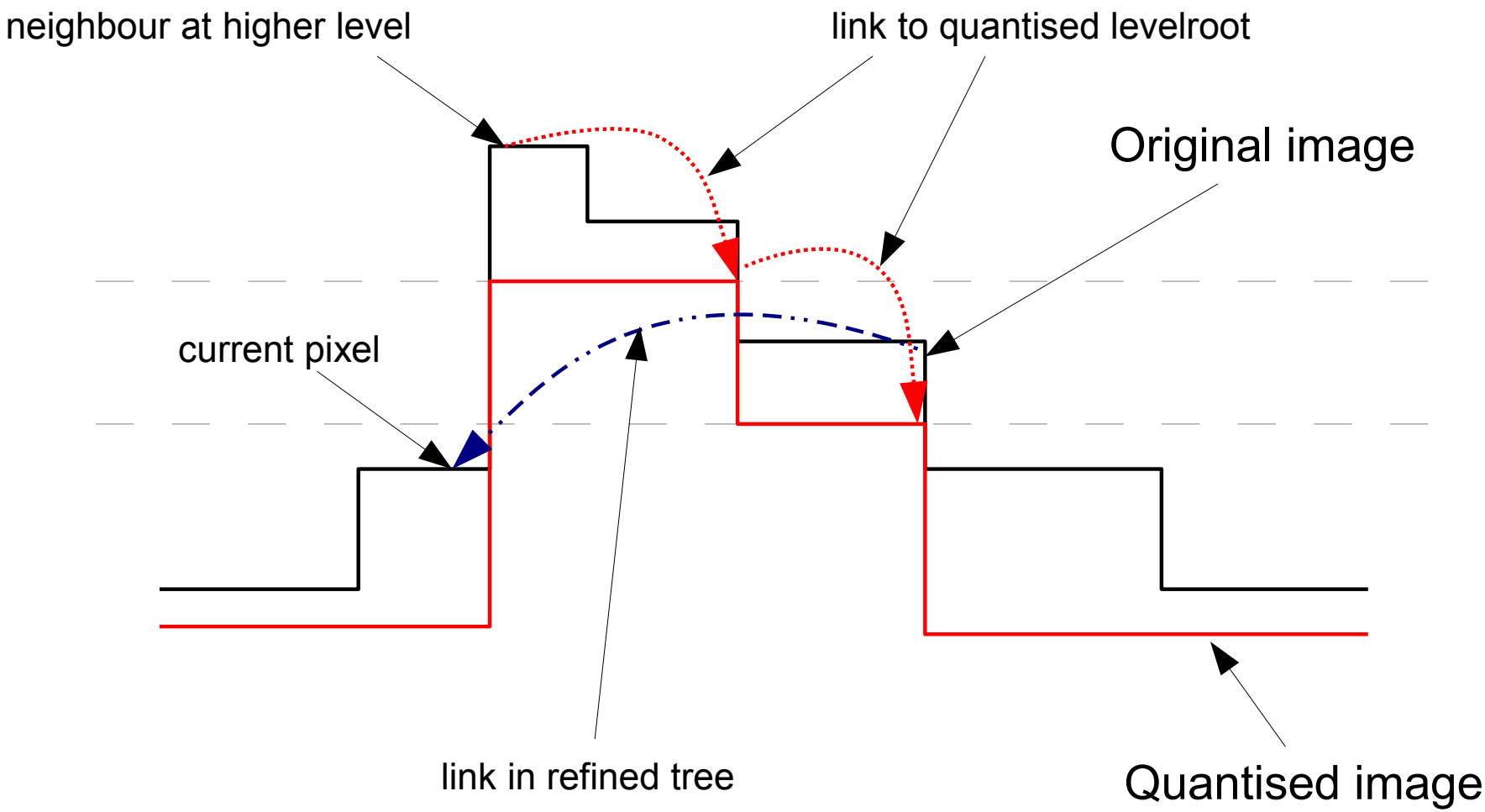
With i the thread number (= quantised grey level)


Original image
current pixel
Quantised image


neighbour at higher level
Original image
current pixel
Quantised image


neighbour at higher level
link to quantized levelroot
Original image
current pixel
Quantised image

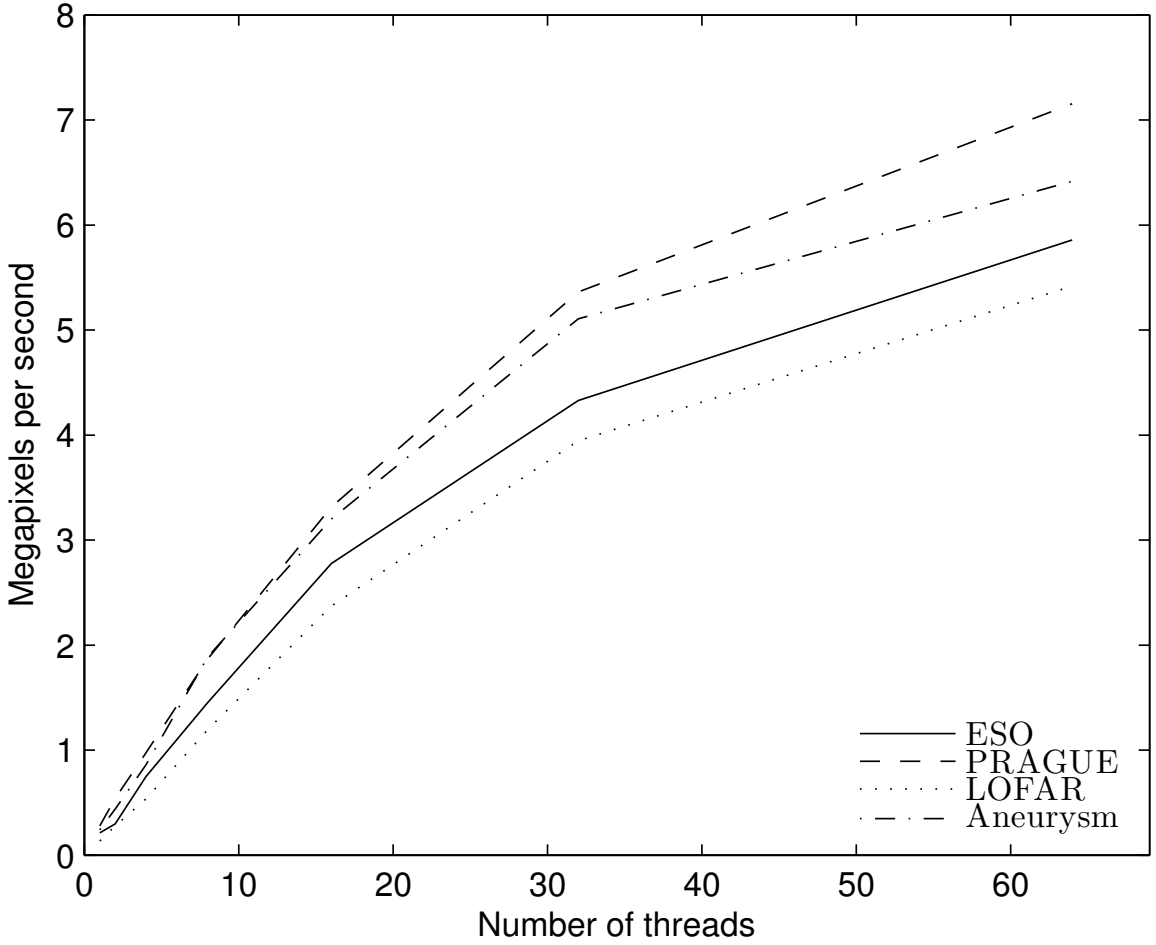

neighbour at higher level
link to quantized levelroot
Original image
current pixel
Quantised image


neighbour at higher level
link to quantised levelroot
Original image
current pixel
link in refined tree
Quantised image

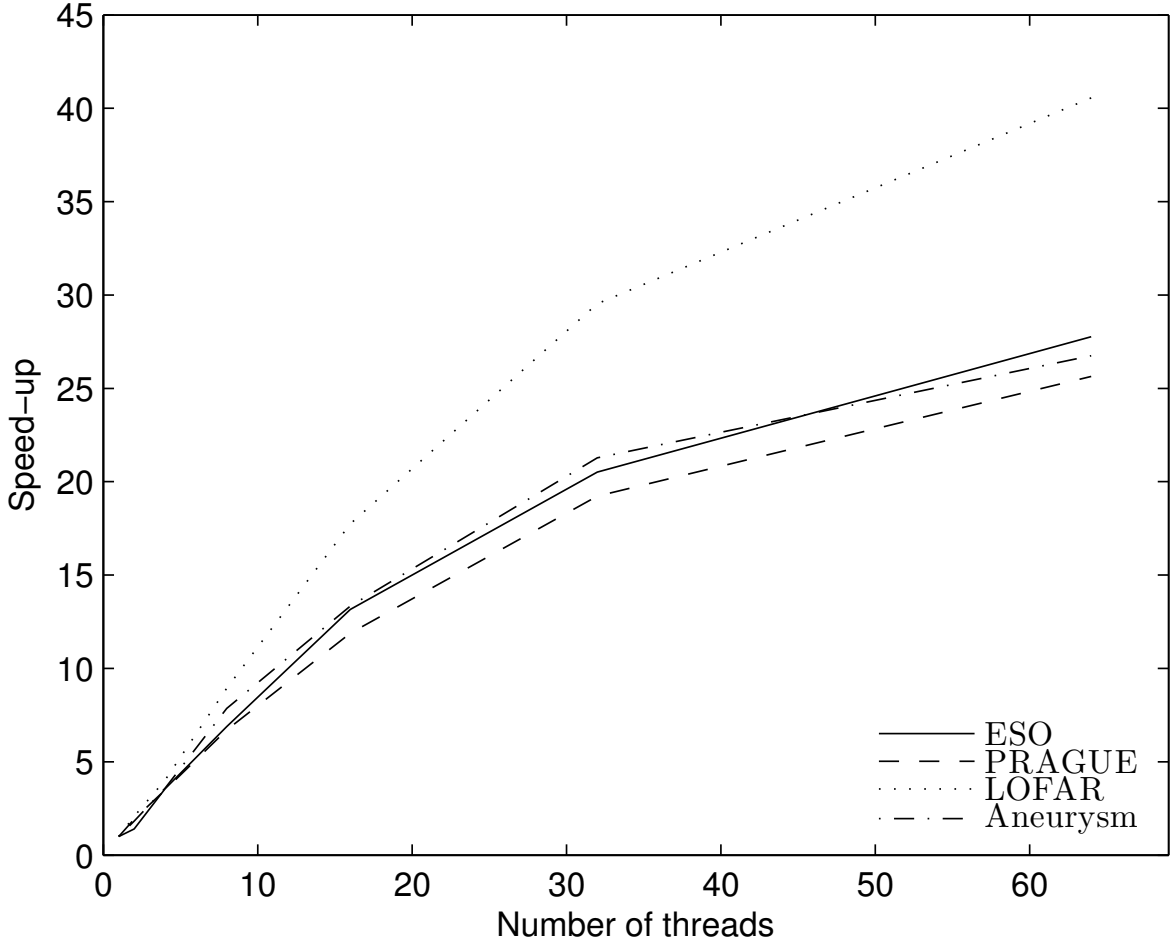
Experiment

- All code was implemented in C using POSIX-threads
- Parallel radix sort was used because it is simple and stable
- Same data structure as in (Wilkinson et al 2008) was used.
- This means that the filtering stage of the algorithm could be left unchanged
- Performance test were carried out on several images and volumes on a Dell R815 compute server with 4 16 core Opteron processors and 512 GB of RAM.

Results: Timing



Results: Speed-up



Conclusions

- Parallel computation of CC-labelling and connected filters is possible
- CC-Labelling is fairly straightforward with Union-Find
- Connected filtering requires extra thought
- Shared-memory algorithms for low and extreme dynamic range (XDR) images Max-Trees have been developed
- The former algorithm has a speed-up of up to $50\times$ on 64 cores
- The XDR algorithm $25 - 40\times$ speed-up on 64 cores
- Roughly $14 - 20\times$ faster than fastest sequential algorithm (Berger et al. 2007)

Next Time

- Multi-scale analysis using Max-Trees
- Moving to distributed memory

?