Connected Component Labelling

Alexandra Vegelien 2552901 Niels Wouda s2533561

Parallel Algorithms

13th January 2020

Contents

1 Introduction

Connected components labelling (CCL) is not only an elegant computer science problem, but has an important practical use. The goal is to assign a unique label to each component of an image, a fundamental problem in *i.a.* image processing (Zhao et al., 2010, p. 1). Many algorithms exist for labelling dense matrices. As technological advancements result in large 3D—and nowadays even 4D—images, memory concerns play an increasing role. Exploiting sparsity in binary images reduces memory consumption significantly, and parallel algorithms can further reduce both memory and computational needs for each individual machine (Ohira, 2018, p. 1). Reducing computation time might allow the application of CCL in real time (Nina Paravecino and Kaeli, 2014, p. 1).

We limit ourselves to 3D matrices in this study, although the methods discussed generalise to arbitrary dimensions. One of the many practical applications of the proposed algorithm is counting spots in a 3D images obtained by fluorescence microscopy (Ram et al., 2012, p. 1); identifying the trajectory of moving vehicles over time (Molinier et al., 2005, p. 1); or the real time identification of objects in airport luggage (Nina Paravecino and Kaeli, 2014, p. 1). Many medical applications exist also, in 3D (Ohira, 2018, p. 1) and even 4D CT scans (Kwong et al., 2015, p. 1).

The paper is structured as follows. Section 2 gives an in-depth discussion of the problem. Section 3 discusses the sequential and parallel algorithms, and introduces many of the (computer science) concepts used to arrive at an efficient parallel algorithm for labelling sparse, 3D matrices. In Section 4, we present running times compare these to theoretical cost estimates estimates. Finally, in Section 5 we present several concluding remarks.

2 Problem formulation

The problem is 'easy' enough in dense matrices, as there a labelling strategy can be performed in time linear in the number of elements (which is $O(k^3)$ for a cube with sides of length k), as this requires a single in-order traversal looking back at the preceding neighbouring voxels to determine a label. If the process can be done in linear time for dense matrices, why should we even bother with a parallel algorithm?

Our main motivation is memory consumption, in particular for matrices where $k \geqslant 1000$. Storing such a matrix in memory requires on the order of $O(10^9)$ or more bytes, which might well be prohibitive even on a computer cluster node. The memory requirements drop significantly if we can spread the matrix across various processors, as a parallel algorithm permits. For binary matrices, we will exploit sparsity to further decrease memory consumption, even when this results in non-linear time computation. This is the price to be paid for working with sparse matrices.

We assume the sparse matrix is given in COOrdinate format, as triplets of (x, y, z) coordinates (Sanderson and Curtin, 2019). A COO-matrix usually encodes a value for such a coordinate as well, but in our case this is not needed: we assume each coordinate encodes a 'black'/1 voxel, and those not given in the data (we have a sparse matrix, after all!) represent 'white'/0 voxels. Our task is to label all given voxels with a consistent labelling, such that voxels belonging to the same component are given the same, component-wise unique label. We will allow these labels to be discontinuous integers: their ordering and values do not matter, only their uniqueness is relevant to the problem at hand.

We furthermore assume the structure of the given matrix is such that the coordinates are sorted

in a particular order. The assumption is without loss of generality, as the matrix can always be pre-sorted such that the assumption holds. We refer to this as the *iteration order*, and it is defined as follows: for fixed x and y, z is sorted in non-decreasing order; for fixed x, the y's (and consequently the z's) are sorted in non-decreasing order, and finally the x's are themselves sorted in non-decreasing order—in short, the elements adhere to a nested sorting. A clarifying example is given in Table 1 and Figure 1a.

Table 1: Example COO-matrix with eight coordinates given. The variable i presents a linear index into the data. Observe that the data adhere to the iteration order as we iterate from left to right.

i	0	1	2	3	4	5	6	7	$\cdots$
x	0	0	0	1	2	2	2	2	$\cdots$
y	0	0	0	2	0	2	2	2	$\cdots$
z	0	1	2	2	0	0	1	2	$\cdots$

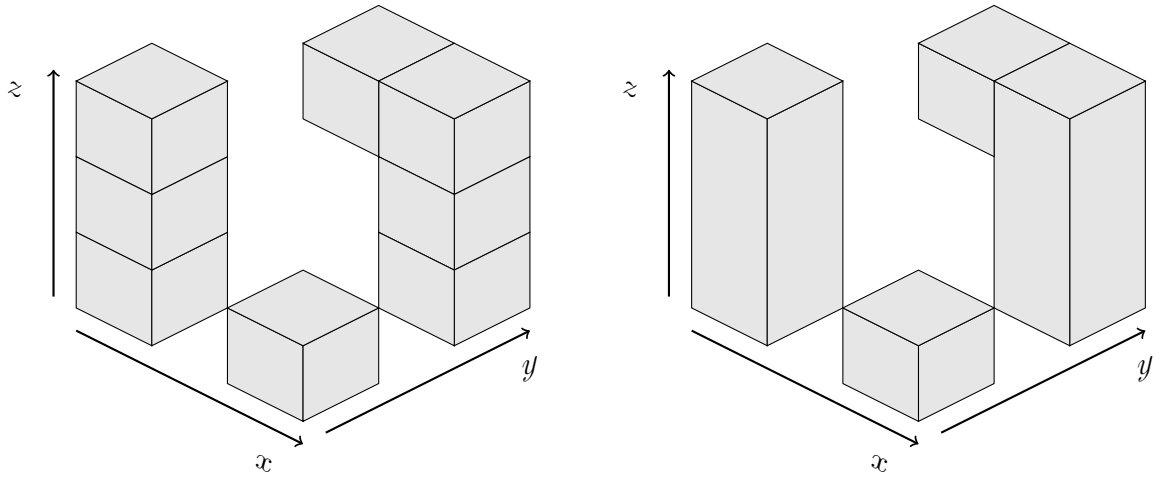
Since we aim to find and label *connected* components, we assume the input matrix is at least somewhat connected. As such, purely random matrices, or those with a chequered pattern (where the number of components equals the number of non-zeroes) are not considered 'appropriate' for the algorithm at hand. In general, we assume the number of components to be much, much smaller than the number of non-zeroes. The examples given in the introduction (cells in fluorescence microscopy and trajectories of cars over time) all satisfy this assumption, as do the datasets we consider here. The assumption motivates the design of the algorithm, but it necessarily cannot be enforced at runtime: even when the number of components is large relative to the number of non-zeroes, the algorithm will still terminate with a correct labelling—but performance might suffer.

3 Solution approach

In this section we develop a parallel method for labelling connected components in sparse, binary 3D matrices of arbitrary size. In Section 3.1 we implement a sequential algorithm for the problem. A sequential algorithm offers valuable insight into the problem at hand, and illuminates opportunities for parallelisation. The sequential algorithm is even re-used in the parallel algorithm we develop in Section 3.2.

3.1 Sequential algorithm

Assume we are given a sparse, binary matrix in the format explained in Section 2. As an intermediate step we first construct segments to label. A segment is a compressed representation of voxels that share a face in the z-dimension (*cf.* Zhao et al. (2010) for a similar approach to 2D binary images). It consists of the coordinate fields x, y, z_{first}, and z_{last}, and additional data fields *parent*, *rank*, and *label* (these will be discussed shortly). Neighbours in the z-dimension also neighbour in our COO-matrix, and thus segments can be constructed in linear time while preserving the iteration



(a) **(b)**

Figure 1: Figure 1a shows the voxels of Table 1. When compressed into segments, we obtain Figure 1b.

order. There is no guarantee this will significantly reduce the computational load, but in practice it is likely there are far fewer segments than voxels. After labelling each segment, we reverse the compression and obtain labelled voxels. In Figure 1, the example of Table 1 reduces from eight voxels to four segments.

Having obtained these segments, we are now ready to determine the individual components. For this we first need to find neighbouring segments we have visited previously (in iteration order). Finding such neighbours superficially appears to be a trivial task: they are either at $(x-1, y)$ if $x > 0$ for appropriate values of z, or else $(x, y-1)$ if $y > 0$. In a dense matrix all this requires is a constant-time index look-up. For the sparse matrix we have here, however, we do not know the indices of such segments, should they exist: we need to find them instead. Due to the sorted nature of our data, this can be done via an efficient binary search.

Now that we know how to find neighbouring segments, we need a way to connect them into components (union), and a way to find out to which component a segment belongs (find). An efficient data structure supporting both these operations is often called a *union-find* or *disjoint-set* data structure (Cormen et al., 2009, ch. 21). Our implementation largely follows the treatment given there. Union-find data structures form trees where each segment points towards its parent, eventually reaching the root segment which points to itself. The root segment uniquely determines the component: different components have different root segments. Initially, each segment is assigned its own component.

A naïve implementation might use a linked list to represent a component, in which each segment points to a preceding neighbour. Although simple in its implementation, such a list makes determining the root segment a linear time operation, which in turns makes the find and union operators rather inefficient. Instead, it is best to ensure a modicum of balance for our component trees, using a procedure termed *path compression*. There are many flavours to this: Cormen et al. (2009, ch. 21) give a two-pass method that recursively traverses up the component tree (first pass), and then unwinds the recursion to reassign each parent to the root segment (second pass). This

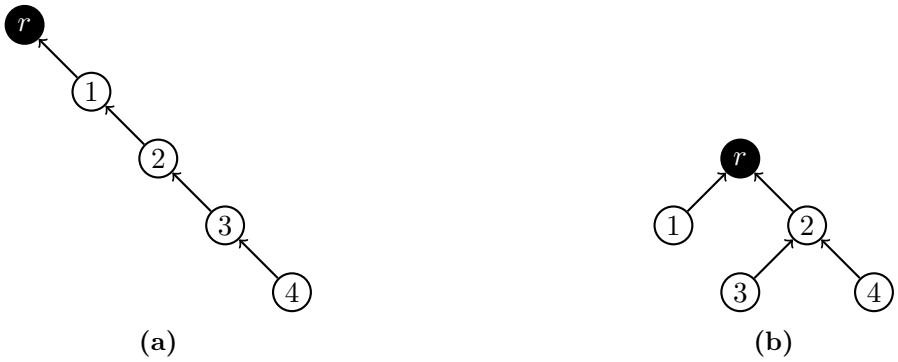


Figure 2: In Figure 2a, we have a linked list representing a component of five segments. As we traverse up the list from segment 4 to reach the root r, every other segment is made to point to its grandparent instead (segments 4 and 2, as in Figure 2b). This halves the length of subsequent traversals.

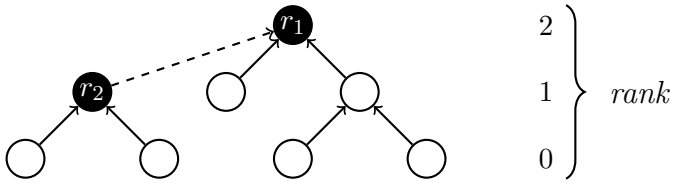


Figure 3: An example illustrating the union-by-rank heuristic. As the root segment r_1 has greater rank than r_2, we point r_2 to r_1. No ranks need to be updated as the heights of r_1 and r_2 remain unchanged.

ensures perfect balance, as each intermediate parent directly points to the root segment after such a traversal. It is, however, inefficient to use recursion in this setting, and non-recursive one-pass methods exists. We rely on *path halving*, which is one such method due to Tarjan and Van Leeuwen (1984). It works as follows: every time the root segment needs to be determined, it assigns every other segment on the path leading to the root to its grandparent instead. This effectively halves the path each time the method is invoked—hence its name. An example highlighting the improved tree balance path halving brings is given in Figure 2.

Having improved the find operation using path compression, we now turn to another heuristic for the union operator. Merging two components requires one root segment to point to the other. But how should the algorithm decide which root segment to keep, and which to point to the other? We would like to point the shorter tree to the larger, to ensure the merged tree's height does not increase needlessly. To keep track of a segment's height in the component tree, we use the *union-by-rank* heuristic. Each segment is initially assigned rank zero, and the root segment's rank is updated every time two components of equal rank are merged. If the ranks are unequal, the shorter tree is pointed to the larger as in Figure 3, and no rank needs to be updated. It follows that the rank provides an upper bound on the height of a component tree.

The use of both the path compression and union-by-rank heuristics results in a union-find data structure that achieves optimal asymptotic complexity (first shown in Tarjan (1975)). In a single pass over all segments, we find preceding neighbours, merge them with the currently considered segment, and as such build our set of components. All that remains is to determine a unique component label, and to reverse the segment compression to obtain labelled voxels. As the root segment

uniquely determines the component, it suffices to label only the root. We use an implementation detail to obtain such labels efficiently: the root segment's index in the segments array by definition determines a unique label. Finally, for each segment we retrieve the root segment's label, and decompress the z-dimension to obtain individually labelled voxels in iteration order. A high-level view of the full algorithm is given in Algorithm 5. Anticipating its use in Section 3.2, Algorithm 5 does not decompress the labelled segments.

Algorithm 1: FIND. The algorithm takes after the *path halving* method of Tarjan and Van Leeuwen (1984, p. 252).

Input : Segment s.
Output: Root segment s

```
while s.parent ≠ s do
    s.parent := s.parent.parent
    s := s.parent
return s
```

Algorithm 2: UNION. The algorithm takes after Cormen et al. (2009, p. 571).

Input: Two (disjoint) segments s_1 and s_2.

```
r1 := FIND(s1)
r2 := FIND(s2)
if r1.rank > r2.rank then
    r2.parent := r1
else
    r1.parent := r2
    if r1.rank = r2.rank then
        r2.rank := r2.rank +1
```

Algorithm 3: LABEL-SEGMENTS.

Input: List of segments S.

```
for s ∈ S do
    s.label := FIND(s).label
```

3.1.1 Cost analysis

We have seen that many steps depend on the structure of the matrix, instead of just the number of non-zeroes or the density. Since such structure is difficult to express concisely, we will not attempt to give a precise count of the number of operations performed. Instead, we content ourselves with an asymptotic analysis of the various parts of the sequential algorithm. Algorithm 5 consists of three parts: segmentation, the construction of components, and finally the labelling of each segment according to its component root label.

Algorithm 4: MAKE-COMPONENTS.

Input: List of segments S.

```
for s ∈ S do
    if s.x > 0 and s has neighbours N_x at (s.x − 1, s.y) then
        for s_x ∈ N_x do
            UNION(s, s_x)
    if s.y > 0 and s has neighbours N_y at (s.x, s.y − 1) then
        for s_y ∈ N_y do
            UNION(s, s_y)
```

Algorithm 5: SEQUENTIAL-CCL.

Input : A sparse, binary 3D matrix A in COO format.

Output: A list of labelled segments S, in iteration order of A.

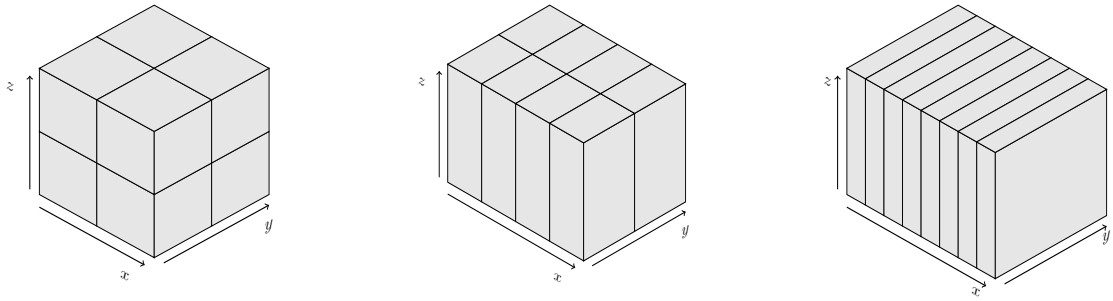
```
Compute segments from the voxels in A, and store these in S.
MAKE-COMPONENTS(S)
LABEL-SEGMENTS(S)
return S
```

It facilitates the cost analysis to introduce some formal notation. Let $A \in \{0,1\}^{a\times b\times c}$ be a sparse, binary 3D matrix of arbitrary dimensions $a, b, c \in \mathbb{N}$. Let $nz(A)$ be the number of non-zero voxels of this matrix, and equivalently the length of our COO representation. Let S be the set of segments, constructed in iteration order of A. Let $\alpha(n, m)$ be an inverse of the Ackermann function for n set creations and m applications of FIND, as explained in Tarjan and Van Leeuwen (1984).

Segmentation requires two passes over the matrix A, first to compute the number of segments, and second to construct them. The number of operations in each iteration is constant, such that this step is in $O(nz(A))$. Constructing components requires, first, the creation of $n = |S|$ individual sets for each segment, and subsequently the union of neighbouring segments. Finding such neighbours takes two binary searches over S (in $O(\log_2 n)$) for each segment $s \in S$: once for neighbours at $(x-1, y)$, and another pass for neighbour at $(x, y-1)$. The neighbouring segments are then merged using UNION. The number of such neighbouring segments depends directly on the structure of the matrix, but is generally small. We will assume the components are sufficiently smooth, such that the maximum number of neighbouring segments is indeed a small constant for each segment, and the aggregate number of times a 'neighbour' action must be performed is m, $m \in O(n)$—generally, $m \geqslant n$ (Tarjan and Van Leeuwen, 1984). Then, the merging of these components is in time $\alpha(n, m)$. We have for the entire MAKE-COMPONENTS step a complexity of $O(n \log_2 n + m\alpha(n, m))$. Finally, labelling each segment takes time $O(n\alpha(m, n))$, such that the entire algorithm is in $O(nz(A) + n \log_2 n + m\alpha(n, m) + n\alpha(m, n)) \in O(n \log_2 n + m\alpha(n, m))$. Since $\alpha(n, m)$ is near constant in its input parameters, we will leave it out of the analysis and instead write

$$T_{\text{seq}} \in O(nz(A) + n \log_2 n),$$

as $m \in O(n)$.



(a) **(b)** **(c)**

Figure 4: Data distributions of a matrix split along all three dimensions for $p = 8$, giving cubes (Figure 4a); two dimensions, giving piles (Figure 4b); and one dimension, giving slabs (Figure 4c). Each processor would receive one such block.

3.2 Parallel algorithm

Having established a sequential algorithm for the CCL problem, we now develop an parallel algorithm. As usual, we first discuss suitable data distributions in Section 3.2.1. Thereafter, we explain Algorithm 6 in Section 3.2.2, and provide a theoretical cost analysis in Section 3.2.3.

3.2.1 Data distribution strategies

As we have seen for the sequential algorithm, neighbouring voxels belong to the same component. As such, it is much preferred to keep neighbouring voxels together, rather than have them spread out over several processors. The most sensible distribution to achieve this is a block distribution. There are several ways to implement a block distribution over 3D matrices: we could slice along all three dimensions (giving cubes), two (piles), or just one (slabs). Figure 4 illustrates the resulting partitions. How do we decide?

We discussed segmentation for the sequential algorithm, which compresses contiguous voxels in the z-dimension. We want to implement this for the parallel algorithm as well, so we should the z-dimension to the same processor. This immediately rules out cubes as a valid partitioning strategy. It is more difficult to decide between piles and slabs, as superficially both appear equally valid partitioning strategies.

Suppose we were interested in minimising the shared boundary between different processors. Suppose, for simplicity, the input matrix is $k \times k \times k$, with ρ its uniform density, and p processors are used. Using slabs a processor has at most two neighbouring processors, with each boundary consisting of a slice of size $1 \times k \times k$, with an expected number of ρk^2 voxels. It follows that each processor shares $2\rho k^2$ voxels at its boundaries, if we use slabs as our partitioning strategy. For piles, we would similarly have $4\rho\frac{k^2}{p}$ shared voxels. A boundary-minimising argument thus suggests we should prefer piles over slabs. This, however, overlooks two important considerations:

1. Piles split along the x- and y-dimensions, but the y-dimension is not contiguous in our input matrix. As such, any work that must be performed along this dimension cannot be done efficiently via indexing, and would instead require searching for the relevant segments.

2. It is highly questionable if the shared *boundary* in any way relates to the number of communicated data words.

The first is stated matter-of-fact. The second needs further justification. Consider a shared boundary of size $1 \times k \times k$ between two arbitrary processors (a *slice*). Each processor has labelled the voxels on this slice with its own labels. For each disjoint component on this slice, the processors need to communicate only a single segment: the component root segment. As both processors iterate through the slice in iteration order, they will communicate the same segment, albeit with different labels, which can then be joined to obtain a consistent labelling. We will go into more detail in Section 3.2.2, but the key takeaway here is that for each disjoint component on the boundary slice, only a single segment needs to be communicated. It is not the size of the boundary that determines communication, but rather the number of shared components. As such, we immediately conjecture the communication volume after the distribution of the blocks does not depend on the actual number of voxels or segments, but rather on the number of shared components, which is generally significantly smaller.

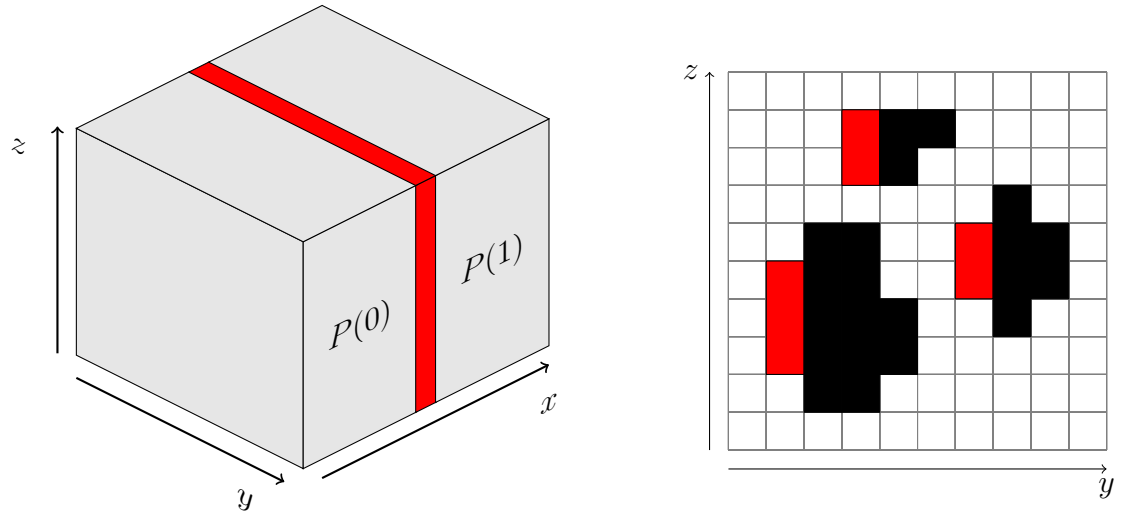
Summarising the above, we decide on a block distribution where the matrix is cut only in the x-dimension, and neighbouring processors share an overlapping slice of voxels. For balance, we need to ensure each processor receives a similar number of voxels—this, and more, will be discussed next in Section 3.2.2.

3.2.2 Supersteps

Algorithm 6 for a processor $P(s)$ consists of three supersteps. In the first superstep (0), the sparse matrix A is read from the filesystem, and distributed across all processors. In practice, for truly large matrices, one would likely want to have each processor read its voxels independently from disk, but for our datasets this was a convenient work-around implementing such file seeking behaviour. The rest of the algorithm follows all the same, no matter the precise details of how the matrix is made available across all processors in this superstep. Each processor requires the number of non-zeros $nz(A)$ for its labelling space (discussed below), and a block of contiguous voxels A_t. These voxels are assigned such that two consecutive processors share a complete slice in the x-dimension (for such a slice on the boundary between two processors, see Figure 5a). This will drastically limit communication in later supersteps, as we shall see. For balancing, each processor is assigned about $nz(A)/p$ voxels, not counting the small imbalance due to the overlapping voxels.

In the next superstep (1), each processor applies the sequential algorithm of Algorithm 5 to determine the local components in its allocated sub-matrix A_s. This results in a locally labelled set of segments S_s. If we can ensure such labels are globally unique, we would only need to communicate those components that cross over into sub-matrices owned by the other processors. This might save a lot of communication! Once way to achieve this is by sharing $nz(A)$ with each processor, as there cannot be more components than there are voxels. Each processor then uses $[snz(A), (s+1)nz(A))$ to label its local components, guaranteeing there will be no overlap in assigned labels. This cleanly labels the components local to $P(s)$, but the difficulty is in those components that cross processor boundaries, as we will see next.

After labelling local components, each processor revisits the slices it shares with its neighbours to determine if there are any component labellings that need to be communicated. This communication is necessary, as different processors have labelled such shared components using different labels, and we would like them to obtain a consistent labelling. In Figure 5, such a slice is given graphically. We will explain the argument in reference to this simple example. We could communicate all segments in this slice, but ideally we would only share those segments that belong to disjoint components. A



(a) (b)

Figure 5: Example for $p = 2$ and a single overlapping slice between the processors. In Figure 5a, such a slice is shown in red on the boundary between $P(0)$ and $P(1)$. In Figure 5b an example overlapping slice is shown in 2D. The root segments of the boundary-disjoint components are coloured red and exactly these segments will be communicated in superstep 1.

natural way to do so is to share the first segment for each component on this boundary, but this is problematic: there might components that are disjoint at the boundary, but connect at some other place in S_s. Only one segment would be communicated in this case, but the other processors might not be aware the—in their view—disjoint components belong to the same component. To circumvent this, we will copy these 2D slices, and re-apply Make-Components to the copies. By comparing the root segments of these copies against the root segments of the original slices, we can then obtain all *boundary-disjoint* component roots in a single pass over the copies, and communicate the relevant segment in the original slice. This is done the first time a copied segment has a root different from the relevant segment in the original slice: this must be a component root, and we send it to the neighbouring processor. Thereafter, we point this root to the original root, which immediately takes care of the entire boundary-disjoint component. We then iteratively traverse the slice, and repeat this procedure for each segment. In the example of Figure 5b, this ensures all the red segments are communicated, one for each boundary-disjoint component.

As both processors process these slices in iteration order, the segments they communicate *must* be the same: the procedure is entirely deterministic. In the next superstep (2), we thus have a list of boundary segments where some segments are coordinate-wise duplicates, with different labels: these are the same root segments on the boundary, from the perspective of two different, neighbouring processors. To ensure each processor can arrive at a consistent labelling in a single superstep, we communicate the boundary segments to all other processors once they are found. This is an $O(B_{\max}p)$-relation, for some processor sharing the maximum number of boundary segments $B_{\max}$.

In the third superstep (2), the processors apply some work to arrive at a consistent labelling. First, the iteration order is restored on the received boundary segments. Then, each boundary segment b_1 is compared with appropriate other boundary segments b_2 in a nested loop. The segments

b_1 and b_2 belong to the same component if one of two conditions are met: either they encode for the same coordinates, such that they came from two neighbouring processors, or they share a label. If two segments have the same label, they must be in the same component (and came from the same processor) as the label space is unique to each processor. Due to the iteration order, such appropriate segments b_2 can only succeed a boundary segment b_1 in iteration order: everything before b_1 has already been merged into a single component at this point. This immediately restricts the nested loop, but we would like to restrict it further to avoid a quadratic runtime complexity. Observe the following: as two boundary segments are merged only when their labels agree (same origin processor), or their coordinates are the same (neighbouring processors), we can stop iteration the moment the b_2 came from a processor that is not either the same as b_1's origin processor, or its next neighbour. Such origins can be determined from the assigned labels, using integer division as b_1.label div $nz(A)$ (and similarly for b_2). The lower and upper bounds ensure this nested loop is nearly linear in practice.

Finally, once these boundary segments have been merged into new 'boundary' components, we update our local labels using this newly formed component structure. We already have all the machinery in place for such a relabelling with Label-Segments, and only need to update the appropriate root segments in S_s. For each boundary segment b, we first determine if we own it using the label space trick described earlier. If we do, we find the segment s' in S_s that occupies the same coordinate as b using a binary search over S_s. We then assign b's root label to the label of s''s root segment. Once this completes, we have updated the relevant component roots in S_s, and can relabel each such segment using Label-Segments. All that remains is to decompress S_s to obtained a valid labelling of all voxels in A_s.

3.2.3 Cost analysis

The first superstep (0) is a communication superstep where $P(0)$ distributes the matrix A. We have p processors. Each processor receives $O(\frac{nz(A)}{p})$ voxels from $P(0)$, such that we have an $O(nz(A))$-relation (the numbers are not exact, as there are some overlapping slices which should not really influence the analysis). The amount of work $P(0)$ has to perform is in $O(nz(A))$, such that the first superstep is of cost $O(nz(A)) + O(nz(A))g + l$.

For superstep 1, let n_s be the number of segments in a submatrix A_s. A_s is of size $O(\frac{nz(A)}{p})$. The application of the sequential algorithm costs time $O(\frac{nz(A)}{p} + n_s \log_2 n_s)$. Thereafter, Make-Components is applied to both boundaries, at cost again $O(n_s log_2 n_s)$ (we ignore the term in m for the analysis here). Let $B_{\max}$ be the maximal number of boundary-disjoint components any one processor finds. Generally, we have $\frac{nz(A)}{p} \gg B_{\max}$. Then, we have an $O(B_{\max} p)$-relation, and the total superstep is of cost $O(\frac{nz(A)}{p} + n_s \log_2 n_s) + O(B_{\max} p)g + l$. As $n_s \in O(\frac{nz(A)}{p})$ (albeit a very weak bound), we expect this computation step to decrease in p.

The last superstep starts by sorting all received segments, at cost $O(pB_{\max} \log(pB_{\max}))$. Next, each boundary segment is compared (this step is about linear, as we explained before), and the segments are merged. Thereafter, in two loops, we apply the received segment labels to our locally owned segments. These steps are linear in $O(pB_{\max})$ and $O(n_s)$, respectively. The total cost of this step is thus of order $O(pB_{\max} \log(pB_{\max}) + n_s)$. This step could increase in p if $B_{\max}$ does not remain appropriately small, or decrease if n_s decreases faster than $pB_{\max}$ increases.

Algorithm 6: PARALLEL-CCL for processor $P(s)$, $0 \leqslant s < p$.

Input : A sparse, binary 3D matrix A in COO format on disk.
Output: Distributed, consistent component labelling of the voxels in A_s.

if $s = 0$ **then** `// Superstep 0`
  Read A from the filesystem.
  for t **from** 0 **to** $p - 1$ **do**
    Send $nz(A)$ to $P(t)$.
    Send a subset A_t to processor $P(t)$, with a single x-value overlap with processors $P(t-1)$ (if $t \neq 0$) and $P(t+1)$ (if $t \neq p-1$).
Receive $nz(A)$ and A_s from $P(0)$. `// Superstep 1`
$S_s :=$ SEQUENTIAL-CCL(A_s), using the label space $[snz(A), (s+1)nz(A))$.
if $s \neq 0$ **then**
  Copy the segments in S_s overlapping with $P(s-1)$ into S_{s-1}.
  Apply MAKE-COMPONENTS to S_{s-1}. For each root segment in S_{s-1}, send the equivalent segment in S_s to all processors $P(t)$, $0 \leqslant t < p$.
if $s \neq p - 1$ **then**
  Copy the segments in S_s overlapping with $P(s+1)$ into S_{s+1}.
  Apply MAKE-COMPONENTS to S_{s+1}. For each root segment in S_{s+1}, send the equivalent segment in S_s to all processors $P(t)$, $0 \leqslant t < p$.
Receive the boundary component roots B. `// Superstep 2`
Sort B to restore the iteration order.
for b_1 **from** 0 **to** $|B| - 1$ **do**
  for b_2 **from** $b_1 + 1$ **to** $|B| - 1$ **do**
    if $B[b_2]$'s origin processor does not neighbour or equal $B[b_1]$'s processor **then**
      break
    if $B[b_1]$.label $= B[b_2]$.label **or** b_1 and b_2 are at the same coordinate **then**
      UNION$(B[b_1], B[b_2])$
for $b \in B$ **do**
  if b is a coordinate-wise copy of an element s' in S_s **then**
    FIND(s').label := FIND(b).label
LABEL-SEGMENTS(S_s)
Decompress S_s to obtain the lablled voxels in A_s.

Taking all this together and simplifying, we find

$$T_{\text{par}} = O\left(nz(A) + \frac{nz(A)}{p}\log_2(\frac{nz(A)}{p})\right) + O(nz(A))g + 2l.$$

4 Results

We test our algorithm on the Cartesius computer cluster. Cartesius is a BSP machine with $g = 607.1$, $l = 52416.1$, and $r = 6.090$ Gflop/s, according to the program `bspbench64` of the `Edupack` software package. Its organisation varies somewhat across nodes, but most nodes consist of 24 2.4 to 2.6 GHz processors, with 64 GB of memory attached. The results below were gathered as follows: once we cross 16 cores, we scale up across nodes: every *next* 16 cores are allocated on different nodes, such that for 32 cores we use two nodes, and for 64 we use four.

To reliably test our algorithms, we first need several datasets. Some good benchmarking datasets exist, so there is no need to reinvent the wheel. We will use the 3D images made available by the YACCLAB ("Yet Another Connected Components Labelling Benchmark") project for our analyses (Allegretti et al., 2020; Bolelli et al., 2018; Grana et al., 2016), including a number of synthetic images to test the effects of various matrix densities. As a pre-processing step, we first convert these images to the familiar COO format our algorithm expects. Characteristics of the selected datasets are given in Table 2, and their origin is briefly explained below.

Table 2: Three datasets we apply and test the algorithm on. The datasets are described by the characteristics, such as the number of components, dimensions, number of non-zeroes, and number of segments.

Name	Num components	Dimensions	$nz(A)$	Num segments
hilbert2	1	$128 \times 128 \times 128$	$2,647$	$1,723$
mitochondria1	45	$1024 \times 768 \times 165$	$6,888,434$	$346,922$
mitochondria2	42	$1024 \times 768 \times 165$	$7,306,042$	$354,980$

The **hilbert2** dataset gives the second iteration of the 3D Hilbert curve, which is a continuous, space-filling fractal. The curve is fully connected and thus consists only of a single component. It consists of a single voxel 'line' in 3D space. Because of its many twists and turns, the number of segments is relatively close to the number of non-zeroes, which makes for a large number of boundary-disjoint components. These features make the Hilbert curve an interesting dataset to verify the correctness of the algorithm. Due to this unique nature, the Hilbert problem is an instance where segmentation has only limited success, as the number of segments is only a factor 1.5 smaller than the number of non-zeroes.

The **mitochondria1** and **mitochondria2** datasets present the presence of mitochondria in a section of the CA1 area of the hippocampus. The original measurements are on a $5 \times 5 \times 5\mu$m section of the brain obtained via electron microscopy, corresponding to a $1065 \times 2048 \times 1536$ volume. The datasets are subsets of this original measurement where the ground truth of the location of a mitochondrion is annotated. These present a real-world problem instance for which the algorithm

is suitable because of the matrix sparsity (its density is around 5%) and its large size. It is also interesting to note the number of segments is over a factor 20 smaller than the number of non-zeroes.

In Figure 6 we give runtimes of the algorithm for each superstep. We split this into an analysis of the computation times, and subsequent global synchronisation where communication takes place. We omit these plots for superstep 2, as this step has no communication volume, and is computationally rather inexpensive. These and other details are given in Tables 3 to 5 in Appendix A. It is immediately clear that most time is spend on the computation step of superstep 0 (Figure 6a), where the matrix is distributed across the processors. The increasing computation times for $p \geqslant 16$ are worrisome, and indicate an inefficient implementation of the distribution process. Furthermore, the communication costs of Figure 6c are rather high and seem to increase in the number of processors (although only a negligible amount of additional data is distributed). Emboldened by the reduction in size segmentation may provide (*cf.* Table 2), we improve the distribution process and move segmentation from the second superstep to the first, and communicate segments, rather than voxels.

The results of the improvement can be seen in Figure 8 and are rather impressive. As we had hoped, the computation time in the first superstep no longer increases in the number of processors, and appears lower overall—despite the additional segmentation. Furthermore, the communication volume is indeed much smaller, as indicated by the shorter communication time for the first superstep (for both mitochondria datasets). For the **hilbert2** dataset, we do observe an increase in communication cost due to the limited reduction segmentation provides, and the increased cost of a segment over a simple voxel. Because of these apparent improvements, the remainder of our analysis will focus on the results of the new implementation, sharing segments rather than voxels.

As we explained previously, memory efficiency is a particular concern for our algorithm, as it is intended to work with large, sparse matrices. As such, we investigate memory use as a function of the number of processors. In Figure 7a memory consumption is given as a function of the number of processors, on a log-log axes. For the datasets **mitochondria1** and **mitochondria2** the figure shows a (nearly) linear downward slope, indicating that for k times as many processors, memory needs decrease by the same factor k. The memory reduction for these two datasets is thus nearly perfect, indicating a very good load balance. The **hilbert2** dataset shows the same downward line until about $p = 4$, but remains almost constant thereafter. This is due to the specific shape of the Hilbert curve, which results in a large number of line segments for one x value, whereas an entire slice might only be a single voxel for another. Although the number of slices in a processor roughly halves when the number of processors is doubled, the number of segments will largely be determined by a single slice with many segments. Once we hit this lower bound, the maximum memory consumption by any one processor cannot decrease further. The communication volumes of Figure 7b are rather modest for all datasets: several kilobytes in the worst case, for all matrix sizes. This observation reinforces our claim that there is hardly any relation between the size of the matrix, and the communication volume between processors in later steps of the algorithm.

The first computation superstep in Figure 8a is expensive, and remains so as the number of processors increases. This is to be expected, as here only processor $P(0)$ performs work, including expensive file I/O to obtain the sparse matrix. Its associated communication cost is given in Figure 8c, and slowly increases until we exceed a single node, and remains fairly flat thereafter (the slight increases are due to the extra cost of have more overlapping slices for larger p). Altogether, for the amount of data shared (up to a hundred MB for the mitochondria datasets), the communication speed is rather high.

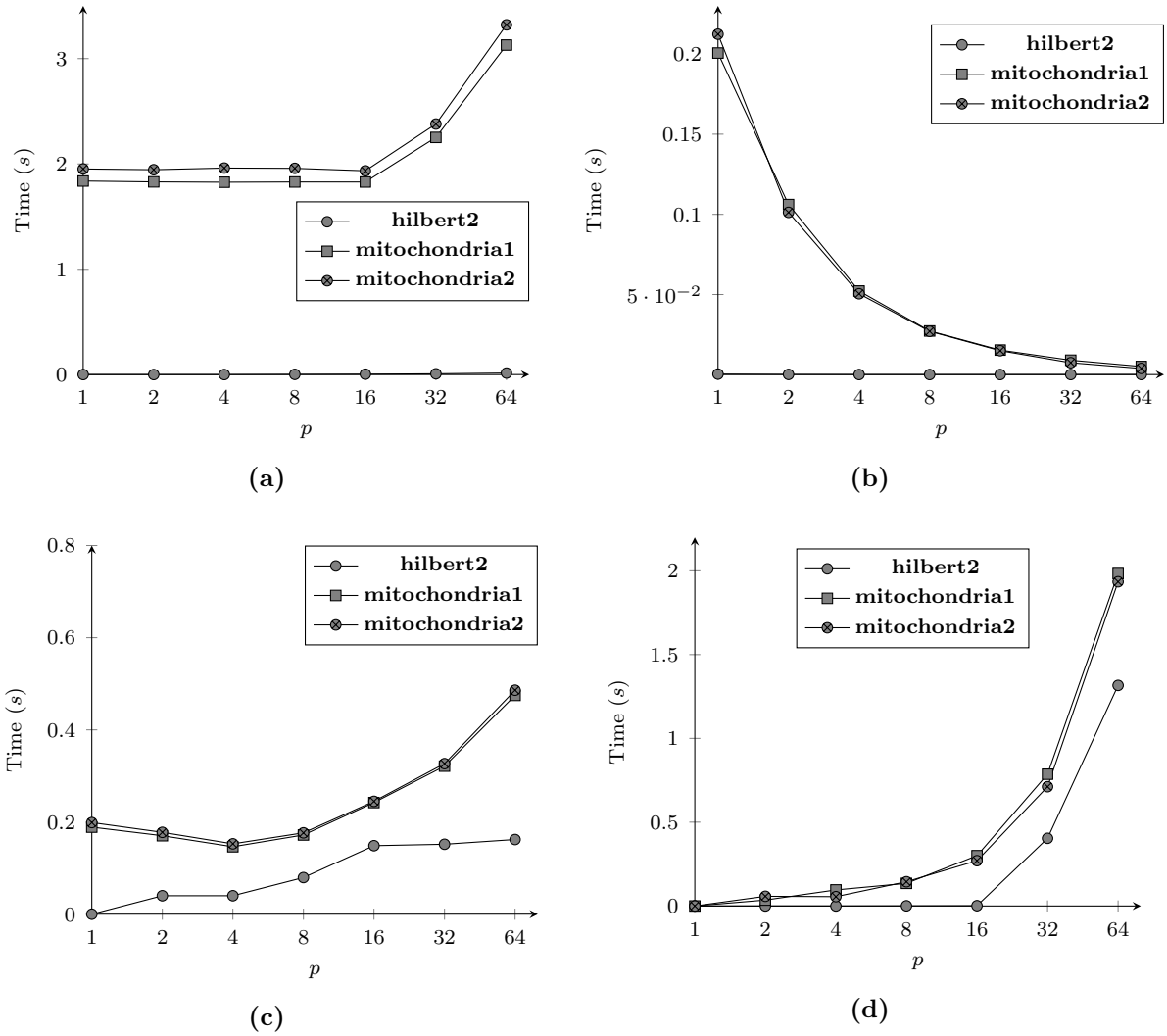


Figure 6: Running times (seconds) for the first two super-steps when communicating voxels, as functions of the number of processors. Figures 6a and 6b give the computation time for respectively supersteps 0 and 1; Figures 6c and 6d the communication times. Each data point is the average of ten runs, to rule out any statistical artefacts.

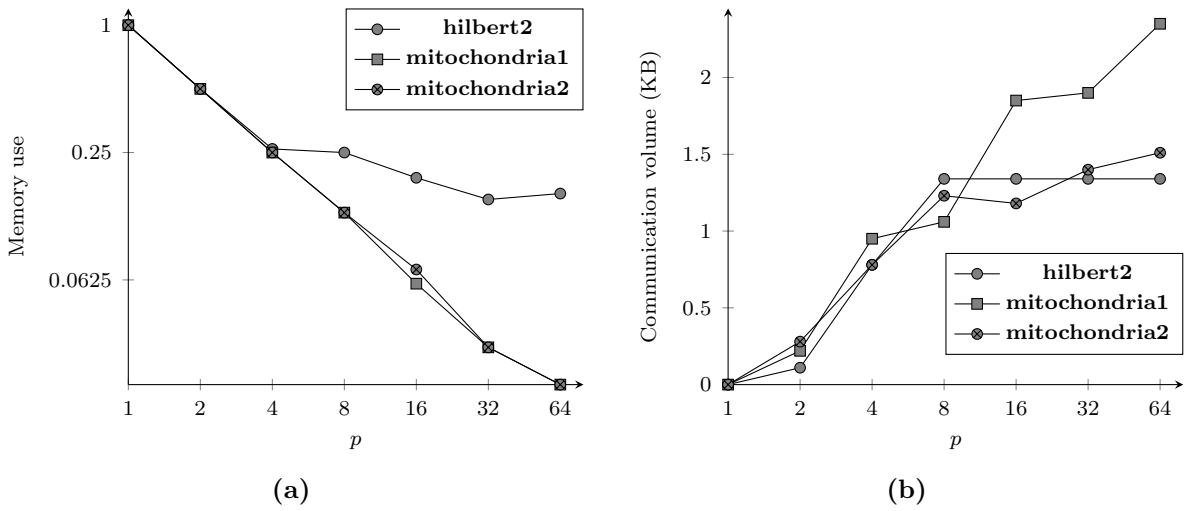


Figure 7: Figure 7a shows the decrease in memory consumption, as a function of the number of processors (for $p > 1$, this gives the maximum memory used by any one processor). Memory use is normalised to 1 for $p = 1$. Figure 7b gives processor communication volumes, as the maximum communication volume originating from one processor to the others in superstep 1.

The computation cost of the second superstep in figure Figure 8b and decreases as the number of processors increase, as we expected in Section 3.2.3. As can be seen in Tables 7 and 8 in Appendix A, the speed up for the mitochondria datasets are linear in the number of processors. The speed up for the **hilbert2** dataset is about 4 for $p \geqslant 4$ and about 2 for $p = 2$. This means a linear speed up until $p = 4$ and no more speed up for more processors, which is similar to the results of the memory use shown in Figure 7a. The Hilbert dataset is simply too small to make effective use of many processors.

The communication cost associated with superstep 1 are given in Figure 8d. This process is decidedly non-linear, although it is unclear *why*: the communication volume increases as the number of processors increases, but it does not do so according to an explosive process. To understand this better, see *e.g.* Figure 7b, where after $p = 8$ for **hilbert2** and **mitochondria2** the communication volume does not appear to increase significantly. Furthermore, the computation cost of the subsequent superstep increases sublinearly in *e.g.* Table 7, but depends directly on the number of communicated boundary segments as explained in Section 3.2.3. In an initial implementation we sent each boundary root segment separately, resulting in several messages to be passed to each processor. Suspecting the growing cost of communication might have something to do with these messages not being bundled appropriately, we switched to passing a single message containing all root segments, such that each processor emits only a single, large message to all other processors. This, however, resulted in no improvement in time spent communicating. The `BSPonMPI` profiler indicated these messages were all passed using the `MPI_Alltoall` routine, which we initially suspected to be the cause for this suboptimal performance. Decreasing the number of bytes that triggers a switch to the more efficient MPI message passing methods (the `--small-exchange-size` flag) did not cause a significant change in the duration of this communication step. As a final attempt, we decided to switch to the dedicated message-passing methods using the `--msg` flag. This again had only a negligible impact on the communication step. There is some inefficiency here, and it is unclear to us who's at fault.

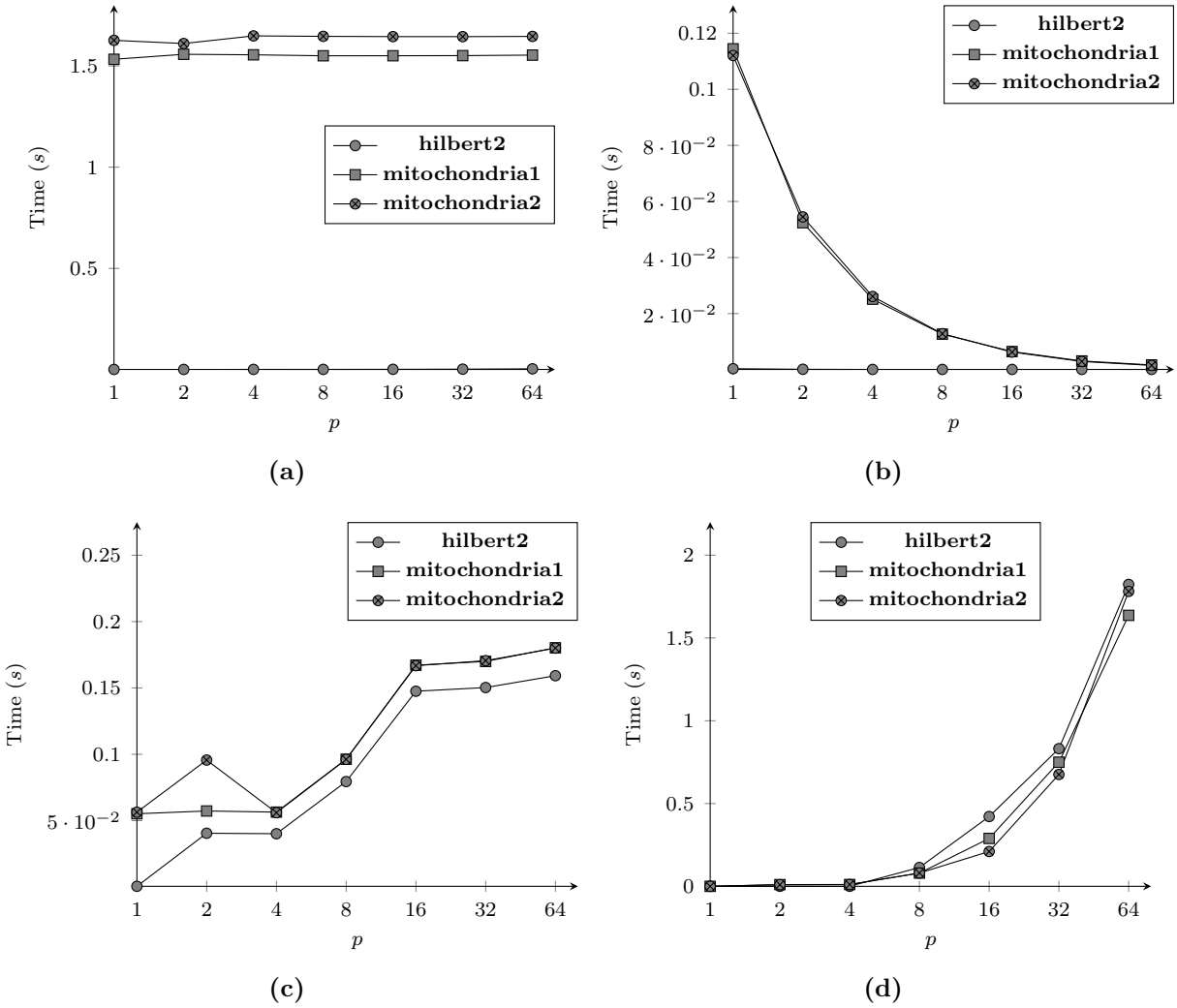


Figure 8: Running times (seconds) for the first two super-steps when communicating segments, as functions of the number of processors. Figures 8a and 8b give the computation time for respectively supersteps 0 and 1; Figures 8c and 8d the communication times. Each data point is the average of ten runs, to rule out any statistical artefacts.

5 Conclusions

In summary, we introduced a parallel algorithm to label connected components in a sparse, binary, 3D matrix. Our main was reduction in memory consumption, while retaining good, near-linear computation speeds. The results show that for large enough problem size the memory reduction is nearly perfect. The communication volume remains small (at most a $2.4p$ KB is send per processor, which is hardly a tight bound—in practice, much less is shared), we obtain a near perfect speed up in sequential labelling. For the first data-sharing superstep, we obtain a high communication speed for the amount of data distributed, although for truly large datasets, this will most likely be replaced by direct filesystem access. Most can be gained by reducing the communication cost of the boundary components. We have no good explanation for the increasing communication times in this step. The reason for this increase should be investigated and solved.

The algorithm can be tested further by looking at the larger input size. This can be achieved by upsampling the matrices used by doubling each voxel in every dimension, and thus replacing every voxel by eight voxels. The resulting matrix now represents an image with larger resolution with the same connectivity properties. This can be repeated as often as needed. As the above analysis indicates, this should not result in increases in communication beyond the first superstep, as the number of boundary components does not increase. Other properties of the algorithm can be evaluated by testing it on synthetic images with varying degrees of connectivity.

Connected component labelling is often used as an intermediate (pre-processing) step in larger algorithm, including machine learning pipelines. In such settings, specific knowledge can be used to reduce computation times further. As a heuristic, select for x a large dimension to reduce the overhead of overlapping slices, and for z a dimension that can be compressed well using segmentation. Which dimensions these are depends very much on the problem domain, but it is very worthwhile to investigate this on a representative dataset.

6 Author contributions

Alexandra wrote the introduction, problem definition, parallel solution method, and parts of the results and conclusions. She contributed to the parallel solution method. Niels wrote the sequential solution method, and parts of the results and conclusions. He implemented the algorithms in computer code and tested them on the Cartesius cluster.

References

Allegretti, S., F. Bolelli, and C. Grana (2020). Optimized block-based algorithms to label connected components on GPUs. *IEEE Transactions on Parallel and Distributed Systems 31*(2), 423–438.

Bolelli, F., M. Cancilla, L. Baraldi, and C. Grana (2018). Toward reliable experiments on the performance of connected components labeling algorithms. *Journal of Real-Time Image Processing 1*(1), 1–16.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2009). *Introduction to Algorithms* (3 ed.). MIT Press.

Grana, C., F. Bolelli, L. Baraldi, and R. Vezzani (2016). YACCLAB - yet another connected components labeling benchmark. In *23rd International Conference on Pattern Recognition*. ICPR.

Kwong, Y., A. O. Mel, G. Wheeler, and J. M. Troupis (2015). Four-dimensional computed tomography (4DCT): A review of the current status and applications. *Journal of Medical Imaging and Radiation Oncology 59*(5), 545–554.

Molinier, M., T. Häme, and H. Ahola (2005). 3D-connected components analysis for traffic monitoring in image sequences acquired from a helicopter. In H. Kalviainen, J. Parkkinen, and A. Kaarna (Eds.), *Image Analysis*, Berlin, Heidelberg, pp. 141–150. Springer Berlin Heidelberg.

Nina Paravecino, F. and D. Kaeli (2014). Accelerated connected component labeling using cuda framework. In L. J. Chmielewski, R. Kozera, B.-S. Shin, and K. Wojciechowski (Eds.), *Computer Vision and Graphics*, Cham, pp. 502–509. Springer International Publishing.

Ohira, N. (2018, Mar). Memory-efficient 3D connected component labeling with parallel computing. *Signal, Image and Video Processing 12*(3), 429–436.

Ram, S., J. J. Rodríguez, and G. Bosco (2012). Segmentation and detection of fluorescent 3d spots. *Cytometry Part A 81A*(3), 198–212.

Sanderson, C. and R. Curtin (2019). Practical sparse matrices in C++ with hybrid storage and template-based expression optimisation. *Mathematical and Computational Applications 24*(3), 70.

Tarjan, R. E. (1975). Efficiency of a good but not linear set union algorithm. *Journal of the ACM 22*(2), 215–225.

Tarjan, R. E. and J. Van Leeuwen (1984). Worst-case analysis of set union algorithms. *Journal of the ACM 31*(2), 245–281.

Zhao, H. L., Y. B. Fan, T. X. Zhang, and H. S. Sang (2010). Stripe-based connected components labelling. *Electronics Letters 46*(21), 1434–1436.

A Appendix

Table 3: Runtimes for each superstep when communicating voxels for the **hilbert2** dataset. These runtimes are in seconds, averaged over ten runs.

	Superstep 0		Superstep 1		Superstep 2		Total
p	Comp.	Comm.	Comp.	Comm.	Comp.	Comm.	
1	0.0010568	0.0002327	0.0003776	0.000004	0.0000119	—	0.0022366
2	0.0010506	0.0402068	0.0001716	0.0000816	0.0000088	—	0.0424275
4	0.0010528	0.0402305	0.0001009	0.000117	0.0000291	—	0.0427047
8	0.0029695	0.0796423	0.0000848	0.0008953	0.0005304	—	0.0859279
16	0.0048409	0.1487747	0.0000911	0.0017106	0.0009263	—	0.1583268
32	0.0083874	0.1518469	0.0000898	0.4036277	0.0013254	—	0.8566008
64	0.0155049	0.162147	0.0001062	1.3161445	0.0022687	—	2.1094997

Table 4: Runtimes for each superstep when communicating voxels for the **mitochondria1** dataset. These runtimes are in seconds, averaged over ten runs.

	Superstep 0		Superstep 1		Superstep 2		Total
p	Comp.	Comm.	Comp.	Comm.	Comp.	Comm.	
1	1.8379985	0.1889993	0.2006006	0.0000127	0.0017709	—	2.2299373
2	1.8298724	0.1703733	0.1061063	0.0353236	0.0010252	—	2.1436019
4	1.8266324	0.1464775	0.0522004	0.0957951	0.0006821	—	2.1230036
8	1.8288703	0.1719733	0.0272589	0.1349888	0.0009883	—	2.1659117
16	1.828686	0.2423228	0.0153457	0.300022	0.0010245	—	2.3893935
32	2.2517814	0.3213009	0.0090324	0.786378	0.0011159	—	3.6567071
64	3.1284062	0.4748454	0.0051255	1.9836856	0.0011869	—	6.2060883

Table 5: Runtimes for each superstep when communicating voxels for the **mitochondria2** dataset. These runtimes are in seconds, averaged over ten runs.

	Superstep 0		Superstep 1		Superstep 2		Total
p	Comp.	Comm.	Comp.	Comm.	Comp.	Comm.	
1	1.9525178	0.1989298	0.2123538	0.0000124	0.0018208	—	2.3662074
2	1.9453446	0.1777549	0.1012364	0.0577727	0.0011523	—	2.2841625
4	1.9611393	0.1528101	0.0504757	0.0556217	0.0005911	—	2.2218276
8	1.9585322	0.1766468	0.0270452	0.1446304	0.0009061	—	2.3094772
16	1.9350603	0.2449775	0.0149655	0.2700871	0.0008877	—	2.4681002
32	2.3789658	0.326779	0.0074629	0.7116242	0.0011056	—	3.7160492
64	3.3202251	0.4860833	0.0039028	1.93475	0.0013317	—	6.3623235

Table 6: Runtimes for each superstep when communicating segments for the **hilbert2** dataset. These runtimes are in seconds, averaged over ten runs.

	Superstep 0		Superstep 1		Superstep 2		Total
p	Comp.	Comm.	Comp.	Comm.	Comp.	Comm.	
1	0.0010711	0.0005134	0.000336	0.0000037	0.0000109	—	0.0024801
2	0.0010346	0.0404956	0.0001428	0.0001817	0.0000087	—	0.0427411
4	0.0012943	0.0401492	0.0000769	0.0001851	0.0000284	—	0.04291
8	0.0013327	0.0794449	0.0000799	0.1137375	0.0009197	—	0.198002
16	0.0017292	0.1475254	0.0000828	0.4214611	0.0014486	—	0.5755945
32	0.0025156	0.1503	0.0000836	0.8314953	0.0031986	—	1.2707456
64	0.0040956	0.1592218	0.0000872	1.8232684	0.0036901	—	2.6012516

Table 7: Runtimes for each superstep when communicating segments for the **mitochondria1** dataset. These runtimes are in seconds, averaged over ten runs.

	Superstep 0		Superstep 1		Superstep 2		Total
p	Comp.	Comm.	Comp.	Comm.	Comp.	Comm.	
1	1.532867	0.0551919	0.1144468	0.0000119	0.0017624	—	1.7048327
2	1.5575475	0.0572742	0.052411	0.010409	0.0008911	—	1.679429
4	1.5550043	0.0564295	0.025142	0.0109737	0.0004672	—	1.6492053
8	1.5502021	0.0962745	0.0126708	0.0797585	0.0015777	—	1.7425459
16	1.5503842	0.1671327	0.0064598	0.2892527	0.0023201	—	2.0182083
32	1.5509681	0.1699763	0.0031037	0.7497802	0.0025056	—	2.7640018
64	1.5535455	0.1802447	0.0016886	1.6365925	0.003375	—	3.9814784

Table 8: Runtimes for each superstep when communicating segments for the **mitochondria2** dataset. These runtimes are in seconds, averaged over ten runs.

p	Superstep 0 Comp.	Comm.	Superstep 1 Comp.	Comm.	Superstep 2 Comp.	Comm.	Total
1	1.6265047	0.0563774	0.1120498	0.0000115	0.0018027	—	1.7973044
2	1.6107397	0.0956572	0.054485	0.0095522	0.0009147	—	1.7722569
4	1.6484189	0.0559996	0.026108	0.0107771	0.0004773	—	1.742982
8	1.6461539	0.096205	0.0128261	0.0803306	0.0014813	—	1.8391852
16	1.6448088	0.1667902	0.0062621	0.2102239	0.0027548	—	2.0330808
32	1.6446857	0.1705708	0.0029228	0.6756472	0.0028178	—	2.7841201
64	1.6457987	0.1800116	0.0015107	1.7812471	0.0033063	—	4.239296

Table 9: Memory (Mem.) use and communication (Comm.) volumes for the various datasets, as a function of the number of processors. Both quantities are expressed in kilobytes. The communication volume gives the largest communication in superstep 1 performed by any one processor, and thus offers an upper bound on the total communication performed by all others.

Dataset		p 1	2	4	8	16	32	64
hilbert2	Mem.	96.5	48.3	24.6	24.3	18.8	14.4	15.1
	Comm.	0	0.1	0.8	1.3	1.3	1.3	1.3
mitochondria1	Mem.	19,427.6	9,721.3	4,882.8	2,466.7	1,258.8	655.8	355.1
	Comm.	0	0.2	1.0	1.1	1.8	1.9	2.4
mitochondria2	Mem.	19,878.9	9,953.6	4,997.6	2,523.1	1,292.4	677.8	372.5
	Comm.	0	0.3	0.8	1.2	1.2	1.4	1.5