Connected Set Filtering on Shared Memory Multiprocessors

Submitted in partial fulfillment of the requirements
towards the M.Sc. degree

by

Florian Biermann
fbie@itu.dk

IT University of Copenhagen
Denmark

The research work in this thesis has been carried out
under the supervision of Prof. Peter Sestoft.

June 2, 2014

Abstract

This thesis reports on the development of a family of parallel algorithms for computing area openings and closings on gray-scale images, in order to increase performance on large images. Previous parallel and sequential algorithms did not fully exploit the parallel nature of contemporary computers. The novelty in this thesis is that the developed algorithms use variants of a shared union-find data structure. This shared union-find approach reduces the amount of sequential work substantially.

By conducting structured performance experiments, I show that, in particular, a certain type of the new, parallel algorithms outperforms the original, sequential algorithm by far on large images. Moreover, by identifying formal correctness properties and by using Java Pathfinder to model check the respective implementations, I show that the new parallel algorithms are formally correct. Additionally, I discuss the feasibility of model checking complex image filtering algorithms.

Throughout the thesis, I review different variants of concurrent union-find and asses their performance on random graphs, as well as their adaption to parallel area opening. The presented results show that the structure of random input graphs has a greater influence on performance than their mere size. Furthermore, I discuss input image structure in relation to the performance, exhibited by the various parallel area opening algorithms. Performance experiments show that performance limitations on random graphs do not necessarily apply to the performance on images. This thesis opens up for many future research directions on parallel morphological algorithms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over recent years, the quality and average size of digital images has increased tremendously. Additionally, the mere count of digital images produced every day is increasing, too. There are various reasons to analyze digital images automatically, for example in the medical domain. Here, modern clinical scans, such as magnetic resonance tomography (MRT) or computing tomography (CT), produce huge amounts of data. In order to speed up the analysis of such huge images, there has always been a trend to parallelize image analysis algorithms, as some of the problems on images are embarrassingly parallel. Nevertheless, there are also operations on images that are hard to parallelize, because the possibilities for parallelism are not obvious. One of these operations is morphological filtering.

Morphological filters are operators on two or three dimensional images that filter connected components from the input image. They are defined in terms of openings and closings, where an opening removes bright regions from an image while a closing removes dark regions [1].

More complex, shape preserving morphological filters are *area opening and closing* (I will refer to these filters as *area opening* through the rest of this thesis). Given a minimum area parameter λ, all bright or dark elements respectively of size less than λ are removed [2]. Area opening is useful for image segmentation and noise removal. There already exist a number of algorithms to compute area opening, differing tremendously in performance [2, 3, 4]. These three algorithms are all implemented sequentially.

In this thesis, I report on the development of a family of parallel algorithms for area opening for the sake of increased performance on large images. Wilkinson et al. [5] developed a parallel algorithm to compute morphological attribute filters, a generalization of morphological area filtering, using Max-Trees. However, it consists of a major sequential part [5]. The novelty in this thesis is that the algorithms presented here are based on the union-find area opening algorithm by Meijster and Wilkinson [4] and use variants of a shared union-find data structure [6] in order to reduce the amount of sequential work substantially. I will show that some of the new parallel algorithms can outperform the original, sequential algorithm and also that the new parallel algorithms are formally correct.

Throughout this thesis, I use valid Java code to describe algorithms, except where stated otherwise. For brevity, some less important implementation details may be omitted. These are replaced by descriptive comments.

1.1 Contributions

The parallel algorithms, that I developed in this thesis, are based on the union-find based area opening algorithm by Meijster and Wilkinson [4]. To fully exploit parallelism, it is therefore important to address shared union-find data structures. Concurrent union-find data structures have already been subject of research [7, 8].

The contributions of this thesis are the following:

- Reviewing concurrent union-find data structures, their performance and how they can be applied to our problem. Not all union-find algorithms directly map to area opening. Dedicated performance experiments provide us with insights into which algorithm best fits our needs.
- Developing and evaluating a number of parallel algorithms for morphological area opening. I will evaluate algorithms in terms of performance and provide a set of repeatable experiments that show how the algorithms behave, given certain input over a range of processors to run on.
- Verifying the correctness of the improved algorithm using Java Pathfinder [9]. Even though experimental testing can increase confidence, the inherent non-determinism of thread scheduling and delays in parallel programs makes it strictly speaking impossible to test all possible execution branches. Therefore, I show the correctness of the parallel area opening algorithms by more formal methods.

1.2 Programming Shared Memory Multiprocessors

Software for shared memory multiprocessors has been subject of research for a long time already [10]. Yet, programming concurrent and parallel applications, especially with low-level synchronization primitives, has not hit the main stream entirely yet [10]. In this section, I will give a brief overview over the basics of programming for shared memory multiprocessors.

1.2.1 Blocking Programs

Blocking parallelism describes programs where independent processes, or threads, wait for each other in order to make progress [10]. This waiting is achieved by the use of locks, which protect resources from concurrent access.

In such a program, a thread A has to wait, while thread B holds the lock to a resource R. As soon as B releases the lock, A can attempt to acquire the lock for R, blocking B from accessing R [10]. This scheme can result in deadlocks (A waiting indefinitely for B to release the lock or vice versa), if access to the shared resource via locking is not implemented consistently or, if a thread terminates unexpectedly while holding the lock, thus never being able to release it again.

Also, waiting threads cannot make any progress while the resource is locked, such that precious computing time goes wasted [10].

One can not only block resource access, but also principal progress, using barriers. Barriers typically block the progress of each thread, that reached the barrier, until all other threads also arrived at this point in the code. Only then all threads are allowed to continue together [10].

1.2.2 Lock-Free and Wait-Free Programs

Lock-free and wait-free programs are closely related. The wait-free property is stronger than the lock-free property: lock-free algorithms are defined by the absence of locks, while wait-free algorithms are guaranteed to terminate in a bounded number of steps [11] with all threads *always* making progress.

Theoretically, wait-free algorithms outperform lock-free algorithms, but recent research by Alistarh et al. [11] suggests that lock-free programs in practice actually exhibit wait-free performance. Their results suggest, that constructing extremely complex, wait-free algorithms might be unnecessary [11].

Lock- and wait-free algorithms perform in general much better in comparison to lock-based algorithms. To be able to update values and references consistently, there exist synchronization primitives as CPU instructions, which, in Java, are exposed to the programmer as function calls on container types.

Compare and Swap

The compare-and-swap (*CAS*) primitive atomically, i.e. as a single instruction on the CPU, performs the equivalent to the following code (here denoted in C++) [10, 12]:

```cpp
template <typename T>
bool CAS(T* r, const T& e, const T& n)
{
  if (*r == e) {
    *r = n;
    return true;
  }
  return false;
}
```

Given a pointer to some variable `r`, a reference to an expected value of the same type `e`, and a replacement value `n`, the primitive replaces the value, which `r` points to, only, if `r` points to `e` at the time of the call. *CAS* returns a boolean to the caller, indicating, whether the update was successful. The caller can then react to the failure, for example by simply updating its local copy of the value and re-trying the operation until it succeeds.

Essentially, the caller only can update a variable if he is up-to-date with its latest value [12]. This control structure can be used to build entire data structures, as we will see next. A major issue is the lack of composition, meaning that it is impossible to update two or more independent variables atomically [10].

Java does not pass values by pointer, so the value of a function parameter cannot be changed globally by a function. Therefore, Java implements *CAS* through the `AtomicReference` class, which references some object internally. Its public member function `compareAndSet` corresponds to *CAS* [12]. Java also provides multiple specialized atomic data types, like for instance `AtomicInteger` or `AtomicLong`, as well as array variants of those types [12].

Michael-Scott Queue

A well known wait-free data structure is the Michael and Scott [13] queue (*MSQ*). The *MSQ* is a first-in-first-out queue, using only the compare-and-swap primitive to synchronize between threads that concurrently access it.

Internally, the *MSQ* maintains two pointers to the `head` and the `tail` of a linked list, where the `head` always is a dummy node. By checking the state of the next pointer of the `tail` and the `head` node, the algorithm can figure out, whether some thread currently is executing an update on the queue. If that is the case, all threads work towards maintaining consistency on the queue. If not, a new update can be initiated. This pattern makes the *MSQ* wait-free and prone to unexpected thread termination [12, 13]. The `ConcurrentLinkedQueue` class provides an *MSQ* based implementation in the JVM.

1.2.3 Transactional Memory

Transactional memory is a concept borrowed from relational data bases, where consistency is the most important property. In data base systems, a transaction is a concatenation of modifications that only commit (i.e. become permanent), if no other changes have been made to the data touched by this transaction, since it began [10].

When applied to shared memory multiprocessors, concurrent threads issue a transaction for any number of memory modifications they want to perform atomically. Equally, these transactions only commit, if there are no conflicts due to other threads writing to the same memory locations. Otherwise, the changes get rolled back and the transaction, depending on the re-try policy of the system, can be re-tried or simply aborted [10].

From the outside, a transaction is seen as a single atomic update of multiple variables. This is a huge advantage compared to *CAS*. Also, it reduces complexity, since the synchronization is entirely transparent to the programmer [10]. Today's main-stream hardware does not support transactional memory. Instead, software transactional memory (*STM*) has been subject to research, originally proposed by Shavit et al. [10, 14].

Neither OpenJDK nor the Oracle JVM implement transactional memory. Instead, there exist a number of *STM* libraries for Java. The code for this thesis uses the *multiverse* library, which is open source and freely available [15]. The algorithms presented in this thesis use *multiverse* as a black-box to gain access to transactional memory capabilities. Therefore, I will not provide a detailed description of *STM* algorithms. Further information on the implementation details of *multiverse* can be found on the library's website [15].

Chapter 2

Mathematical Morphology

This chapter introduces the notion of mathematical morphology, reflects on its formalism and details the special case of morphological area opening. We will take a look at use cases and show the effects of area opening on 2D images. The chapter concludes with the introduction of the algorithm, which is the basis for the algorithms developed in this thesis.

Sections 2.1 and 2.2 are based on a previous project I conducted [16]. The contents are corrected and cut down to the most relevant parts, for brevity.

2.1 Morphological Area Opening

This section briefly outlines the formalism of morphological area opening. I will not go into the details of lattice theory and the foundations of mathematical morphology. A good reference point for the interested reader is *An overview of morphological filtering* by Serra and Vincent [1].

Mathematical morphology is, intuitively speaking, the formal study of shapes. Morphological filters operate on partially ordered sets. Therefore, morphology only applies to single-channel images, as their elements (i.e. pixels) can be ordered by their height (i.e. gray-scale value) [1]. While morphological filtering can also be performed on binary images, I will omit this case. Computing morphological area openings on binary images is trivial and reduces to flat component labeling.

Two elementary operations in mathematical morphology are *dilation*, denoted δ, and *erosion*, denoted ϵ, which are mathematical duals. They are digitally implemented using a structuring element, for example a disk, a square or a hexagon. Intuitively, erosion extends dark components of an image, by moving the structuring element around their outer borders. Dilation does the opposite, narrowing dark elements, by moving the structuring element on the inner border of the component. The concatenation of both, $\gamma = \epsilon \circ \delta$, produces the morphological opening of an image. Closing, the respective dual, is defined as $\varphi = \delta \circ \epsilon$ [1]. As already pointed out in chapter 1, we will for the remainder only focus on morphological openings, since area opening and closing are mathematical duals: computing area opening on an inverted image and inverting the result again, corresponds to computing area closing [3].

Computing the morphological opening of an image I, for a structuring element S, removes all bright components from I, that do *not* resemble the shape of S [1]. This essentially means that, in order to separate information from the image (i.e. to perform segmentation), we are required to provide concrete, a priori, non-parametric knowledge [2], which sometimes is not available or not applicable. To work around this problem, a union of multiple openings with different structuring elements can be constructed, at the cost of filtering an image repeatedly.

To provide more general means of filtering connected components from an image, Vincent [2] introduced *morphological area opening*, denoted as γ_λ^a. In contrast to classical morphological opening, area opening is performed with a parameter of *area*, or size, called λ. One can see area opening as classical morphological opening with a dynamically shaped structuring element, that adjusts to the structure of the image and grows up to a certain size. All bright components of an image of size less than λ are removed in its area opening [2]. Section 2.2 details the effects of morphological area opening.

Let an image be represented by a function $f : I \rightarrow \mathbb{R}^*, I \subset \mathbb{R}^2$. Let, moreover, $T_h(x) \subseteq I, x \in I$ be the connected component to which x belongs at some gray-value threshold level h. Also, let $||$ denote set cardinality. Then, area opening is defined by [2]:

$$\gamma_\lambda^a(f)(x) = sup\{h \leq f(x) |\ |T_h(x)| \geq \lambda\} \tag{2.1}$$

This is essentially incremental region-growing for each gray-level threshold of the input image [2]. This formalism can directly be projected onto code, resulting in very poor performance, however. A number of more efficient algorithms have been conceived, which are outlined in the following section 2.3.

2.2 Usage Example

Typically, area opening is used to either remove unwanted elements, like noise, from an image, or to identify objects in a scene. Vincent originally proposed to use area opening to identify micro-aneurysms in images of eye blood-vessels [2]. One would first compute the area opening of the image for some λ, which corresponds to the expected size of aneurysms in pixels. To identify the location of the aneurysms, it is enough to compute the pixel-wise difference between the original image and its area opening.

This approach can be used in many different settings. Morphological area opening has been used in literature for the segmentation of red blood cells in the case of automated Malaria diagnosis [17, 18]. In figure 2.1a, we can see an image of human red blood cells (800×600 pixels), infected with Malaria parasites. The successive images, 2.1b through 2.1d, show the area closing of the source image for increasing values of λ in pixels. We use area closing here, because the blood cells are darker than the background. Alternatively, one can simply compute the negative of the image before and after opening it [3]. The bigger λ gets, the more cells are removed from the image, while the background remains unmodified. By those means, we are able to compute an approximation of the background model, which is helpful for separating the single blood cells from the image background.

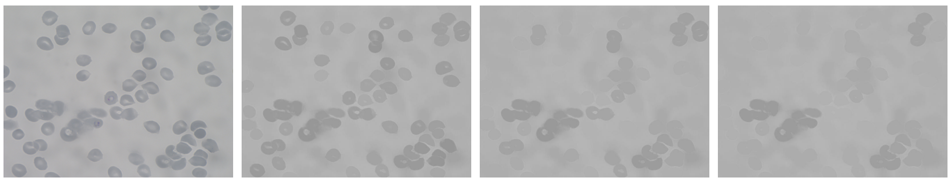

(a) Original image (b) $\lambda = 1500$ (c) $\lambda = 3000$ (d) $\lambda = 4500$

Figure 2.1: Area closing (dual to area opening) for increasing values of λ on an image of red blood cells infected with Malaria (800×600 pixels). The larger λ becomes, the more blood cells are filtered from the image.

2.3 Algorithms

There exist three well studied algorithms for computing morphological area opening. Vincent [2] originally described an algorithm based on gray-scale reconstruction, that requires multiple scans of the input image using priority queues.

A faster algorithm based on Max-Trees was proposed by Salembier et al. [19]. Max-Trees are rooted trees where each node represents a connected, flat level component. Regional maxima are represented by the leaves of the tree. Filtering is then only a matter of removing all nodes of size less than λ. It is easily generalized for a wide range of morphological filters, such as morphological thinnings [20]. Wilkinson et al. [5] developed a parallel version of this algorithm. They split the input image in $n \geq t$ sub-images, where t is the number of available hardware threads. For each sub-image, a Max-Tree is computed in parallel. In a subsequent, sequential routine, the independently computed trees are merged and, finally, filtered.

Meijster and Wilkinson [4] proposed a region growing algorithm to compute morphological area openings based on Tarjan's union-find data structure [6]. They extended their algorithm to compute the size distribution of elements on an image, called area granulometry [21]. The parallel algorithms developed in this thesis extend this union-find based algorithm. The remainder of this section contains a detailed description of the algorithm by Meijster and Wilkinson [4].

A pixel i at the coordinates (x, y) is represented as single integer value, computed by $i = w \cdot y + x$ where w is the width of the input image [4]. Connected components are represented by a disjoint-tree structure, where the root of each tree is represented by the last visited pixel. An input image is represented using two arrays:

- `value`, which stores an integer representation of the gray-scale value of each pixel. During the main loop, this array is read only.
- `parent`, which represents the index of the parent pixel in the union-find data structure if `parent[i]` ≥ 0, or the size of the tree of which `i` is the root, if `parent[i]` < 0.

For simplicity, both arrays are also accessible through equally named functions. Using negative integers to represent tree size instead of parent index, allows us to minimize memory usage [4]. We are furthermore given the following functions:

```
public void union(final int n, final int c, final int lambda) {
  final int nr = find(n);
  final int cr = find(c);

  // Already in the same set.
  if (nr == cr)
    return;

  // Join level and peak components.
  if (value(nr) == value(c) || -parent[nr] < lambda) {
    parent[cr] += parent[nr];
    parent[nr] = cr;
  } else {
    // De-activate tree.
    parent[cr] = -lambda;
  }
}
```

Figure 2.2: The `union` function for conditional region growing.

```
public void uniteNeighbors(final int c, final int lambda) {
  for (final int n : neighbors(c)) {
    if (value(c) <= value(n))
      union(n, c, lambda);
  }
}
```

Figure 2.3: The `uniteNeighbors` function that calls `union`.

- `union(n, c, lambda)` unites the trees containing the pixels at `n` and `c` for a given size parameter `lambda` (see figure 2.2) [4].
- `find(x)` returns the root index of the tree containing the pixel at `x` and compresses the path between `x` and the root [4, 6].
- `neighbors(x)` returns all indices of the pixels adjacent to `x`, assuming 8-connectivity.
- `uniteNeighbors(x, lambda)` unites the pixel at `x` with all its valid neighbors.

To compute area opening, we must first identify regional maxima on the image [2]. The algorithm by Meijster and Wilkinson [4] does this by sorting image pixels by gray-scale value. The so sorted pixel indices are stored in an additional array `sorted`. Sorting can be performed in linear time using counting-sort [4].

The next step is to let connected components of pixels grow conditionally, if, after the definition of area opening, neighboring connected components belong to the same connected component. The function `uniteNeighbors(c, lambda)` unites a pixel with all its valid neighbors and is shown in detail in figure 2.3.

If `c` is at level with `n`, the pixel pair is valid. This is essentially flat component labeling [4]. Moreover, a pixel is allowed to join a set “upwards” in gray-level hierarchy, so to unite with a pixel of higher gray-value: this is the first part of growing connected components. The gray-level sorted pixels are iterated in decreasing gray-value order. This is the main loop of union-find based area opening [4]:

```
// Sort using counting sort.
final int[] sorted = CountingSort.sort(0, image().size(), image());

// Filter pixels
for (int i = sorted.length - 1; i >= 0; --i) {
  final int c = sorted[i];
  uniteNeighbors(c, lambda);
}
```

However, `union` does not unite disjoint trees unconditionally. Instead, it allows connected components only to be united, if either the root of `n` is at level with `c`, or, if the area of the connected component containing `n`, is less than `lambda`. If none of these conditions are fulfilled, `c` is deactivated: its size is simply set to `lambda` and no component may exceed `lambda`, unless it is a level-component (i.e. all pixels of the component are at the same gray-level) [4]. If the trees are united, the last visited pixel `c` becomes the new root. Thereby, the darkest pixel of the connected component is always the root of the tree. Building paths in this fashion directly contradicts the idea of ranking and swapping trees, in order to maintain short paths [6]. Meijster and Wilkinson [4] leave this contradiction uncommented.

This algorithm a simplified version of the one given by Meijster and Wilkinson [4]. Their original algorithm performs on-the-fly initialization of the disjoint set structure, which is possible because, in a sequential program, counting sort is stable w.r.t. the prior order of elements. This implies that it is enough to find the root of `n` during `union`, as `c` has not yet been initialized as a singleton. Every pixel's singleton is initialized when `uniteNeighbors` is called on it. The original condition (compared to figure 2.3), to determine whether a pixel should be united with its neighbors, is [4]:

```
value(c) <= value(n) || (value(c) == value(n) && n < c)
```

This structure gives the linear code advantages, which are hard to achieve in parallel algorithms. For the remainder of this thesis, I will only consider the simplified version of the algorithm, so that the order of level pixels is neglected. The detailed implementation of the simplified `union` algorithm is depicted in figure 2.2.

Always making the last visited, and thereby darkest, pixel root of its set enables us to use a simple resolving step, that finalizes the algorithm. The list of sorted pixels is iterated in reverse order and each pixel is assigned the gray-scale value of its root. Thereby, we filter regional maxima from the image [4]:

```
// Resolve gray-values.
for (final int s : sorted) {
  if (parent(s) > 0) // Only update child-nodes.
    image().pixel[s] = value(find(s));
}
```

This final step will not be part of further discussion, as it can easily be parallelized. We are mainly interested in building the correct union-find structure to represent connected components.

Chapter 3

Shared Union-Find Data Structures

This chapter describes two different algorithms for a shared union-find [6] data structure. The first one is the wait-free union-find algorithm by Anderson and Woll [8]. The other one relies on fine-grained locking per disjoint tree [7].

The following sections contain many different figures, that convey variants of parallel union-find. In order to explain the algorithms thoroughly, I believe that it is easiest for the reader to look at code examples. This is especially true, because the many variants of the union-find algorithm are very similar and the differences are very subtle, if explained in plain English. I deliberately omit introducing the original union-find by Tarjan [6] in favor of focusing on the concurrent algorithms.

3.1 A Wait-Free Implementation

As described in section 1.2, wait-free concurrent algorithms have a number of desirable properties, compared to blocking algorithms. Therefore, Anderson and Woll [8] developed a wait-free union-find algorithm, based on the *CAS* primitive. Their implementation uses ranking heuristics to balance trees, in order to maintain short chains between leafs and roots.

The wait-free union-find algorithm represents nodes in an array of records, called `record` and again accessible through a function of the same name. Each `Record` contains an integer representation of the rank of a tree and an (atomic) integer pointing to the index of the record's parent in the array. The implementation of `Record` is shown in figure 3.1.

The parent pointer initially points to the index of the record itself, to indicate that it is a root. It is here implemented as an atomic reference, to make implementing a wait-free path-compressing `find` easy. If `find` was not to compress paths, it could simply traverse up to the root. This would be wait-free already, as `find` does not guarantee anything but to return a node that was root at some point in time [8]. To compress paths in a wait-free manner, Anderson and Woll [8] implement their find using path-halving:

```
public int find(int x) {
  while (record(x).next() != x) {
```

```
class Record {
  final int rank;
  final AtomicInteger next;

  public Record(final int rank, final int next) {
    this.rank = rank;
    this.next = new AtomicInteger(next);
  }

  public int next() {
    return next.get();
  }
}
```

Figure 3.1: The `Record` class, as used in wait-free union-find. Maintaining the rank and the pointer to the node's parent in this object allows an atomic update of both variables using *CAS*.

```
    final int p = record(x).next();
    final int q = record(p).next();
    record(x).next.compareAndSet(p, q);
    x = q;
  }
  return x;
}
```

The parent pointer of each visited node is set to the node's grandparent, if no other thread modified the parent pointer in the meantime. If another thread interfered, the failed *CAS* is simply ignored and traversal continues.

Uniting two disjoint trees is a matter of updating the parent pointer of a node, as well as its rank. To perform this update atomically, Anderson and Woll [8] use the `Record` object associated with this node. The record of the node, which will be updated, is atomically replaced by a record pointing to the new record and maintaining a new rank. This update is performed by the auxiliary function `updateRoot`. It terminates early, if another thread modified the soon-to-be child node at `x`. Otherwise, it simply returns to the caller, whether the update has been successful:

```
boolean updateRoot(int x, int oldrank, int y, int newrank) {
  final Record r = record(x);
  if (r.next() != x || r.rank != oldrank)
    return false;

  final Record n = new Record(y, newrank);
  return record.compareAndSet(x, r, n);
}
```

While this would already be enough to implement a wait-free `union`, Anderson and Woll [8] acknowledge, that this can result in long chains, if multiple threads unite trees, without the rank of the root being incremented successfully. To counter this, they add a call to `setRoot`, which performs a `find`-like traversal and compression. Finally, `setRoot` tries to update the rank of the root by one atomically, using `updateRoot`:

```
final void setRoot(final int x) {
  int y = x;
```

```java
public void union(int x, int y) {
  while (true) {
    x = find(x);
    y = find(y);
    if (x == y) // Already in the same set.
      return;

    int rx = record(x).rank;
    int ry = record(y).rank;
    if (rx > ry) { // Maintain short chains.
      // Swap x, y and rx, ry...
    }

    if (!updateRoot(x, rx, y, rx)) { // Try to update root.
      continue; // Re-try if update failed.
    }

    if (rx == ry) { // Increase rank to balance tree.
      updateRoot(y, ry, y, ry + 1);
    }
    setRoot(x);
    return;
  }
}
```

Figure 3.2: Wait-free implementation of `union`. Anderson and Woll [8] acknowledge, that this implementation of concurrent `union` can result in long chains in the disjoint tree structure.

```java
  while (record(y).next() != y) {
    final int p = record(y).next();
    final int q = record(p).next();
    record.get(y).next.compareAndSet(p, q);
    y = q;
  }
  updateRoot(y, record(x).rank, y, record(x).rank + 1);
}
```

The `union` function, in its entire form, is depicted in figure 3.2.

3.1.1 Adaption to Area Opening

A major flaw of Anderson and Woll's wait-free union-find algorithm is the spuriously occurring failure of updating ranks, as well as the temporary inconsistencies between node updates [7, 8]. Updating the size of a root node, together with updating references to it, is key to label connected components correctly. Otherwise, another thread could grow a component, while another is in-between updating the root pointer and updating the new root's size, which are independent from each other. This can result in uniting pixel sets based on wrong priorities. Using `Record` as a container to update rank and parent pointer atomically, works only for updating both values on the same node and does therefore not provide a solution to the problem.

This is a manifestation of the missing composition capabilities of synchronization primitives, as already mentioned in section 1.2.2. Herlihy and Shavit [10] give a proof, that there is no wait-free implementation to assign values to

```
public int find(int c) {
  while (true) {
    final int p = parent[c].atomicWeakGet();
    if (p < 0)  // Not being a root is invariant.
      break;
    final int q = parent[p].atomicWeakGet();
    if (q >= 0) // Not being a root is invariant.
      parent[c].atomicCompareAndSet(p, q);
    c = p;
  }
  return c;
}
```

Figure 3.3: Wait-free `find` for area opening. *multiverse*'s `atomicWeakGet` corresponds to `get` on Java's atomic types; `atomicCompareAndSet` corresponds to `compareAndSet`. As soon as the `parent` value of a node is equal or below zero, it will not be set to below zero again.

multiple, independent fields (i.e. an (m, n)-assignment) consistently.

This means that the wait-free union-find algorithm cannot be directly applied to our problem.

3.1.2 STM Variant

To update two independent nodes atomically when computing area opening, we can resort to an *STM* version of Anderson and Woll's algorithm. The `find` routine stays entirely wait-free and still performs path compression by path-halving, as shown in figure 3.3. It is now mandatory to avoid compressing the path to a negative parent pointer, as that would result in removing a sub-tree from its root.

The concept of records is not required any longer. Also, ranking and the according rank updates are removed, in order to model the correct relation of pixels to each other [4]. This implies also the removal of `updateRoot` and `setRoot`.

The wait-free `union`'s while-pattern is still required by the algorithm, if we use negative values to model component size, while compressing the paths using a wait-free `find`. Otherwise, we can run into complications when checking, whether the negated parent value (i.e. the size) is less than `lambda` (see figure 3.4).

Moreover, we can implement conditional deactivation of not-united root nodes. If a to-be-deactivated root already exceeds `lambda`, it does not need to be deactivated explicitly. Short-circuiting this operation saves overhead due to unnecessary modifications by the transaction, which would otherwise come at the considerable cost of copying and modifying the variable and then committing the changes [10].

To compare the performance of using an explicit while-loop , I implemented a general version of Anderson and Woll's union-find with *STM* modifications, as well as a simple *STM* implementation. The latter simply wraps the sequential `union` in a single transaction.

```java
public void union(final int n, final int c, final int lambda) {
  atomic(new TxnVoidCallable() { // Wrap the atomic code.
    public void call(final Txn txn) throws Exception {
      while (true) {
        final int nr = find(n);
        final int cr = find(c);
        if (nr == cr) // Already in the same set.
          return;

        // Read states consistently.
        final int ns = parent[nr].get();
        final int cs = parent[cr].get();

        // If both elements are roots, unite them.
        if (ns < 0 && cs < 0) {
          if (value(nr) == value(c) || -ns < lambda) {
            parent[cr].set(ns + cs); // Grow component size.
            parent[nr].set(cr); // Unite elements.
          } else if (-cs < lambda){ // Disable only if still active.
            parent[cr].set(-lambda);
          }
          return;
        }
      }
    }
  });
}
```

Figure 3.4: STM `union` for area opening. `TxnVoidCallable` is a *multiverse* wrapper class, similar to the `Callable` interface. Note that the while-loop is still required, in order to catch modifications by wait-free `find`.

```
public void union(final int x, final int y) {
  while (true) {
    int xr = find(x);
    int yr = find(y);
    if (xr == yr) // Already in the same set.
      return;
    lock(xr, yr);

    // Proceed if locked nodes are still roots.
    if (next[xr] == xr && next[yr] == yr) {
      if (rank[xr] > rank[yr]) { // Maintain short chains.
        // Swap xr and yr...
      }
      next[xr] = yr; // Unite nodes.
      if (rank[xr] == rank[yr])
        ++rank[yr]; // Increase rank to balance tree.

      // Compress paths while having lock.
      compress(x, yr);
      compress(y, yr);

      unlock(xr, yr); // Release lock before returning.
      return;
    }
    unlock(xr, yr); // Release lock before retrying.
  }
}
```

Figure 3.5: Optimistic locking implementation of `union`. The algorithm performs `find`, until it has acquired the locks for the actual roots.

3.2 An Optimistic Implementation

Typically, locks are used globally for data structures. This blocks threads from performing changes, while another thread is modifying some part of the data. While easier to implement, this leads to high contention, as the progress of all threads *not* holding the global lock is blocked, even if their modifications would not collide with those of the active thread. This is not acceptable for efficient parallel algorithms.

Another approach is optimistic, or fine-grained, locking of resources. A thread only acquires locks for those parts of a data structure, that it wants to modify consistently. Writing to an array at different indices is probably the best example for a data structure, that invites such a locking scheme, as each index of the array can get assigned a unique lock.

In his thesis, Berman [7] developed a union-find algorithm based on optimistic locking. He claims that this approach outperforms the wait-free union-find implementation by Anderson and Woll [8]. In his implementation, the disjoint sets are represented by three arrays of the same length: one array `next`, that, for each node, holds a pointer to either the parent index or to itself, if it is root. Another array `rank`, that maintains the rank of each node, and finally an array `lock`, that maintains a lock for each node in the disjoint set structure [7].

To update the relation of two trees consistently, Berman [7] uses a (not further described) function `lock_roots` that locks two trees in the forest of disjoint trees by acquiring the locks only for the respective root nodes. Java does not provide a two-object locking mechanism. Therefore, the Java implementation uses a simple double-lock algorithm on the `lock` array, which is of type `ReentrantLock[]`. This algorithm is based on a simple lock algorithm given by Herlihy and Shavit [10]. Unlocking both trees, in case acquiring both locks failed, ensures that no deadlocking occurs, in case a thread only successfully acquired one lock:

```
void lock(final int a, final int b) {
  while (true) {

    // Spin while lock is unavailable.
    while (lock[a].isLocked() || lock[b].isLocked());

    // Try to acquire lock and return on success.
    if (lock[a].tryLock() && lock[b].tryLock())
      return;
    else
      unlock(a, b);
  }
}
```

Berman [7] proposes a scheme that, once it acquired the locks for two nodes, checks that these nodes are still roots. The fine-grained locking makes it possible for the nodes to change their state, while a thread waits to acquire their locks. In the Java implementation, this locking scheme is integrated into `union`, as shown in figure 3.5.

Another notable difference to the wait-free algorithm is the path compression. The optimistic algorithm compresses paths only after successfully uniting two sets, while still holding the locks [7]. This makes `find` fully wait-free, as no locks must be acquired during traversal. Path compression is implemented by re-assigning the parent pointer of each node along the path to the root's index.

3.2.1 Adaption to Area Opening

For the optimistic union-find implementation, the changes for using it in area opening are nearly negligible. Here as well, both pixel's roots must be found prior to proceeding. Updating the size of a tree at the root node and the parent pointer of another node, in order to make it to point to the new root, is not an issue in a lock-based implementation. Again, ranking heuristics are removed [4].

In order to determine whether found nodes are still roots, it is sufficient to check, whether their `parent` value is below zero. As soon as a node's `parent` value is equal to or greater than zero, it will not be re-set to below zero, since this union-find structure does not support deletion. The same goes for the wait-free implementation of `find`.

Another modification is the compression of paths (see figure 3.6), which depends on the success or failure of conditional union. If the trees have been united, both trees are compressed to the same root. Otherwise, their respective original roots are used as compression targets. The complete algorithm for optimistic `union`, adapted to area opening, is shown in figure 3.7.

```java
void compress(int bottom, final int root) {
  while (parent[bottom] >= 0) {
    final int t = parent[bottom];
    parent[bottom] = root;
    bottom = t;
  }
}
```

Figure 3.6: The `compress` function for optimistic area opening. It is only executed when the thread holds the corresponding lock for the node at `root`.

```java
public void union(final int n, final int c, final int lambda) {
  while (true) {
    int nr = find(n);
    int cr = find(c);
    if (nr == cr) // Already in the same set.
      return;
    lock(nr, cr);

    // If both elements are roots, unite them.
    if (parent[nr] < 0 && parent[cr] < 0) {
      if (value(nr) == value(c) || -parent[nr] < lambda) {
        parent[cr] += parent[nr];
        parent[nr] = cr;
        compress(n, cr); // Compress path to new root.
      } else {
        parent[cr] = -lambda;
        compress(n, nr); // Compress path to old root.
      }
      compress(c, cr); // Compress always unconditionally.

      // Unlock after success.
      unlock(nr, cr);
      return;
    }

    // Unlock and re-try.
    unlock(nr, cr);
  }
}
```

Figure 3.7: Optimistic `union` for area opening. The paths are compressed w.r.t. success or failure of uniting the two sets.

Chapter 4

Parallel Area Opening

In this section, I will describe five algorithms that compute the area opening for a given input image and a minimum area λ, in parallel. All algorithms are implemented, so that they can use either variant of shared union-find as presented in chapter 3.

The designs of the parallel area opening algorithms vary to different degrees. The first important notion is that of work items. For images, those are typically either pixels, pixel-lines or multi-line sub-images. Furthermore, there are essentially three types of filtering algorithms:

No sorting Instead of sorting, the pixel array is scanned repeatedly.

Thread-local sorting Subsets of the pixel array are sorted thread-locally.

Shared sorting Pixels are sorted in a lock-free fashion and sorting is a joint effort of all threads.

There is one important property, that may not be violated by any algorithm: pixels must be scanned in decreasing gray-scale order, in order to guarantee, that regional maxima are found implicitly [4]. We will get back to the importance of this property in chapter 5.

In the following sections, we will see implementation details for each of these algorithms. The algorithms can handle any number of gray-scale values. The resolving step, that updates the gray-scale values of each pixel, according to its position in the tree, is omitted entirely: this is a linear time operation [4] and can easily be parallelized.

For the remainder of this chapter, let k denote the number of possible gray-values per pixel, and n the number of pixels in a gray-scale image I. Moreover, let x_I and y_I denote the dimensions of an image I in x- and y-direction, so that $x_I y_I = n$. To simplify the analysis of the algorithms, let us assume that `uniteNeighbors` amortizes to $O(n)$ for an image I of size n. We will only use formal run-time complexity to underline, how differently the parallel algorithms behave, even though they share the same formal complexity.

```
int level = image().values - 1; // Gray-values are in [k - 1, 0].
while (level >= 0) { // Repeat while not all levels processed.
  barrier.await(); // Synchronize across threads.
  for (int c = lower; c < upper; ++c) {
    if (value(c) == level) { // Filter only pixels at current level.
      uniteNeighbors(c, lambda);
    }
  }
  --level; // Decrement gray level.
}
```

Figure 4.1: Simple sub-image filtering algorithm. The interval of pixels assigned to a thread is denoted as the integers `lower` and `upper`. Threads synchronize on gray-level decrease.

4.1 Non-Sorting Variants

A key idea of the algorithm presented by Meijster and Wilkinson [4], is to sort pixels in linear time by gray-value. Originally, Vincent proposed an algorithm, that scans the image repeatedly [2]. In this section, we will take a look at algorithms that neglect the possibility of linear-time sorting of pixels and instead re-scan the image, until all pixels have been filtered. The main purpose is didactic: without sorting, the algorithms are easier to implement and we can focus on key ideas. In section 4.2, we will see algorithms that extend the ones presented here, by sorting the input. Nevertheless, algorithms from this section will be included in the performance evaluation in chapter 6.

4.1.1 Sub-Image Filtering

A typical pattern to parallelize image analysis algorithms, is to simply split up the image in as many sub-images as there are hardware threads available. This requires, that the operation on each pixel is independent. This approach works well for image convolutions, such as the Sobel filter [22].

According to these requirements, conditional region-growing, as performed during area opening, is not parallelizable by splitting the image: correct growing must be ensured across sub-image borders. Using a shared and re-entrant data structure, however, like our concurrent union-find variants, circumvents this requirement of independence. While each operation can be performed independently, the result of each union operation is shared across threads implicitly through side-effects.

Following this, we can implement a simple area opening algorithm on sub-images. Each hardware thread is assigned an upper and a lower index on the pixel array of the image and, for each gray-scale level, iterates repeatedly over all indices, from the lower to the exclusive upper bound, uniting all valid pixels for the current gray-scale level. All threads are synchronized on the current gray-level, which is maintained locally by each thread. Synchronization is performed through a barrier: on reaching the barrier, a thread may not proceed, until all other threads also reach it.

If executed on only a single hardware-thread, the complexity of non-sorting sub-image area-opening is $O(kn)$. This algorithm is detailed in figure 4.1.

```
while (true) {
  final Queues<Long> local = queues.get();
  if (local.level < 0) // All levels have been visited.
    break;

  // Get next pixel from upper buffer.
  final Long c = local.upper.poll();
  if (c == null) {
    barrier.await(); // Synchronize threads.
    local.swapAndDecrement(queues); // Swap if no pixels left.
  } else {
    // Filter pixel only if it is at current level.
    if (value(c.intValue()) == local.level) {
      uniteNeighbors(c.intValue(), lambda);
    } else {
      local.lower.add(c); // Enqueue pixel to lower buffer.
    }
  }
}
```

Figure 4.2: Pixel queue filtering algorithm. Pixels are re-added to the `lower` queue, if their gray-level is less than the current level.

4.1.2 Pixel Queues

The naive sub-image filtering algorithm, as shown in section 4.1.1, always runs in $O(kn)$: all pixels need to be scanned again for each gray-level of the image. If we want to take advantage of the fact, that each pixel only needs to be filtered explicitly once, we need a way to remove already filtered pixels from the work-list.

A way to represent not yet filtered images is a concurrent queue data structure, for instance a Michael and Scott [13] queue (see also section 1.2.2) or a lock-free, bounded array queue (see implementation details of this data structure in appendix A.1). At initialization time, all pixels are added to the queue. This can be done in parallel by all threads. Then, each thread removes the next pixel from the tip of the queue and either filters the pixel, if it is at the current gray-level, or adds it to the end of the queue again.

This approach has two issues: first, we would need to keep track of how many pixels we already have filtered and how many pixels we still expect to be in the queue, in order to decrease the current gray-level correctly. Secondly (and actually less importantly), we would encounter many conflicts during concurrently removing and adding pixels.

To simplify the approach, we can instead use two queues, wrapped in a container class `Queues` (see appendix A.2 for details). It maintains one active, or upper, queue, that contains all pixel indices, that still need to be filtered for the current level, and one inactive, or lower, queue, to which all pixels that have not been filtered yet, are added. Now, the current gray-level can safely be decreased when the active queue is empty. The auxiliary function `swapAndDecrement` decrements the current gray-level and swaps active and inactive queues atomically. The implementation of `swapAndDecrement` is detailed in appendix A.2.

Analytically, this is still a linear-time operation. Assuming, that both adding and removing a pixel from a queue happens in constant time, the worst-case has

```
// Pixels sorted after gray-value level.
final int[] sorted = CountingSort.sort(lower, upper, image());
int level = image().values - 1; // Gray-values are in [k - 1, 0].
int p = sorted.length - 1; // Indices are in [n - 1, 0].
int c = sorted[p];

// Repeat while not all levels processed.
while (level >= 0) {

  // Decrement gray level if no pixels left.
  if (c < 0 || value(c) < level) {
    --level;
    barrier.await(); // Synchronize across threads.
  } else {
    uniteNeighbors(c, lambda);

    // Get next pixel or indicate all pixels are filtered.
    c = p > 0 ? sorted[--p] : -1;
  }
}
```

Figure 4.3: Sub-image filtering algorithm with thread-local sorting. The sub-images are sorted prior to filtering in order to avoid multiple scans of the image.

a complexity of $O(n + kn) = O(kn)$. The algorithm is detailed in figure 4.2.

4.2 Thread-local Sorting Variants

In this section, we will consider algorithms, that sort pixels in a thread-local manner. That means, that each thread executes a sequential sorting algorithm on some portion of pixels. The algorithms in this section extend the algorithms presented in section 4.1.

4.2.1 Sub-Image Filtering

The naive image splitting algorithm (section 4.1.1), does not make any use of the pixel properties, that allow linear-time sorting. To improve the algorithm's run-time complexity, we now first let each thread sort the pixels in its assigned interval and thereby remove the need for multiple scans of the image. Threads still synchronize on gray-level decrease. This results in a run-time complexity of $O(k + n + n) = O(k + n)$.

In the Java implementation (figure 4.3), I use counting sort, as suggested by Meijster and Wilkinson [4]. The algorithm terminates, if the current gray-level is below zero, i.e. all gray-levels have been filtered. Otherwise, we need to check, whether the current pixel's value is equal to the current gray-level and only filter it, if this is the case. If all pixels in this interval have been filtered, we assign -1 to the current pixel pointer `c`. If `c < 0`, we can directly jump to decrementing the thread-local level pointer and to synchronizing threads.

```
while (true) {
  final Queues<Line> local = queues.get();
  if (local.level < 0) // All levels have been visited.
    break;

  // Get next line from upper buffer.
  final Line l = local.upper.poll();
  if (l == null) {
    barrier.await(); // Synchronize across threads.
    local.swapAndDecrement(queues); // Swap if no lines left.
  } else {

    // Filter all pixels on current level.
    while (value(l.current()) == local.level) {
      final int c = l.current();
      uniteNeighbors(c, lambda);
      if (!l.advance()) // Advance pixel pointer if possible.
        break;
    }
    if (l.workLeft()) // Enqueue line to lower queue if work left.
      local.lower.add(l);
  }
}
```

Figure 4.4: Line queue filter algorithm with thread-local per-line sorting. The pixels of each line are sorted via counting sort. Lines are shared across threads, in order to avoid starvation. The `Line` interface provides transparent access to the sorted pixels and maintains an internal pointer to the current pixel.

4.2.2 Line Queues

The queue-based approach from section 4.1.2 to parallel area opening can easily be extended to the next greater unit of work-items in images. A class `Line` represents a line, containing an array of pixel indices (see appendix A.3). It maintains an internal pointer to the current pixel index. The pointer can be moved transparently by a call to `advance`, which returns a Boolean indicating the success of the operation. It will fail, if there is no work left (i.e. all pixels have been filtered), which can also be queried directly using `workLeft` (see the full implementation in appendix A.3).

All threads join in the initialization phase, where each processes a line at a time thread-locally. Each line is sorted separately, using again counting sort. After the initialization phase, the double queue principle, as shown in section 4.1.2, is used. The Java implementation is displayed in figure 4.4.

The advantage of this algorithm is two-fold. First, removing the requirement for re-scanning the image, or parts of it, is removed entirely. All work-items, pixels and lines, are removed from the queues as soon as they have been filtered. Secondly, this approach avoids a problem that can hurt the performance of sub-image filtering badly, causing extreme starvation. Consider the case of a gray-scale gradient in the image's y-direction. Using a sub-image filtering approach, this case essentially forces the threads to work in sequence, waiting idly for the one thread that is lucky to be assigned the sub-image, which holds all pixels of the current gray-level. This is not longer possible with the line-queues algorithm, as threads are not assigned a fixed portion of work, but share all data to support

```
// sorted is of type LinearPixelSort.
sorted.sort(); // Sort pixels in parallel.
int level = image().values - 1; // Gray-values are in [k - 1, 0].
while (level >= 0) {
  barrier.await(); // Synchronize across threads.

  // Get initial pixel for current level.
  Integer c = sorted.next(level);
  while (c != null) {
    uniteNeighbors(c, lambda);
    c = sorted.next(level);
  }
  --level;
}
```

Figure 4.5: Per-level blocking filter algorithm with shared sorting. The threads are again synchronized on gray-level decrease.

progress actively.

The worst-case run-time complexity of the line-queues algorithm bases on the initialization-step, which is (sequentially) of $O(y_I(k+x_I)) = O(y_I k+n)$. An additional cost is, of course, the union of pixels with their neighbors and the re-adding of each line of $O(y_I+n)$, so overall we get $O(y_I k+y_I+2n) = O(y_I k+n)$.

4.3 Shared Sorting Variants

The last variant is to share the work of sorting the pixels of the image between all threads. Each thread iterates, in parallel, over the shared array of sorted pixels and performs `uniteNeighbors`, synchronizing again on gray-scale level decrease.

There are two linear-time sorting algorithms that can be parallelized easily. The first one is the already mentioned counting-sort. To obtain a fully parallel variant, we cannot rely on partitioning the input data. Instead, the implementation here uses atomic integers and barriers to sort in parallel. One side-effect is, that the target array cannot be re-used, causing an increase of memory usage by $O(n)$. Also, the parallel sorting algorithm is not wait-free, since we need to synchronize between counting and insertion steps.

The second algorithm is bucket sorting by maintaining an array of re-entrant queues. The array is of length k, with each index representing the subset of pixels at the indexes gray-scale level. To not cause a memory usage of $O(kn)$ (k levels with possibly n pixels each), it is mandatory to use an unbounded data structure, in order to maintain those subsets. I therefore chose to use Michael-Scott queues in order to keep memory usage low.

Both algorithms are implemented in a data structure, that implements the `LinearPixelSort` interface. A thread can seamlessly start sorting by calling `sort` on a sub-type of `LinearPixelSort`, since the lock-free structure of both algorithms requires no knowledge of the number of threads participating in parallel sorting. Calling `next`, with a gray-scale level as argument, returns the next pixel at the given level, or `null`, if there are no pixels left at this level.

Shared sorting is followed by gray-level wise filtering of pixels. The `level` variable is thread-local, to avoid additional contention. Each thread then retrieves the next pixel from the current level's queue. If retrieval was successful, i.e. the queue at the current level is not yet empty, the pixel is united with its valid neighbors. If not, the local level is decreased and the threads synchronize at the barrier. The full Java implementation of the filtering part of the algorithm is given in figure 4.5. If executed sequentially, its run-time complexity is $O(k + n + n) = O(k + n)$

Chapter 5

Correctness

This chapter is devoted to the correctness of the algorithms presented in chapter 4. An inherent problem of parallel and concurrent programs is the non-determinism of interleavings between independent threads. Because of this, standard unit-testing of parallel code, as done with sequential code, is not enough to ensure the absence of errors.

Instead, there exist a number of formal methods to verify parallel programs. One of the first formal frameworks to reason about correctness in concurrent programs developed, was *Communicating Sequential Programs* (CSP) [23]. CSP allows manual reasoning and proving of programs, but there also exist digital implementations [24].

Another, more practical approach is model checking [9]. The advantage of model checking is that it can easily be fully automated. However, model checking suffers from analyzing huge state-spaces. In this chapter, I will focus on showing the absence of errors in the parallel area opening algorithms through model checking by explicit state enumeration using *Java Pathfinder* [9]. Additionally, I will briefly touch upon the possibilities of unit-testing parallel area opening.

5.1 Experimental Correctness Verification

In contemporary software engineering, unit-testing is an often used tool to assure that independent parts of a program behave correctly. Input values must be provided manually, which means that testing is basically experimental correctness verification and by no means exhaustive – this is true for both sequential and concurrent programs. Nevertheless, it is a good starting point in practice. Especially parallel algorithms can, as a first step, be tested on a single thread to assure conceptual correctness.

Due to the already pointed out non-determinism of thread interleavings – caused either by true hardware parallelism or by the intransparent heuristics of a thread scheduler – testing of truly parallel algorithms is practically impossible. Errors might seem to occur randomly, or so spuriously that they are not found during testing [12]. To increase the chance to hit erroneous execution paths, tests can of course be repeated automatically with a huge variety of input data. Still, this is not exhaustive. Thread scheduling is non-deterministic, but not entirely random. Executing a concurrent program multiple times does not guarantee to execute different thread interleavings. Therefore, more elaborate methods are required, in order to verify the correctness of parallel programs.

For development purposes, I implemented an experimental black-box testing suite, which asserts a single post-condition for the parallel algorithms. Recall from section 2.1 that γ_λ^a denotes area opening and f a projection from a subset of $\mathbb{R}^2$ into $\mathbb{R}^*$. Let, moreover, ρ_λ^a denote any parallel variant of area opening. We assert that:

$$\gamma_\lambda^a(f) = \rho_\lambda^a(f) \tag{5.1}$$

Again, due to the non-determinism of thread scheduling, this assertion might only fail occasionally for incorrect programs.

5.2 Java Pathfinder

Java Pathfinder (JPF) is an explicit state enumerator and model checker [9]. JPF has been used to verify concurrent real-life *NASA* programs and has discovered serious bugs [25]. The model checker uses search heuristics to execute all relevant states of a program. It reports erroneous operations, such as accessing fields on `null` objects or division by zero. Additionally, programs can be annotated with pre- and post-conditions, as well as class invariants, to model more complex correctness properties.

JPF can be seen as a drop-in JVM replacement and has been especially designed to uncover concurrency related bugs [9]. Any Java program can be run directly in JPF, with the exception of programs that use native code. JPF aware programs (those that are annotated with properties) benefit most from model checking and, theoretically, no code-changes are required when moving the model checked target code to a production system running a typical JVM.

5.2.1 Symbolic Execution

Symbolic execution is a method of model checking that computes values of variables, in order to execute each path of the program once [26]. Thereby, it is not necessary to execute the system under test (SUT) with many different input variables to trigger each path once, possibly re-taking already executed and checked program paths. Symbolic variables are assigned as needed, to trigger all possible program paths. This is trivial for data types like Boolean, as there are only two possible values to assign. In the case of numeric data types, JPF uses linear programming solvers to compute appropriate values, depending on the program's control flow.

The state space of the area opening algorithm is dependent on the image structure. If we focus on the possible number of orderings of the pixels of an image containing n pixels, times the valid range of values for $\lambda \in [0, n]$, the input space becomes $O(nn!)$. Symbolic execution can effectively reduce the checking time, by triggering the execution of each combination of `union` and `uniteNeighbors` once instead of using real pixel values

5.2.2 Current State of Implementation

JPF, in its most recent version (*jpf-core*, commit 1155:1f82eee5f139), provides automatic state exploration and performs the most basic correctness checks. Further modules allow to annotate Java code with pre- and post-conditions and class invariants (*jpf-aprop*, commit 64:0f5c04466e1c) and model checking with symbolic values (*jpf-symbc*, commit 583:b1bdfb401e9f)[1].

Annotations are limited to accessing class fields and method parameters. Functions cannot be called from within annotations, since functions might have side effects. JPF has no notion of *pure* functions. (There is a `@SandBox` annotation, but it has no effect in this regard.) In order to model check parallel area opening, we require access to a number of auxiliary functions, to find the root of a connected component, to get a pixel's gray-value and so on (see section 5.3.1). One method to implement more complex properties through a `Property` interface is to use the `satisfies` directive. The JPF implementation to check properties in this way seems to be buggy. Another major bug in the JPF implementation is an error, that occurs when using the Java `Logger` class. This makes it currently impossible to verify any STM variants of the algorithms using *multiverse*. Even though fixing bugs in JPF is out of scope of this project and the verification of STM algorithms is therefore identified as future work, I performed some minor changes, in order to correct JPF's error reporting.

JPF has a vast number of configurable options. There is an active community around JPF and it is subject to active research [9, 25, 26]. The configuration process is very trial-and-error driven. JPF is by no means a light-weight tool for "drive-by" software verification.

5.3 Verifying Area Opening

5.3.1 Correctness Properties

In order to verify that all algorithms behave the same, it is necessary to define correctness properties, which are universal to the family of union-find based area opening algorithms. The following properties are universal and all algorithms presented in chapter 4 must respect these. We need to verify that (1) the pixels are filtered in the correct order and (2) that the connected components of two pixels are always united with respect to the correct conditions.

Because we do not want to alter the program state when executing assertions, we need to introduce a pure version of `find` that does not perform path compression. The pure implementation of `find` is recursive and does not have any side-effects:

```
int find(final int c) {
  if (parent(c) < 0)
    return c;
  return find(parent(c));
}
```

All invocations of `find` in the remainder of this chapter refer to the pure, recursive implementation.

[1]All JPF modules have been checked out on the 27.04.2014.

Area Opening

As already mentioned in chapter 4, in order to guarantee implicit scanning of the regional maxima, the algorithms need to filter the pixels in *decreasing gray-scale order* [4]. This is an operational correctness property. In the sequential algorithm, this can be ensured by simply sorting pixels by gray-scale value and then iterating the pixel array from bright to dark pixels.

Obviously, this is harder to achieve in parallel programs, since we need to ensure that all threads synchronize on the decrease of the current gray-level value. Otherwise, it might occasionally occur that one thread proceeds to scan a pixel of lower gray-value and unites it with a connected component, that thereby reaches a size of λ, whereas the component should have instead been united with a different pixel of higher gray-level first and thereby would already have reached λ. This pattern can occur, because of non-deterministic delays of threads between retrieving a pixel and calling `uniteNeighbors` on it and results in wrongly grown regions.

In order to assure that pixels are scanned in correct order across independent threads, we can formulate a simple invariant for `uniteNeighbors`: its collected input over program execution must be a function of time t to gray-value, $g : \mathbb{R}^* \to \mathbb{R}^*$, which is *monotonically decreasing*, so that $g(t) \leq g(t-1)$.

To use another, more imperative model, let Q be the set of all already filtered pixels and P of the pixels yet to process. Initially, $Q = \emptyset$ and $P = I$, where I is the set of all pixels in the image. Let moreover $f : I \to \mathbb{R}^*$ denote the gray-value of a pixels (recall this definition from chapter 2.1). After each invocation of `uniteNeighbors`, $Q \leftarrow Q \cup \{c\}$ and $P \leftarrow P \setminus \{c\}$ are assigned atomically:

$$J : \forall q \in Q, \forall p \in P, f(q) \geq f(p) \tag{5.2}$$

For an actual implementation of this check, it is more convenient to only maintain a pointer to the last encountered gray-level, θ, and assign $\theta \leftarrow f(c)$ after each invocation of `uniteNeighbors`. The invariant in equation 5.2, as well as the property of monotonic decrease, are Markov chains: the state is only dependent on the last encountered pixel, as it is the *infimum* of all pixels in Q w.r.t gray-level. Therefore, referencing θ in the invariant is sufficient:

$$J' : \qquad \theta \geq f(c) \tag{5.3}$$

$$\{J'\}\ \texttt{uniteNeighbors(c, lambda)}\ \{J'\} \tag{5.4}$$

Asymmetric Union-Find

Until now, we have not seen any special properties of the asymmetric area opening union-find algorithm. Instead, we will use the sequential algorithm and the fact that already united components may not be separated again in order to deduce pre- and according post-conditions for `union`.

The rules for uniting two connected components are simple and can directly be inferred from the code (see again figure 2.2). We encounter essentially three cases: (1) the pixels `n` and `c` are already in the same set, (2) `find(n)` and `find(c)` are of the same gray-level, or the size of the connected component of `n` is less than λ, or (3) `find(n)` and `find(c)` are of different gray-level and the size of the connected component of `n` exceeds λ.

For each case, consistency is mandatory. The post-conditions of `union` are dependent on the initial state. In the first case, we must assure that both pixels remain in the same set after calling `union` on them:

$$P_1 : \quad \texttt{find(n)} = \texttt{find(c)} \tag{5.5}$$
$$Q_1 : \quad \texttt{find(n)} = \texttt{find(c)} \tag{5.6}$$
$$\{P_1\}\ \texttt{union(n, c)}\ \{Q_1\} \tag{5.7}$$

In the case that both pixels fulfill the requirements for uniting them, we need to make sure that no other thread tampered with the connected components of `n` and `c` in an inconsistent way. We can do this, by verifying that the sum of the sizes of both components is the same, before and after executing `union`. Of course, the roots of both pixels are required to be the same after uniting them. Let `old` denote the old-operator, i.e. return the state of a variable before it was modified. Let now furthermore, $C_x \subseteq I$ denote the set of pixels in the current connected component of a pixel $x \in I$. This results in the following post-condition:

$$P_2 : \quad \neg P_1 \wedge (f(\texttt{find(n)}) = f(\texttt{find(c)}) \ \vee\ |C_n| < \lambda) \tag{5.8}$$
$$Q_2 : \quad \texttt{find(n)} = \texttt{find(c)} \wedge |C_n| = \texttt{old}(|C_n|) + \texttt{old}(|C_c|) \tag{5.9}$$
$$\{P_2\}\ \texttt{union(n, c)}\ \{Q_2\} \tag{5.10}$$

The third case is the direct consequence of P_2 not being fulfilled. The pixels have not been united beforehand, they are not at level and C_n exceeds λ. Therefore, C_c must be deactivated and the connected components of both pixels must remain disjoint.

$$P_3 : \quad \neg P_1 \wedge \neg P_2 \tag{5.11}$$
$$Q_3 : \quad \texttt{find(n)} \neq \texttt{find(c)} \wedge |C_c| \geq \lambda \tag{5.12}$$
$$\{P_3\}\ \texttt{union(n, c)}\ \{Q_3\} \tag{5.13}$$

Finally, the following defines the complete correctness property for asymmetric `union`:

$$P : \quad P_1 \vee P_2 \vee P_3 \tag{5.14}$$
$$Q : \quad Q_1 \vee Q_2 \vee Q_3 \tag{5.15}$$
$$\{P\}\ \texttt{union(n, c)}\ \{Q\} \tag{5.16}$$

5.3.2 Verification

Using JPF to verify a complex algorithm, such as parallel union-find based area opening, poses a number of obstacles, as already outlined in section 5.2.2. Because of the bugs in and limitations of the JPF implementation, the pre- and post-conditions defined in section 5.3.1 can not be expressed as JPF annotations. As a work-around, I implemented sub-classes of the algorithms under

Algorithm	**Acronym**
Shared sorting (bucket sort)	*block-bucket*
Shared sorting (counting sort)	*block-counting*
Sort-free pixel queues (Michael and Scott [13] queue)	*pixel-queues-msq*
Sort-free pixel queues (bounded array queue)	*pixel-queues-array*
Sort-free sub-image filtering	*split*
Non-parallel line queues (Michael and Scott [13] queue)	*line-queues-msq*
Non-parallel line queues (bounded array queue)	*line-queues-array*
Non-parallel counting sort sub-image filtering	*split-counting*
Simplified, sequential algorithm by Meijster and Wilkinson [4]	*sequential*
Sequential algorithm with bogus `union`	*bogus-union*
Sequential algorithm with bogus main loop	*bogus-loop*

Table 5.1: Algorithms and their corresponding acronyms.

test that call the methods of their super classes and internally and atomically perform Java assertions before and after. JPF can be configured to handle failed assertions as property violations. In order not to modify the state of the program during observation (for example, calling `find` during checking correctness properties can compress the path of a disjoint tree, see the beginning of section 5.3.1), I use pure auxiliary implementations of methods that perform optimization. Additionally, I prune the state space explored, using the `Verify` API [9]. This API can also be used to force JPF to execute a sequence of commands atomically – thread scheduling is simply disabled in the JVM when JPF executes atomic sequences.

Even though the model checking is performed using symbolic execution, the memory consumption and the time required for model checking *all* possible program states is enormous. This means that, in order to achieve some usable results, the time used for model checking must be limited.

This is reasonable, because of an interesting observation. In order to assure that JPF correctly finds bugs in the algorithms, I implemented a bogus `union` algorithm that always and unconditionally unites pixels if they are not already in the same set. Additionally, I added a bogus (linear) main-loop that provides the input for `uniteNeighbors`. It simply reverses the order of the sorted pixels and starts at the lowest gray-level. JPF always quickly finds the violation of the correctness properties after about two minutes (see table 5.2). This suggests that errors are found quickly, but that exploring the entire state space of a correct program is very time consuming.

All algorithms were model checked, providing the JVM with 4 gigabyte of memory (except for *bogus-union* and *bogus-loop*, which were checked on a different machine with only 2 gigabyte). The algorithms use the optimistic fine-grained locking union-find variant, except for the bogus and the linear implementation, which use the original algorithm. The parallel algorithms utilize only two independent threads to reduce the number of possible interleavings. Otherwise, the check would suffer from an immense state space explosion. The time limit is set to one hour. If JPF has model-checked an algorithm for an hour without any result, it terminates with the message “No errors detected”.

The collected results of the model-checking are shown in table 5.2. The

Algorithm	**Time**	**Depth**	**States**	**Result**
block-bucket	01:00:00	1079	23245	No errors detected
block-counting	01:00:00	1091	20426	No errors detected
pixel-queues-msq	01:00:00	3878	17847	No errors detected
pixel-queues-array	00:03:32	4146	8101	Out of memory
split	01:00:01	1113	3285	No errors detected
line-queues-msq	01:00:03	3423	9577	No errors detected
line-queues-array	01:00:03	3937	10665	No errors detected
split-counting	01:00:03	1070	3628	No errors detected
sequential	01:00:05	15	101	No errors detected
bogus-union	00:02:18	14	99	`union` violates post-condition.
bogus-loop	00:02:16	14	98	Not monotonically decreasing.

Table 5.2: Verification results for the various area opening algorithms. Time is formatted in *hours:minutes:seconds*. The algorithms are denoted after table 5.1.

algorithms are denoted with acronyms after table 5.1. All parallel algorithms pass the model check for the correctness properties, as presented in section 5.3.1, except for the obviously wrong implementations. We can see that only a few states are required to recognize a violation of the correctness properties.

Another important result is that JPF runs out of memory in case of *pixel-queues-array*. Still, JPF explores more states than for the Michael and Scott [13] queue variant, even though it proceeds not as deep into the state space. It is not able to find any property violation.

Based on these results we can conclude that model checking is not necessarily the best method to verify complex parallel algorithms. JPF has mostly been used for concurrent applications, that have a somewhat simpler control flow and a much smaller input space [9, 26]. Other formal methods, for example the application of CSP implementations [23, 24], could provide a true proof of correctness. These would, of course, require much more manual work. From this point of view, model checking is the right choice, since it can be executed automatically, because the correctness properties for union-find based area opening are universal to this algorithm family. The results underline that it is unfeasible to model check the algorithms without a time limit. Given the state space of $O(nn!)$ and the fact that JPF can, according to the results, maximally check roughly 25000 states per hour, depending on the algorithm, it would take more than just days to verify the correctness of all programs: already for an image of 3×3 pixels, we can expect an accumulated checking time of about 43 days. While it would, in principle, still be possible to do this, it is, due to a limit of resources accessible during this thesis, not possible to model check the algorithms for several weeks.

Chapter 6

Performance Evaluation

In this chapter, I will present experimental results on the performance of the introduced parallel algorithms. First, we will take a look at the set-up of the experiments and the environment (section 6.2), on which they have been conducted. Then, we will consider the performance of stand-alone parallel union-find (section 6.3) and afterwards analyze the experimental results obtained for parallel area opening (section 6.4). In each section, I will introduce the test data used and how it was generated, followed by a description, visual presentation and discussion of the obtained results.

6.1 Micro-benchmarks in Managed Languages

Proper performance measurements in managed languages, such as Java, are difficult for various reasons. Java source code is compiled to byte code in order to be executed by a JVM. The byte code is not optimized. The JVM uses Just-in-Time (JIT) compilation and either interprets the byte code, or compiles it when it deems this necessary [27].

In order to avoid spending time on compilation during the benchmark, the code is usually executed a couple of times before the performance is actually measured. Repeated execution triggers compilation of code, as it seems to the JVM that certain functions need to be executed a lot and should therefore run fast. Also, redundant code (e.g. a function's return value is never used) can be optimized away by the JVM. Therefore, one needs to make sure that extremely high performance of code is not related to the functions under test not being executed at all [27].

In order to avoid costly JIT-compilation during performance measurement, the micro-benchmark framework, which I developed for the experiments presented here, executes each algorithm three times before starting measurements in its respective configuration. Since union-find is an algorithm based on side-effects, the JVM cannot determine, whether resulting values are unused and therefore executes them.

Garbage collection is contained by calling `System.gc()` right before the benchmark. Since this call is just a heuristic for the garbage collector, it might not necessarily push all garbage collection outside of the benchmarked sections. In order to handle variances in system-load and other factors, each algorithm is executed ten times and the average duration (and other metrics) of all ten executions is returned.

6.2 Set-Up

The machine used for benchmarks runs on an *AMD Opteron^{TM}Processor 6386 SE* with 16 cores at 2800 megahertz [28]. The operating system is Ubuntu Linux 12.04 LTS with kernel version *3.2.0-60-generic*. For the Java experiments, I used OpenJDK, *Java HotSpot^{TM}* 64-Bit Server VM, build *23.25-b01* with 2 gigabyte of memory assigned.

All processor-level metrics, like accumulated cache-misses, number of issued instructions and the like, have been collected using *perf* [29] (version *3.2.55*). The micro-benchmark framework starts *perf stat* anew from within the program for each run of the benchmarked algorithm and only for the current process, so that exclusively events, which occur during the benchmark, are recorded. Note that *perf* is a Linux-only tool.

All source code, as well as the resulting raw data and scripts required to reproduce and display the results of this thesis, can be obtained at `https://bitbucket.org/fbie/wait-free-morphology`.

6.3 Union-Find

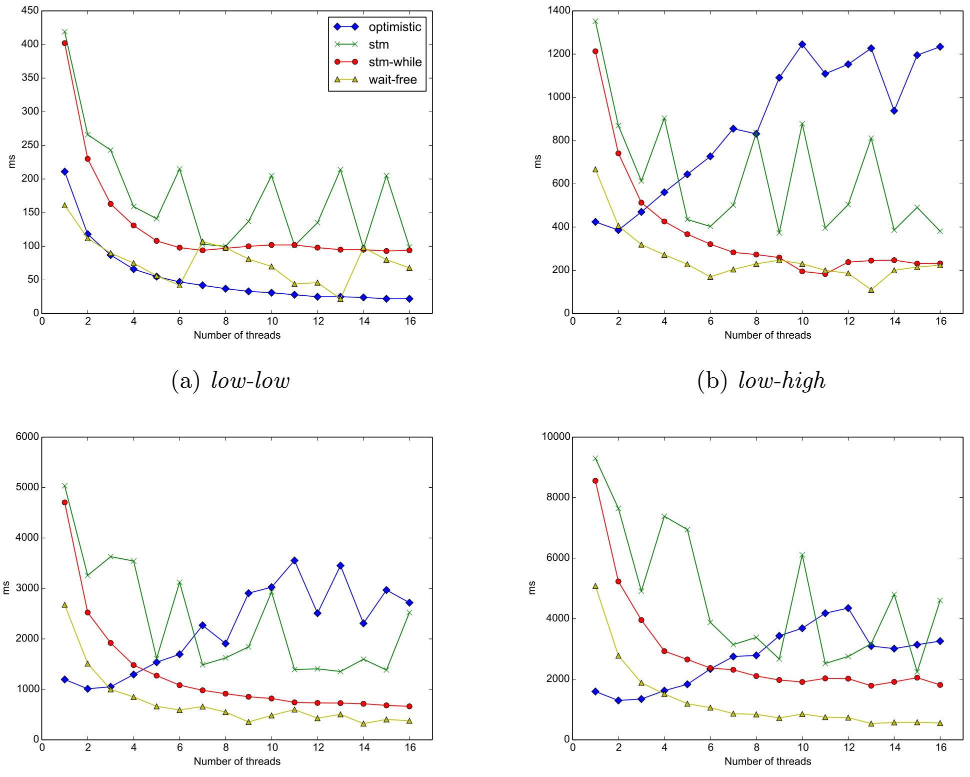
6.3.1 Input Data

The input data for parallel union-find benchmarks consists of graphs generated with the graph generator package *GTgraph* [30], which is based on the Erdös and Rényi [31] model. The Erdös and Rényi [31] model describes an undirected graph $G = (n, p)$ through a number of nodes n, and a number $0 \leq p \leq 1$, which expresses the probability that any two nodes of the graph are connected by an edge. In this model, p can be seen as a measure of connectivity of the graph. The input data has been generated in the same way as by Berman [7], in order to reproduce his results. The experiments were conducted on four different input graphs. Two graphs for $n = 5 \cdot 10^5$ with $p = 10^{-6}$ (*low-low*) and $p = 5 \cdot 10^{-6}$ (*low-high*) and additionally, two graphs for $n = 10^6$ with respective probabilities $p = 5 \cdot 10^{-6}$ (*high-low*) and $p = 10^{-5}$ (*high-high*).

During benchmarking, the union-find data structure was initialized with n nodes. Then, each thread was assigned a subset of the graph's edges, which were represented by pairs of nodes, pseudo-randomly and called `union` on each pair once. Each node in the graph is initially a singleton in the set of disjoint sets [7].

6.3.2 Results and Discussion

The timing results of the union-find benchmark are depicted in figure 6.1, where the duration for building the graph is shown in milliseconds across the number of used hardware threads. The benchmark results contradict the conclusions drawn by Berman [7]: the optimistic, fine-grained locking algorithm (denoted *optimistic*) outperforms the wait-free algorithm (*wait-free*) only in the *low-low* case (see figure 6.1a). Here, *optimistic* scales very nicely with the number of



(a) *low-low* (b) *low-high*

(c) *high-low* (d) *high-high*

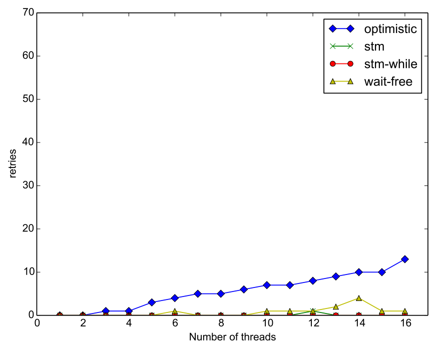
Figure 6.1: Execution time for parallel union-find in milliseconds. Note, how the performance of *optimistic* degrades with increasing connectivity of the input graph.

available hardware threads. All other graph configurations with more nodes or higher connectivity, force *optimistic* to run very slowly in comparison to the other images (figures 6.1b through 6.1d). The results for the various union-find algorithms for the latter cases are very similar. I will therefore focus on the analysis of the results shown in figure 6.1a and 6.1b in the following.

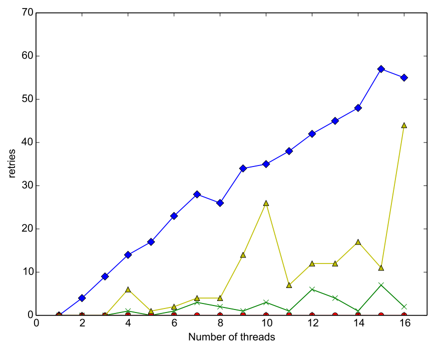
Analysis

To find the reason for this behavior, we need to investigate other metrics as well. Figure 6.2 shows the number of times the root nodes have become child nodes during `union`. Recall that this can happen between locking two root nodes or between finding roots in a wait-free fashion and then trying to unite them (see chapter 3). In the case of the STM variants (denoted as *stm* and *stm-while* for the while-loop STM variant), the number of transactional retries were counted. The curves for *optimistic* are stable across both cases (compare figures 6.2a and 6.2b) and do not exhibit any major, non-linear increase. Note, that the retries under *wait-free* and *stm* are slightly and irregularly increasing.

In figure 6.3, we see the amount of accumulated cache misses during the

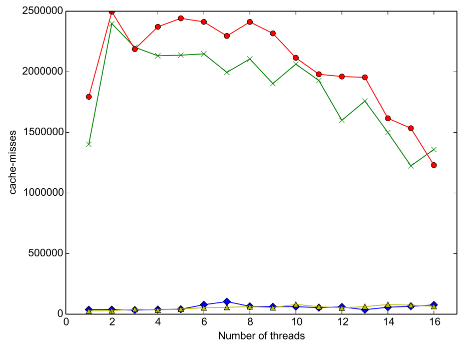


(a) *low-low*

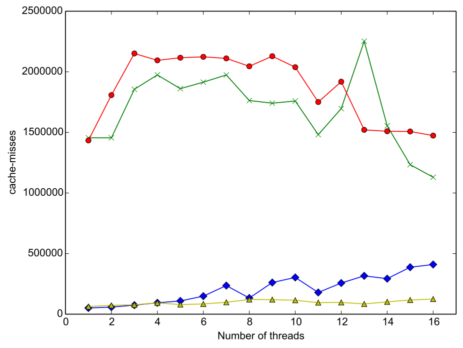


(b) *low-high*

Figure 6.2: Number of union-find retries. The recorded values are the number of STM-retries and the number of re-executions of the while-loop for *optimistic* and *wait-free*. The increase of retries is modest for increased connectivity.

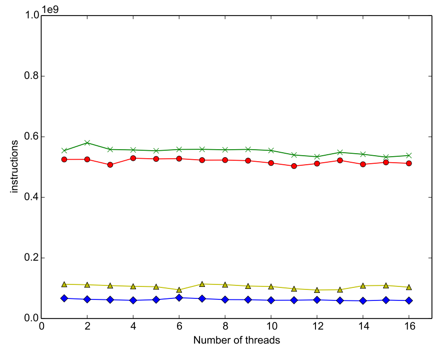


(a) *low-low*

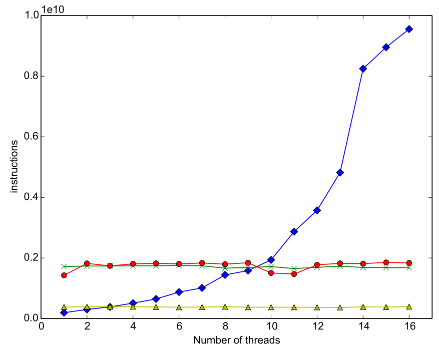


(b) *low-high*

Figure 6.3: Cache misses for parallel union-find. The recorded metrics are very similar in both cases, which suggests that cache-misses are not the reason for the exhibited performance drop.



(a) *low-low*



(b) *low-high*

Figure 6.4: Number of issued CPU instructions for parallel union-find. Note, how the number of instructions increases nearly exponentially in the case of *low-high*. Furthermore, note the different scale.

benchmark. In the optimistic case, there is a slight increase of cache misses measurable (compare figures 6.3a and 6.3b), but not to any extent that would explain such a dramatic performance drop. Also, the STM variants both consistently encounter cache misses that are roughly two orders of magnitude higher than for *optimistic*. If neither re-tries, nor cache misses are responsible for the performance loss, it is reasonable to hypothesize that the reason must simply be high lock contention due to higher connectivity of the graph and therefore fewer disjoint sets.

The number of CPU instructions issued during benchmarking, as displayed in figure 6.4, confirms this hypothesis. In case of *wait-free*, *stm* and *stm-while*, the instructions stay roughly constant across the used number of hardware threads and also across input graphs. This is not the case for the optimistic algorithm (compare figures 6.4a and 6.4b, note the different scale). The executed instructions increase nearly quadratically, which hints at intense spinning when waiting for a lock to become acquirable.

To understand, why increasing the probability by a factor of five has that much influence on the performance, we need to look at the number of connected components in the graph. According to the Erdös and Rényi [31] model, the number of connected components is dependent on np. If $np < 1$, then, with a probability close to 1, no component will be larger than $O(log(n))$. If $np > 1$, then, with equal probability, there will be a giant connected component with a size of at least $O(n^{2/3})$ [31, 32, 33].

Since for *low-low*, we have $(5 \cdot 10^5)(10^{-6}) = 0.5$, and $(5 \cdot 10^5)(5 \cdot 10^{-6}) = 2.5$ for *low-high*, the latter is likely to introduce a giant connected component to the graph. Since in optimistic union-find, all elements of a connected component share the same root, we encounter high lock contention when adding further vertices to the giant connected component.

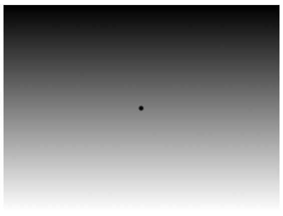
Other than that, it becomes clear that re-trying transactions is very costly. The performance of *stm* compared to *stm-while* is not only significantly slower but also less stable. Figure 6.2 shows not a single re-try for *stm-while* is recorded. It is very likely that this is due to *multiverse* spawning new threads for every transaction internally. Thereby, more software threads than hardware threads are run, directly influencing performance negatively. Conflicts are results of indeterministic thread scheduling. The more threads need to be scheduled, the more potential conflicts to handle and the more new threads are spawned.

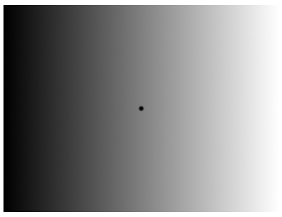
Conclusively, it seems that the optimistic algorithm is suitable for graphs of low connectivity and that the performance of the pure STM algorithm is very unreliable. Both these conclusions are not very encouraging with regard to image analysis. One would, of course, like to run image analysis algorithms that exhibit good performance, no matter the structure of the input, and also to enjoy stable performance running the same algorithm on the same input twice.

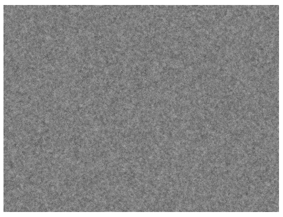
6.4 Area Opening

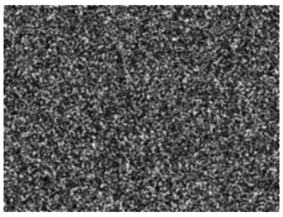
6.4.1 Input Data

The input data for performance experiments on area opening are, of course, images. We have to distinguish between two kinds of images, namely synthetic images and natural images. Synthetic images are images generated by a com-


(a) Vertical gray-scale gradient (*grad-vert*).


(b) Horizontal gray-scale gradient (*grad-hrz*).


(c) Fine grained noise (*noise-fine*).


(d) Coarse grained noise (*noise-coarse*).

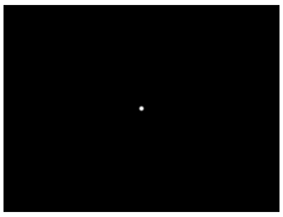

(e) Flat black component (*flat*).

Figure 6.5: Synthetic input images for area-opening experiments. Each image models an extreme case of gray-scale image structure.

puter program (either programatically or manually). Natural images are images obtained through cameras.

For the experiments presented here, synthetic images were used. Synthetic images yield the advantage that one can benchmark the algorithms in a very controlled manner and that it is easy to model extreme cases. The images are each of 1600×1200 pixels size each. They were generated using *GNU Gimp* and model five extreme case, displayed in figure 6.5. The extreme cases are chosen not only to asses the performance of the parallel algorithms systematically, but also to verify a number of hypotheses.

Hypothesis 1 The vertical gradient image in figure 6.5a (*grad-vert*) is supposed to force the sub-image filtering algorithms to work sequentially when scanning pixels from light to dark. Each thread has to wait until the global level pointer reaches the level that is within the threads pixel range. While figure 6.5b also consists of a gradient from black to white, the gradient direction is rotated by $90°$(*grad-hrz*). The hypothesis is that sub-image filtering on *grad-vert* is slower than on *grad-hrz*, because there is no expected starvation for the latter, and that, overall, *grad-vert* is the worst-case for sub-image algorithms. Furthermore, one can assume that filtering by lines outperforms sub-image filtering in this case.

Hypothesis 2 Figures 6.5c (*noise-fine*) and 6.5d (*noise-coarse*) show images generated from random white noise to which a Gaussian filter has been applied. The granularity of the noise differs, in order to vary the size of the connected components in the image. We can expect high performance on both *noise-fine* and *noise-coarse*, since the initial connectivity is low, when the images are seen as graphs where each pixel is definitely connected to its neighbors through an edge if the two pixels are at the same gray level.

Hypothesis 3 Finally, figure 6.5e (*flat*) shows a single, major level component and therefore has an extremely high connectivity. Based on the results from section 6.3.2, we can expect high contention for each algorithm, since nearly all pixels are part of the same flat component.

6.4.2 Results and Discussion

Notation and Data Selection

In this section, I will systematically analyze the results from benchmarking the different area opening algorithms, based on the hypotheses from section 6.4.1. The figures reference the various algorithms according to table 5.1. The only operation measured is the main-loop of the algorithms. Again, the resolving step is omitted entirely, mainly because the parallel resolving step would be the same for all parallel algorithms. For brevity, I will not show and discuss the entire hypercube of experiments and configurations, and instead present selected results based on their relevance. After all, we face sixteen different algorithms – the algorithms use not only two different variants of shared union-find, but also different sorting algorithms or shared queues – and five different images.

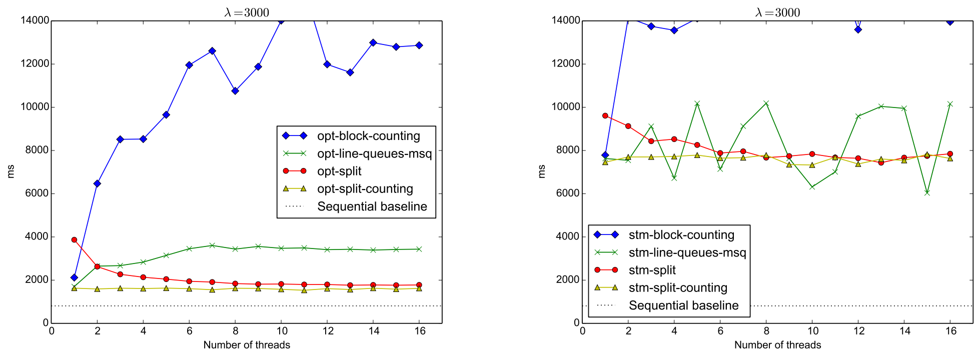
The raw data shows very similar results for *block-counting* and *block-bucket*, as well as *line-queues-msq* and *line-queues-array*, respectively, regardless of the underlying data structures. Therefore, only one configuration of each of these algorithm pairs is depicted in this section. The raw benchmark results for all algorithms can be found in appendix B. The different queue implementations only exhibit different performance in the case of *pixel-queues-msq* and *pixel-queues-array*, which we will discuss in detail in section 6.4.3.

In this section, the optimistic, fine-grained union-find algorithm is denoted *opt*, while the STM variant is simply denoted *stm*. If an algorithm uses any of those algorithms, it is prepended by the according union-find acronym.

The execution times for all synthetic images and for $\lambda = 3000$ are shown in figures 6.6 through 6.10. In the left column, we can see the performance of the optimistic union-find variant, while on the right, the STM variant's performance is depicted. The legend holds for each plot of the same column.

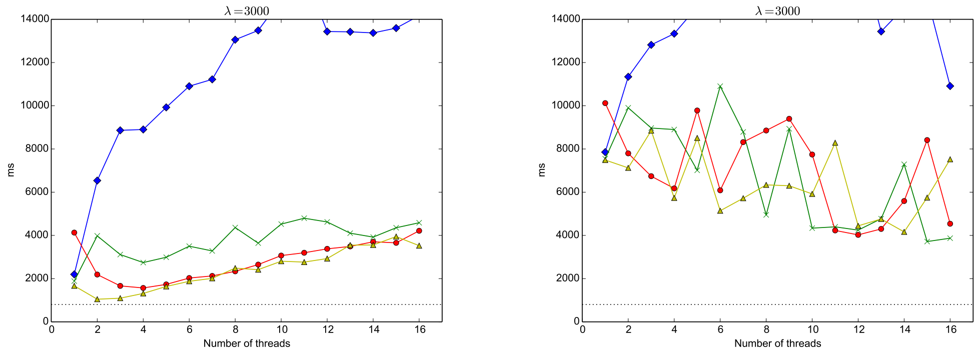
Analysis

When comparing the performance of optimistic and STM algorithms, it becomes clear that the STM variants are systematically slower than the optimistic implementations. This suggests that for practical values of λ, the connectivity of the image is not high enough to provoke as much contention as exhibited in the experiments from section 6.3.2. Not only this, but their performance is developing much more unstable over the number of hardware threads. This is especially explicit in figures 6.8 and 6.9. None of the STM variants comes close to the performance baseline of the sequential algorithm. Because of this obvious infeasibility of the STM algorithms (and for brevity), I will focus on the analysis of those algorithms that use optimistic union-find.



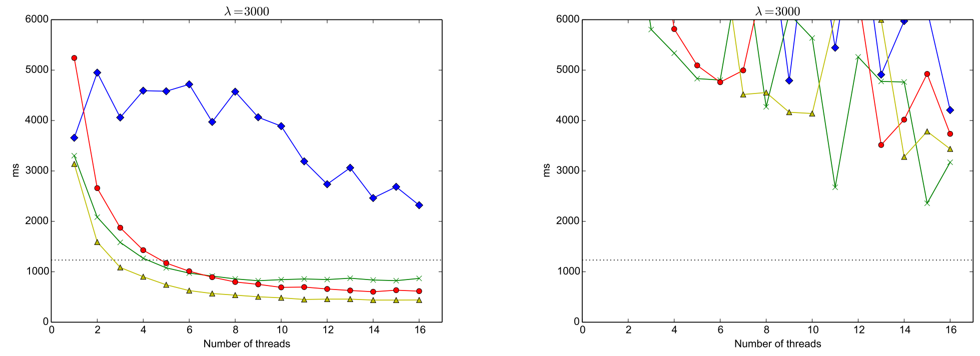
(a) Optimistic (b) STM

Figure 6.6: Execution time in milliseconds for *grad-vert*. In both cases, the performance of sub-image filtering algorithms is stable across number of threads.



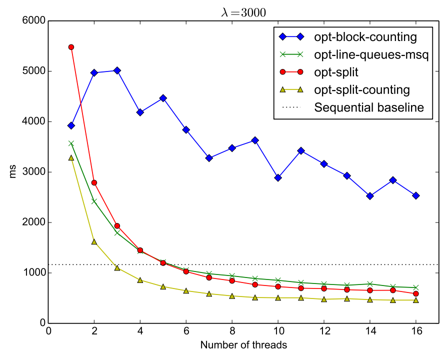
(a) Optimistic (b) STM

Figure 6.7: Execution time in milliseconds for *grad-hrz*. The optimistic algorithms scale negatively with the number of threads, while the STM algorithms remain unstable in performance.

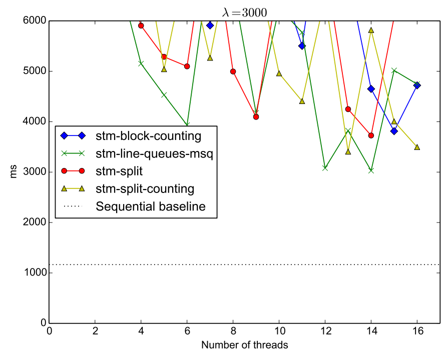


(a) Optimistic (b) STM

Figure 6.8: Execution time in milliseconds for *noise-fine*. The optimistic sub-image filtering algorithms improve over the sequential baseline. The STM algorithms run very slow in comparison.

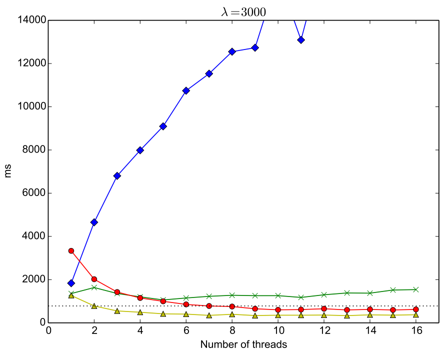


(a) Optimistic

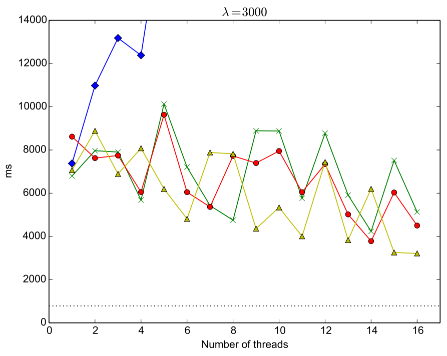


(b) STM

Figure 6.9: Execution time in milliseconds for *noise-coarse*. The performance is comparable to *noise-fine*.



(a) Optimistic



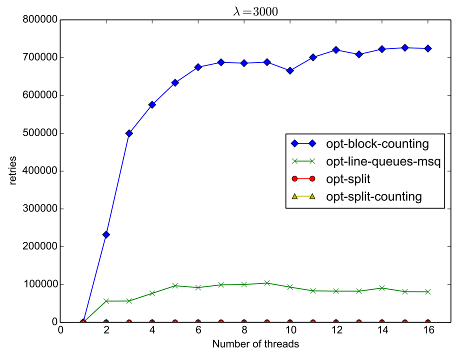
(b) STM

Figure 6.10: Execution time in milliseconds for *flat*. The optimistic sub-image filtering algorithms outperform the sequential algorithm. Still, the STM algorithms do not come close to the sequential baseline.

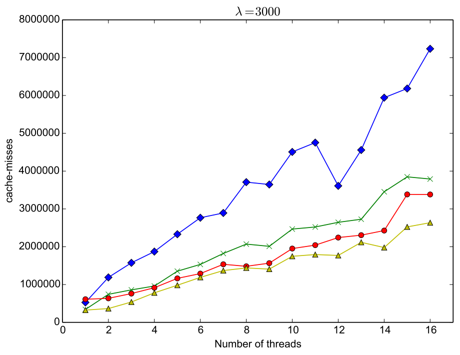
Hypothesis 1 The results, depicted in figure 6.6, support the initial hypothesis that *grad-vert* forces sub-image filtering algorithms to work in a sequential fashion. The performance of those algorithms is stable over the number of threads. There are two more observations that require further analysis.

Firstly, *line-queues-msq* is slower than the already to sequential performance forced sub-image algorithms. Since the pixel-lines are shared between threads, the algorithm should encounter less starvation, and therefore perform faster. The decline in performance is actually due to contention, as we can see in figure 6.11a and due to increased cache misses (not displayed). Because the threads work sequentially, when performing sub-image filtering, there is nearly no lock contention on the union-find roots. In contrast, line-by-line filtering encounters contention already with only a few threads, as the threads are able to make progress in parallel.

Another interesting observation is that the performance of shared sorting



(a) Retries for *grad-vert*.



(b) Cache misses for *grad-hrz*.

Figure 6.11: Performance metrics for *grad-vert* and *grad-hrz* (optimistic). The amount of re-tries for sub-image filtering algorithms on *grad-vert* is zero, as they are forced to work sequentially. The number of cache-misses does not reveal any insights into the negatively scaling performance on *grad-hrz*.

(*block-counting*) is poor in every exhibited case. This is either due to contention during sorting, due to starvation when iterating over the shared list of pixels or even because of conflicts during uniting pixels. Figure 6.11a suggests that the latter is a major factor.

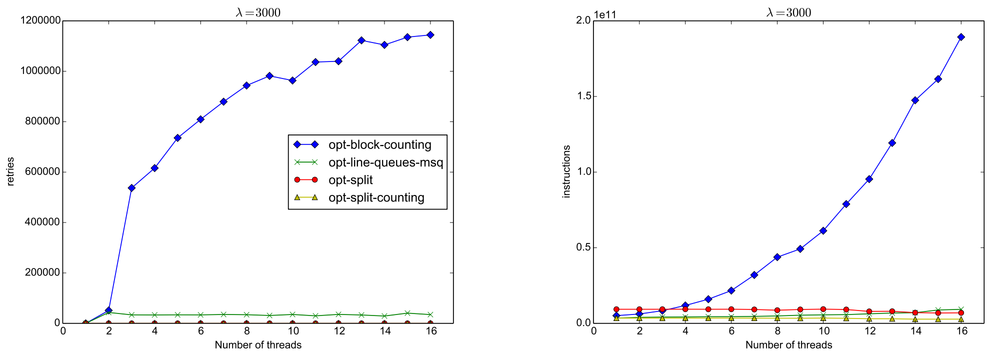
Secondly, the performance on *grad-hrz* scales negatively with the number of threads. This is surprising, because the horizontal gradient should provoke much less starvation than *grad-vert*. The recorded metrics show a linear increase of cache misses for all algorithms over the number of threads (see figure 6.11b). This alone does not explain the performance decrease. We will get back to this behavior later in the text.

The hypothesis can only be confirmed partially: *grad-vert* forces sequential performance as expected, but *grad-hrz* shows decreasing performance for increasing number of threads. Also, line-based algorithms do not provide better stability on vertical gradient images.

Hypothesis 2 Figure 6.8 and 6.9 support the hypothesis that, for irregularly structured images, sub-image filter algorithms excel in performance. Unfortunately, this is still not the case for STM variants. Again, we can see that the overhead of maintaining single pixel-lines does not pay off.

The sub-image filter algorithms and the pixel-line based algorithms scale nicely with the number of threads. Their performance scales effectively up to around eight threads, from where on no further substantial improvement is visible.

In these cases, even the shared sorting algorithms scale with the number of available hardware threads. Nevertheless, their performance is still way below the sequential baseline. We can also see that the STM variants scale according to the number of threads, but their performance remains unstable and the plots contain nearly no useful information. However, the hypothesis can be confirmed.



(a) Number of retries. (b) Number of issued CPU instructions.

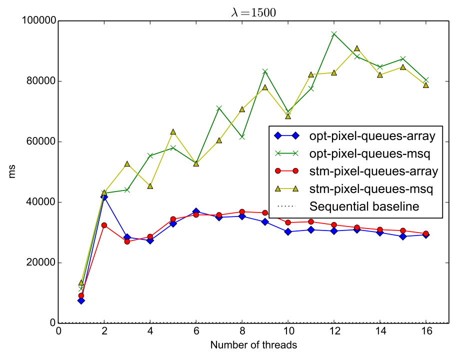
Figure 6.12: Performance metrics for *flat* (optimistic). The performance of the shared sorting algorithm is influenced to a large degree by re-tries and lock contention.

Hypothesis 3 When filtering a giant connected component, we ought to face high contention, as shown by the results obtained in section 6.3.2. Surprisingly, the results shown in figure 6.10 show only a serious performance decrease for *block-counting*. The metrics displayed in figure 6.12 show that the number of instructions increases nearly linearly with the number of hardware threads, while the number of retries is consistently high.

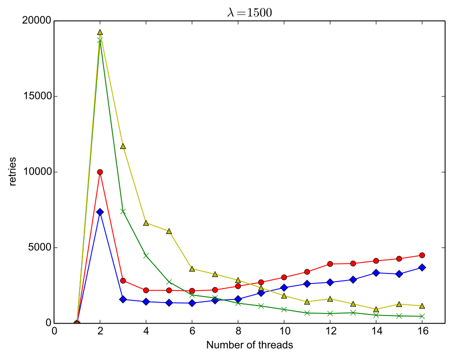
On the other hand, the sequential and sub-image filtering algorithms excel in this case. The pixel-line based algorithm does not show any performance increase, but also no decline. The giant connected component seems not to have any effect on the performance, which is comparable to the optimal case (i.e. *noise-fine* and *noise-coarse*). This is surprising, as it directly contradicts the notion that large connected components cause contention. Neither the number of retries, nor the number of issued instructions increases to any noticeable degree.

This suggests that there is few overlap between the single threads when filtering sub-images. The starting indices of the independent threads are located "far away" from each other (recall the sub-image filtering algorithms described in sections 4.1.1 and 4.2.1). Therefore, increased contention only becomes an issue very late in the filtering process, as soon as threads begin to unite connected components across sub-image borders. Since this is only a small fraction of work compared to building the thread-local connected components, its impact on the performance is minimal.

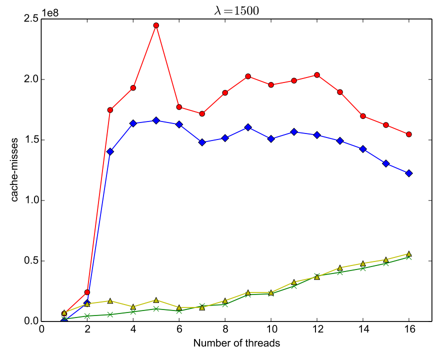
The hypothesis holds for some of the parallel area opening algorithms but cannot be confirmed completely. The structure of the sub-image filtering algorithms counters the effect of a giant connected component. This also suggests an explanation for why the performance on *grad-hrz* was unexpectedly bad for sub-image filtering: since the level components on *grad-hrz* are ordered horizontally and extend across all sub-image borders, contention is encountered very early, as each thread repeatedly tries to unite its connected component with those of other threads already in the beginning, so that all threads work on the same connected component, which is therefore locked frequently.



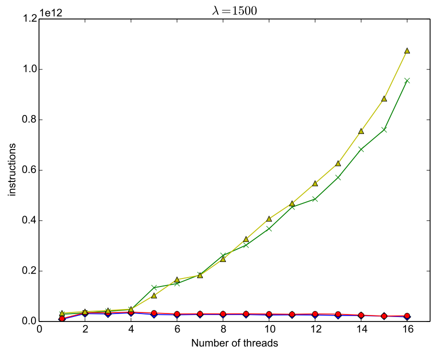
(a) Duration in milliseconds.



(b) Number of re-tries.



(c) Number of cache misses.



(d) Number of issued CPU instructions.

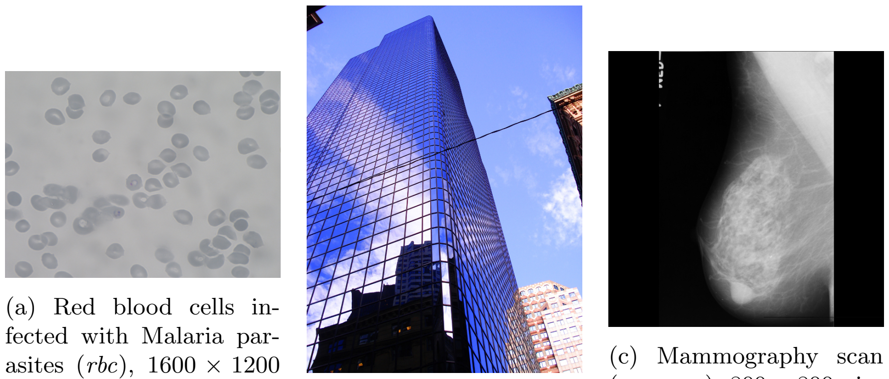
Figure 6.13: Benchmark results for *pixel-queue* variants on *noise-coarse*. The execution time is extremely high for each variant. The sequential performance baseline is barely visible.

6.4.3 Pixel Queues

The results for pixel queue algorithms are special in terms of performance and therefore, I devote a separate section to those. Figure 6.13 displays the benchmark metrics for experiments on image *noise-coarse*. For the experiments, the image has been scaled down to 800×600 pixels, in order to reduce the overall testing time. We can see from figure 6.13a that the performance of *all* variants of *pixel-queues* is dramatically worse, especially with regard to the number of hardware threads, than the base-line performance of the sequential algorithm.

Even though we can expect slow performance because of the missing sorting of pixels, the results are surprising. The naive sub-image filtering algorithm, which also operates in $O(kn)$, exceeds the sequential algorithm in performance. Executing *pixel-queues* algorithms, as described in section 6.1, on a single image took in practice about a day for all combinations of parameters. It is especially interesting to investigate the reasons for this.

Here, both union-find variants behave very similarly for all metrics. We can therefore already conclude that this is not a union-find based problem. Instead, the different queue implementations seem to suffer heavily from high through-


(a) Red blood cells infected with Malaria parasites (*rbc*), 1600 × 1200 pixels.

(b) Facade of an office building (*facade*), 600 × 800 pixels.

(c) Mammography scan (*mammo*), 800 × 800 pixels.

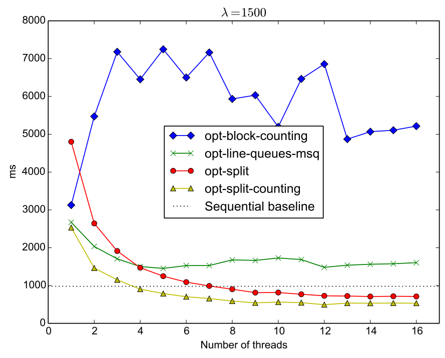
Figure 6.14: Natural input images for area-opening experiments. The images represent different real-life use cases of morphological area opening.

put. The conflicts handled by the union-find implementations do not increase together with the elapsed measured time. Therefore, even though union-find conflicts might be responsible for the initial increase, they are not the reason for the overall dramatic performance.

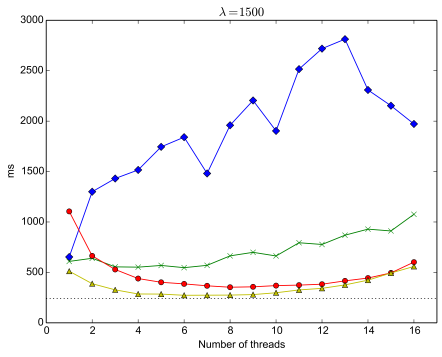
Instead, each queue data structure suffers from a different phenomenon. Figure 6.13c shows how the number of cache misses increases already when using only two threads for the bounded array queue, probably due to false sharing [34]. False sharing describes an effect caused by cache coherence protocols on shared memory processors. Each processor has its own cache, which contains copies of the data of the other processors' caches. When a processor writes a word into one of the cache lines of its local cache, the entire cache line gets invalidated for all other processors. Consequently, all other caches need to update this line. This happens especially often, if more than one independent word is stored per cache line, so that many words are falsely invalidated [34]. In many cases, this invalidation would be unnecessary and it therefore results in a slow down of the processors [34]. Notice how the shape of the plot for the bounded array queue in figure 6.13c resembles the one in figure 6.13a.

The Anderson and Woll [8] queue exhibits no false sharing. The reason for its poor performance is depicted in figure 6.13d. The more threads at this high level of throughput are used, the more conflicts are encountered. Therefore, the number of issued instructions increases with the number of threads.

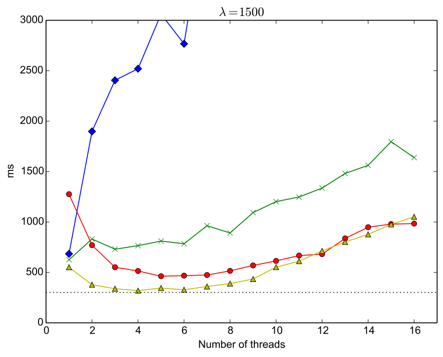
Based on these results, we can conclude that using complex data structures to maintain sole pixels, proves not to be a reliable technique for area opening. Not only the overhead of adding and removing each pixel many times from a queue, but also the described false sharing, make this type of algorithm unfeasible.



(a) Results for *rbc*.



(b) Results for *facade*.



(c) Results for *mammo*.

Figure 6.15: Execution time in milliseconds for natural images using optimistic union-find. It becomes clear that there is a lower bound on the image size with regard to speedup. Also, there is an upper bound on number of threads, as in most cases, the performance declines around eight threads.

6.4.4 Natural Images

To validate the results against real life input, we will concern three realistic cases of natural images, which can be analyzed using morphological area opening in various forms. The images are displayed in figure 6.14. Due to being captured by different means, the dimensions of these images vary. All images were converted to 8-bit gray-scale.

Figure 6.14a shows a blood sample Malaria parasites (*rbc*). The photography in figure 6.14b shows the facade of an office building (*facade*). In figure 6.14c, we see a mammography scan of a female breast (*mammo*), which has been obtained from *The Mammographic Image Analysis Society digital mammogram database* [35].

The benchmark results for the natural images are displayed in figure 6.15. It is easy to see that the behavior of the algorithms on natural images closely resembles the results obtained on synthetic images. These plots also make it very clear, that there is a lower bound on the image size, that needs to be taken into account, in order to gain any performance increase by using parallel area opening.

Also, there is an upper bound on the number of threads with regard to performance in the majority of cases. We see in both figures that the, already encountered, upper bound on threads seems to be a pattern. The performance declines around eight threads (see again figures 6.15b and 6.15c). This might be due to a larger number objects crossing sub-image borders on natural images.

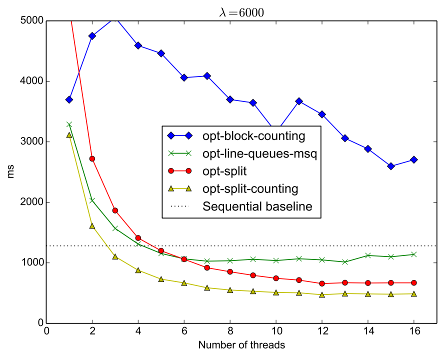
The results for *rbc* (figure 6.15a) closely resemble the results for fine grained synthetic noise images, i.e. figure 6.8. The performance on *facade* and *mammo* (figures 6.15b and 6.15c) is comparable to the results shown in figure 6.9 with a twist: the natural input images have smaller dimensions and the algorithms, at no time, outperform sequential area opening. We can conclude that this is due to the input image size and that it therefore does not pay off to filter small images (i.e. 800×600 pixels) using parallel area opening.

6.4.5 Scaling Behavior for λ

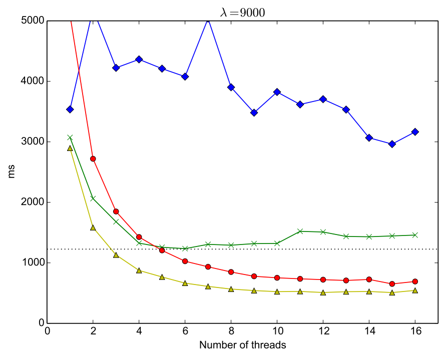
There is one more parameter, which we did not yet address. The size of λ has an effect on the size of connected components: it is the upper bound on size for non-level pixel components (see section 2.1) and therefore potentially influences lock contention. If the structure of the image only allows for components smaller than λ, or, if the image exhibits a large number of flat level components, its effect is countered.

Experiments on *noise-fine* suggest that practical values of λ do not have any immediate effect on sub-image filtering algorithms (see results in figure 6.16). Also, he sequential base-line does not change with increased λ. This is actually true for differently structured images as well, if we take the results presented in section 6.4.2 into account.

The parallel algorithms do not behave equally for increased values of λ. There is a slight performance drop exhibited by *block-counting*. Since its performance already is poor, the changes are barely noticeable. Different so for line-based filtering. By comparing figure 6.16a to figure 6.16b, it becomes obvious that the line-based algorithm does indeed suffer from increased λ. The



(a)



(b)

Figure 6.16: Benchmark results for *noise-fine* and increasing λ using optimistic union-find. Sub-image filtering algorithms are not affected by increasing values of λ, but line-based and shared-sorting variants react with declining performance.

sub-image filtering algorithms, however, exhibit constant performance for increased λ, which is a clear advantage.

We can conclude that, for practical values of λ, its effect on the performance of the sub-image algorithms is negligible in comparison to image structure. This is supported by the experimental results presented by Meijster and Wilkinson [4], which indicate no dependency of the performance of union-find area opening in relation to λ. Therefore, optimistic sub-image filtering with thread-local sorting is not only the fastest, but also the most robust parallel algorithm for area opening.

Chapter 7

Discussion

7.1 Contributions

As a major result, we can conclude, from the findings made in section 6.4, that sub-image area opening with thread-local sorting, using a fine-grained optimistic locking scheme for the union-find implementation, is the most efficient and robust parallel algorithm presented in this thesis.

Model checking, as shown in chapter 5, showed that the parallel algorithm compute the same result as the sequential algorithm by Meijster and Wilkinson [4]. It benefits from an inherent heuristic, which is an expectable large distance between the pixel start indices on the image array for the independent threads. Therefore, lock contention on single trees is low in most of the cases. There are two cases, however, which pose worst case scenarios to the algorithm. These are image gradients in x- or y-direction, which dominate the image structure. In these cases, the algorithm is either (1) forced to work in a sequential fashion, which is slower than the sequential baseline, due to the threading overhead, or (2) encounters high lock contention, due to level components extending excessively across sub-image borders.

It is interesting to see that parallel algorithms with the same formal complexity behave very differently in practice. For example, the shared sorting algorithms, which run in $O(k + n)$, do not perform well in any case, while the most naive algorithm in $O(kn)$ (that is, sub-image filtering without sorting) is second to best in every experiment.

Effects such as false sharing [34] and lock contention have a huge influence on the actual performance. So does the input structure, apart from its size, as we saw in the results obtained by independent union-find experiments, shown in section 6.3.2. Here, the connectivity of the input graph made a large difference in terms of performance.

A similar algorithm for labeling connected flat components on gray-scale images, based on union-find was proposed by Hesselink et al. [36]. The algorithm they proposed does *not* compute area opening, which is the major difference to this thesis. Their approach uses no shared union-find data structure, but a message passing system through which ownership of disjoint sets is moved between threads. This means that Hesselink et al.'s approach does not benefit from implicitly shared data structures. Still, their results show that their algorithm scales linearly with the number of hardware threads. Even though their algorithm and the ones developed in this thesis are closely related, they are only comparable to some degree.

Sub-image parallel area opening, in its current version, does not scale linearly with the number of hardware threads. In the presented experimental results, it first begins to show a noticeable improvement over the sequential algorithm when running on four threads, and already stops improving at around eight to ten threads. This upper bound will presumably increase with the size of the input. Nevertheless, this also means that, in order to gain a substantial performance improvement, we need to provide the algorithm with a sufficiently large image to filter. This suggests that the sequential algorithm by Meijster and Wilkinson [4] already performs optimally.

The number of algorithms developed in this thesis is comparably large in order to give a comprehensive overview over efficient parallelization strategies for area opening. Knowing which algorithms yield good performances, only analyzing a few of those with less parameters (i.e. parallel union-find variants, different implementations of auxiliary data structures, etc.) will provide further interesting insights.

7.2 Future Work

Taken the insights found in this thesis, there are quite a number of possibilities for further research directions and implementation work. For example, a more remote task would be bug-fixing and extending JPF to be able to fully model check the presented algorithms.

In order to perform a more general verification of area opening algorithms, it would be helpful to find the "sweet spot" between operational correctness conditions, as presented in section 5.3.1, and black-box testing conditions, as shown in section 5.1. This could help to identify and verify even more efficient parallel area opening algorithms and also to verify generalizations of parallel area opening. Such properties should also focus more on consistency. A node's parent value is invariant as soon as it is equal or greater to zero, so that $I:\ 0 \leq$ `parent(find(x))` and $\{I\}\ c\ \{I\}$, where c is any function defined in section 2.3. This invariant implies the absence of deletion in the union-find data structure and should be part of an extended formal validation of parallel area opening to ensure consistency.

As already pointed out, there exists a generalized variant of area opening, namely morphological attribute opening [20]. The sub-image area opening algorithm could, in principle, easily be extended to attribute openings, as done by Wilkinson and Roerdink [3] for the sequential algorithm [4]. Additionally, Meijster and Wilkinson [21] showed that union-find based area opening can directly be projected onto the computation of area granulometry, or area pattern spectra [4]. Area pattern spectra are histograms of the size distribution over a given image [17]. Computing these spectra is helpful in order to estimate the average size of elements that are expected to pose the majority of an image for further segmentation.

It would be interesting to assess the performance of sub-image parallel area opening more deeply. In order to avoid the issues of micro benchmarks in managed languages like Java, one could re-implement the algorithm in a mediumlevel language, such as C or C++. Additionally, the algorithm could be extended to also filter 3D voxel images. Based on the findings that large images benefit more from the parallel algorithms, the performance on three dimensional data can be expected to be very fast. Furthermore, the performance of sub-image parallel area opening should be compared systematically to the performance of parallel Max-Tree computation [5].

Also, the number of regional maxima on images could then be controlled to a greater extend, for example by controlling the number of level components explicitly, and there would be more room for more exhaustive performance experiments.

Furthermore, the sub-image filtering algorithm needs to be stabilized against worst case image scenarios. One approach could be to simply use the sequential algorithm in such cases, based on a heuristic analysis of the image structure. Analyzing the image entirely once, and then filtering it, is, on average, probably just as slow as always simply running the parallel algorithm. Instead, one could examine how the image can be split up between threads, as to avoid sub-image patterns, which are prone to contention with regard to image structure. For example, instead of defining intervals on the pixel array, in which each thread may access pixels, one could assign a rectangle of pixels to each thread that is also limited on the x axis.

Lastly, one should mention that modern GPUs provide a fast platform for parallel image algorithms and it is therefore tempting to think about an area opening algorithm for GPUs. An approach to implementing sub-image parallel area opening on GPUs could be a union of the algorithm presented in this thesis and the cross-thread message passing union-find algorithm for flat component labeling, as presented by Hesselink et al. [36].

Chapter 8

Conclusion

In this thesis, I presented a variety of parallel algorithms to compute morphological area opening based on the sequential, union-find based algorithm by Meijster and Wilkinson [4]. In particular, one of these parallel algorithms – namely parallel sub-image area opening with thread local pixel sorting, which is based on a concurrent union-find data structure, that uses a fine-grained locking scheme – proved to outperform the sequential algorithm on irregularly structured and flat images. Nevertheless, there exist at least two pathological cases of input images, which let the performance of the new parallel algorithm deteriorate. Future work should aim at finding a solution to this. Additionally, I defined formal correctness conditions and showed that model checking the new, parallel area opening algorithms through Java Pathfinder [9] does not reveal any errors. JPF identified bugs in deliberately buggy implementations, which suggests that the here developed algorithms are correct.

Furthermore, I provided a detailed discussion of the original sequential algorithm, as well as its formalization and examples of its filtering effect on images. Additionally, I identified parallel union-find algorithms that can be mapped to area opening. The various union-find algorithms have been examined in detail and their performance has been assessed independently on differently structured input graphs. This showed that wait-free union-find and a mixed wait-free and STM union-find algorithm outperform optimistic, fine-grained locking union-find on graphs of high connectivity, since lock contention on a single giant connected component is simply too high.

In contrast to this, we saw that these limitations do not necessarily apply to parallel sub-image area opening, because here, each thread starts its work on pixels on the image, which, on the array of image pixels, are located "far away" from each other. Extensive lock contention is limited to a small portion of work at the end of the filtering process. In the case of parallel area opening, STM implementations of union-find exhibit systematically lower performance. This might be due to the chosen STM library and further research on this is needed.

This thesis opens up for many future research directions. While some of these suggestions focus on validation and extension of the results presented in this thesis, others focus on technical improvements. I believe that both are highly relevant for parallel area opening to become a fast and reliable tool for use in image analysis.

Bibliography

[1] Jean Serra and Luc Vincent. An overview of morphological filtering. *Circuits, Systems, and Signal Processing*, 11(1):47–108, March 1992. doi: 10.1007/bf01189221. URL `http://dx.doi.org/10.1007/bf01189221`.

[2] Luc Vincent. Morphological Area Openings and Closings for Greyscale Images. In Ying-Lie, Alexander Toet, David Foster, HenkJ Heijmans, and Peter Meer, editors, *Shape in Picture*, volume 126 of *NATO ASI Series*, pages 197–208. Springer Berlin Heidelberg, 1994. doi: 10.1007/978-3-662-03039-4_13. URL `http://dx.doi.org/10.1007/978-3-662-03039-4_13`.

[3] Michael H. F. Wilkinson and Jos B. T. M. Roerdink. Fast Morphological Attribute Operations Using Tarjan's Union-Find Algorithm. In *In Proceedings of the ISMM2000*, pages 311–320, 2000. URL `http://www.cs.rug.nl/~michael/ismm2000.pdf`.

[4] A. Meijster and M. H. F. Wilkinson. A comparison of algorithms for connected set openings and closings. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(4):484–494, April 2002. ISSN 0162-8828. doi: 10.1109/34.993556. URL `http://dx.doi.org/10.1109/34.993556`.

[5] M. H. F. Wilkinson, Hui Gao, W. H. Hesselink, J. E. Jonker, and A. Meijster. Concurrent Computation of Attribute Filters on Shared Memory Parallel Machines. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(10):1800–1813, October 2008. ISSN 0162-8828. doi: 10.1109/tpami.2007.70836. URL `http://dx.doi.org/10.1109/tpami.2007.70836`.

[6] Robert E. Tarjan. *Data structures and network algorithms*. Society For Industrial And Applied Mathematics, 1983. ISBN 9780898711875. URL `http://www.worldcat.org/isbn/9780898711875`.

[7] Igor Berman. Multicore Programming in the Face of Metamorphosis: Union-Find as an Example. Master's thesis, Tel-Aviv University, School of Computer Science, Schreiber Building, Tel Aviv University, P.O.B. 39040, Ramat Aviv, Tel Aviv 69978, July 2010. URL `http://mcg.cs.tau.ac.il/papers/igor-berman-msc.pdf`.

[8] Richard J. Anderson and Heather Woll. Wait-free Parallel Algorithms for the Union-Find Problem. In *In Proc. 23rd ACM Symposium on Theory of*

Computing, pages 370–380, 1994. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.8354`.

[9] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engg.*, 10 (2):203–232, April 2003. ISSN 0928-8910. doi: 10.1023/a:1022920129859. URL `http://dx.doi.org/10.1023/a:1022920129859`.

[10] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Elsevier/Morgan Kaufmann, 2008. ISBN 9780123705914. URL `http://www.worldcat.org/isbn/9780123705914`.

[11] D. Alistarh, K. Censor-Hillel, and N. Shavit. Are Lock-Free Concurrent Algorithms Practically Wait-Free? *ArXiv e-prints*, November 2013. URL `http://arxiv.org/pdf/1311.3200.pdf`.

[12] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice.* Addison-Wesley Professional, 1 edition, May 2006. ISBN 0321349601. URL `http://www.worldcat.org/isbn/0321349601`.

[13] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM. ISBN 0-89791-800-2. doi: 10.1145/248052.248106. URL `http://dx.doi.org/10.1145/248052.248106`.

[14] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997. doi: 10.1007/s004460050028. URL `http://dx.doi.org/10.1007/s004460050028`.

[15] Software Transactional Memory for Java & the JVM. URL `http://multiverse.codehaus.org`. Accessed: 29.01.2014.

[16] Florian Biermann. Morphological Segmentation of Blood Images for Automated Malaria Diagnosis, November 2013. URL `http://itu.dk/people/fbie/morphological_segmentation_malaria.pdf`. Semester project report, IT University of Copenhagen. Accessed 23.04.2014.

[17] K. N. R. Mohana Rao and A. G. Dempster. Area-granulometry: an improved estimator of size distribution of image objects. *Electronics Letters*, 37(15):950+, 2001. ISSN 00135194. doi: 10.1049/el:20010635. URL `http://dx.doi.org/10.1049/el:20010635`.

[18] F. Boray Tek, Andrew G. Dempster, and İzzet Kale. Parasite detection and identification for automated thin blood film malaria diagnosis. *Computer Vision and Image Understanding*, 114(1):21–32, January 2010. ISSN 10773142. doi: 10.1016/j.cviu.2009.08.003. URL `http://dx.doi.org/10.1016/j.cviu.2009.08.003`.

[19] P. Salembier, A. Oliveras, and L. Garrido. Antiextensive connected operators for image and sequence processing. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 7(4): 555–570, April 1998. ISSN 1057-7149. doi: 10.1109/83.663500. URL `http://dx.doi.org/10.1109/83.663500`.

[20] Edmond J. Breen and Ronald Jones. Attribute Openings, Thinnings, and Granulometries. *Computer Vision and Image Understanding*, 64 (3):377–389, November 1996. ISSN 10773142. doi: 10.1006/cviu.1996.0066. URL `http://www.sciencedirect.com/science/article/pii/S1077314296900661#`.

[21] A. Meijster and M. H. F. Wilkinson. Fast computation of morphological area pattern spectra. In *Image Processing, 2001. Proceedings. 2001 International Conference on*, volume 3, pages 668–671 vol.3. IEEE, 2001. ISBN 0-7803-6725-1. doi: 10.1109/icip.2001.958207. URL `http://dx.doi.org/10.1109/icip.2001.958207`.

[22] Richard Szeliski. *Computer Vision*. Springer London, London, 2011. ISBN 978-1-84882-934-3. doi: 10.1007/978-1-84882-935-0. URL `http://dx.doi.org/10.1007/978-1-84882-935-0`.

[23] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21 (8):666–677, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL `http://dx.doi.org/10.1145/359576.359585`.

[24] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. ISBN 0136744095. URL `http://portal.acm.org/citation.cfm?id=550448`.

[25] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4): 97–107, July 2004. ISSN 0163-5948. doi: 10.1145/1013886.1007526. URL `http://dx.doi.org/10.1145/1013886.1007526`.

[26] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, chapter 40, pages 553–568. Springer Berlin Heidelberg, Berlin, Heidelberg, February 2003. ISBN 978-3-540-00898-9. doi: 10.1007/3-540-36577-x_40. URL `http://dx.doi.org/10.1007/3-540-36577-x_40`.

[27] Peter Sestoft. Microbenchmarks in Java and C#, September 2013. URL `https://www.itu.dk/people/sestoft/papers/benchmarking.pdf`. Lecture Notes, IT University of Copenhagen. Accessed 17.04.2014.

[28] AMD Opteron (TM) Processor Solutions. URL `http://products.amd.com/pages/OpteronCPUDetail.aspx?id=814`. Accessed: 28.01.2014.

[29] perf: Linux profiling with performance counters. URL `https://perf.wiki.kernel.org/index.php/Main_Page`. Accessed: 18.04.2014.

[30] Kamesh Madduri and David A. Bade. GTgraph: A suite of synthetic graph generators. URL `https://github.com/dhruvbird/GTgraph`. Accessed: 12.02.2014.

[31] P. Erdös and A. Rényi. On Random Graphs, I. *Publicationes Mathematicae*, 6:290–297, 1959. URL `http://www.renyi.hu/~p_erdos/1959-11.pdf`.

[32] P. Erdös and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5:17–61, 1960. URL `https://www.renyi.hu/~p_erdos/1960-10.pdf`.

[33] Bela Bollobas. The Evolution of Random Graphs. *Transactions of the American Mathematical Society*, 286(1):257–274, November 1984. ISSN 00029947. doi: 10.2307/1999405. URL `http://dx.doi.org/10.2307/1999405`.

[34] William J. Bolosky and Michael L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proceedings of the USENIX SEDMS IV Conference*, 1993. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.3255`.

[35] J. Suckling, J. Parker, D. R. Dance, S. Astley, I. Hutt, C. Boggis, I. Ricketts, E. Stamatakis, N. Cerneaz, S. L. Kok, P. Taylor, D. Betal, and J. Savage. The Mammographic Image Analysis Society digital mammogram database. In A. G. Gale, S. M. Astley, D. R. Dance, and A. Y. Cairns, editors, *Excerta Medica*, volume 1069 of *International congress series*, pages 375–378, July 1994. URL `http://discovery.ucl.ac.uk/102193/`.

[36] Wim H. Hesselink, Arnold Meijster, and Coenraad Bron. Concurrent determination of connected components. *Science of Computer Programming*, 41(2):173–194, October 2001. ISSN 01676423. doi: 10.1016/s0167-6423(01)00007-7. URL `http://dx.doi.org/10.1016/s0167-6423(01)00007-7`.

Appendix A

Auxiliary Algorithms

A.1 Bounded Array Queue

The implementation of the bounded array queue, which I use for maintaining work items like pixels or image lines, is lock-free. It maintains two instances of `AtomicInteger`, that point to the `head` and the `tail` of the queue (i.e. indices on the pixel array). The array is always of size 2^n. This makes it possible to use the bit-wise *and* operator to compute modulo of the `head` and `tail` pointers, in order to wrap-around the pointers to point to the beginning of the underlying array again. The queue is empty, if `head` = `tail`. When an element is enqueued, `tail` is advanced, pointing to the next empty array index. When fetching an element from the queue, `head` is advanced again. Each of these operations has a complexity of constant time. The implementation of these functions is shown in detail in figure A.1. For brevity, the constructor and other, rather uninteresting, details are omitted.

A.2 Shared Queues

The `Queues` data structure maintains two concurrent queues, `upper` and `lower`, and a level pointer which indicates the currently filtered gray-scale level. On initialization, all items are enqueued to `upper`. Calling `pop`, the upper queue is successively emptied by consuming threads. After the threads are done with their work on the retrieved item, they put it into the lower queue through `enqueue`.

Atomically swapping and gray-level decrease is implemented in the auxiliary function `swapAndDecrement`. Each thread acquires a local reference of the thread global `Queues` instance. It can then issue a call to update the global queue, using its local reference to check if it is up to date:

```
public boolean swapAndDecrement(
    final AtomicReference<Queues<T>> ref) {

  // Swap queues from last observation, decrement level.
  final Queues<T> swap = new Queues<T>(lower, upper, level - 1);
  return ref.compareAndSet(this, swap);
}
```

```java
public class ConcurrentArrayQueue<E> implements Queue<E> {

  final AtomicLong head;
  final AtomicLong tail;

  final AtomicReferenceArray<E> array;
  final long mask;
  public boolean isEmpty() {
    return head.get() == tail.get();
  }
  public boolean add(final E arg0) {
    while (true) {
      final long i = tail.get();
      if (size() < array.length() - 1) {
        // If CAS fails, someone added in the meantime.
        if (tail.compareAndSet(i, i + 1)) {
          array.set((int)(i & mask), arg0);
          return true;
        }
      } else {
        // Queue was already full.
        return false;
      }
    }
  }
  public E poll() {
    while (true) {
      final long i = head.get();
      if (!isEmpty()) {
        // If CAS fails, someone polled in the meantime.
        if (head.compareAndSet(i, i + 1))
          return array.get((int)(i & mask));
      } else {

        // The queue was already empty or
        // someone else took our element.
        return null;
      }
    }
  }

}
```

Figure A.1: The implementation of the bounded array queue.

```java
public class Queues<T> {

  // Queues containing work items.
  public final Queue<T> upper;
  public final Queue<T> lower;

  // Current level pointer.
  public final int level;
}
```

Figure A.2: The `Queues` container class.

```
public static class Line {
  public final int[] sorted;
  public int p;

  public Line(final int[] pixel) {
    this.sorted = pixel;
    p = pixel.length - 1;
  }

  public final boolean advance() {
    return --p >= 0;
  }

  public final boolean workLeft() {
    return p >= 0;
  }

  public final int current() {
    return sorted[p];
  }
}
```

Figure A.3: The `Line` container class implementation.

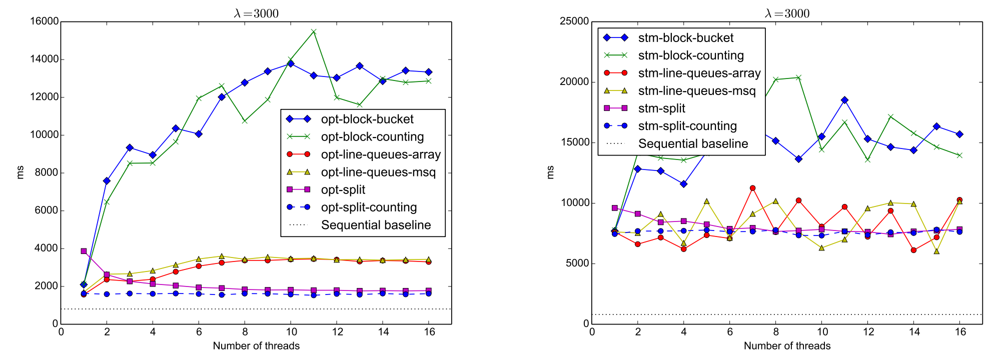
Obviously, the swap will fail if the calling thread does not have the most recent information on the global queues stored locally, so that we can make sure that it will be swapped only once per level.

A.3 Pixel Line Representation

In order to represent a single pixel line, we rely on a container class `Line`, that wraps an array, which in turn contains, by gray-scale value sorted, pixel indices. For convenience, the container class also internally maintains a pointer to the pixel, which will be returned on the next call to `current`. Furthermore, the function `advance` moves the pointer and returns true, if the internal pointer is not yet outside of the array bounds. The same check can be performed by calling `workLeft`. The entire implementation is depicted in figure A.3.

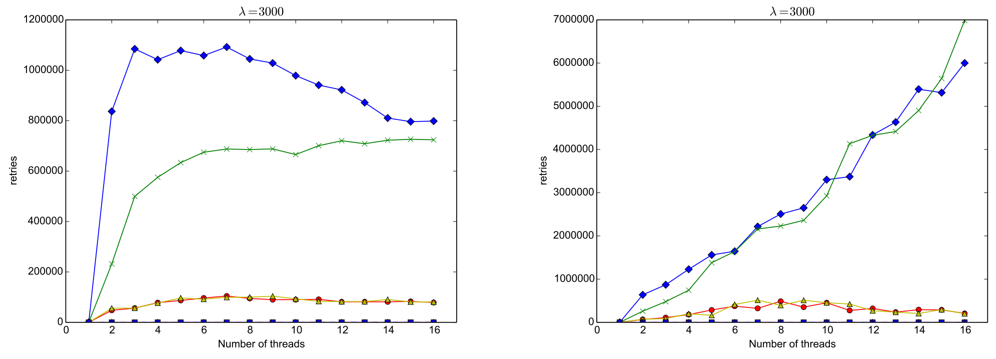
Appendix B

Raw Data Plots



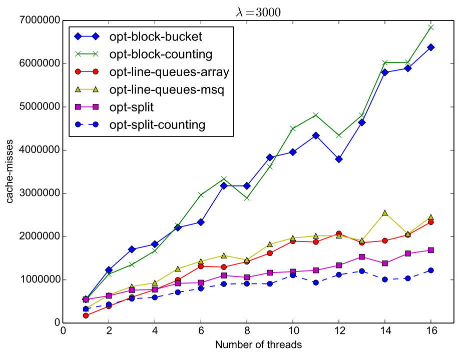
(a) Optimistic (b) STM

Figure B.1: Execution time in miliseconds for *grad-vert*.

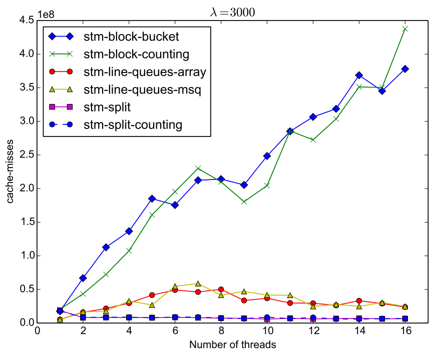


(a) Optimistic (b) STM

Figure B.2: Number of retries for *grad-vert*.

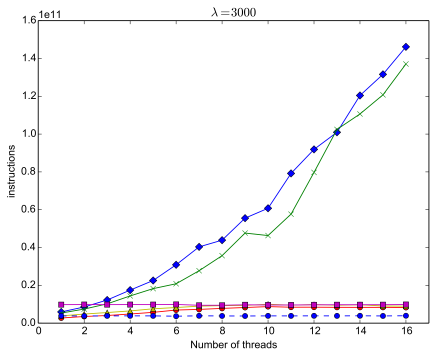


(a) Optimistic

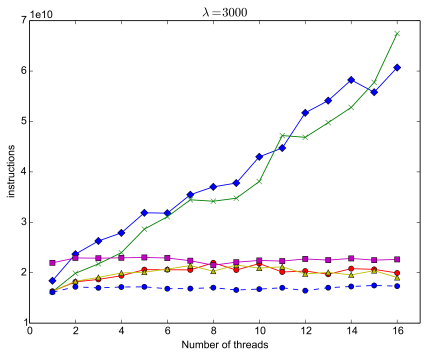


(b) STM

Figure B.3: Number of cache-misses for *grad-vert*.

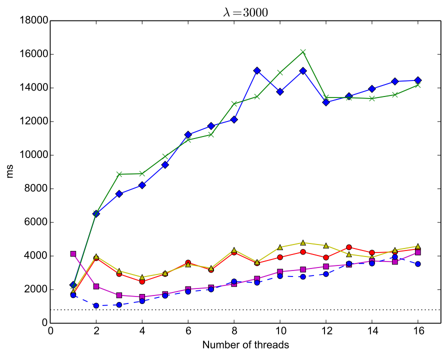


(a) Optimistic

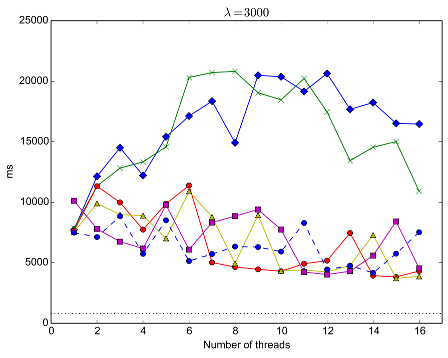


(b) STM

Figure B.4: Number of issued CPU instructions for *grad-vert*.

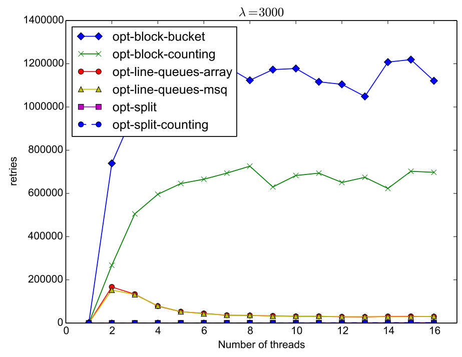


(a) Optimistic

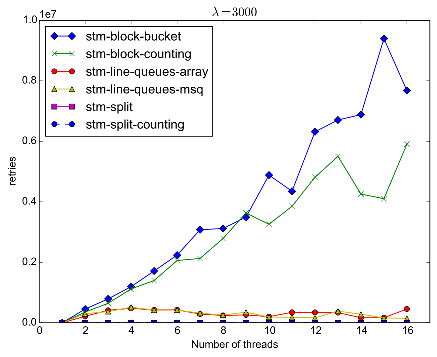


(b) STM

Figure B.5: Execution time in milliseconds for *grad-hrz*.

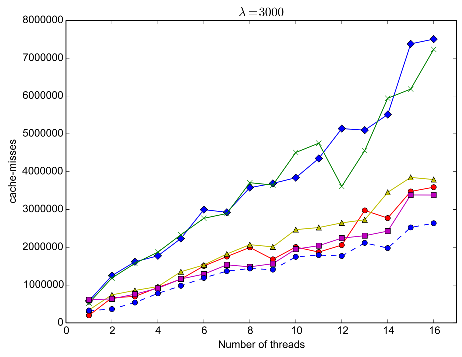


(a) Optimistic

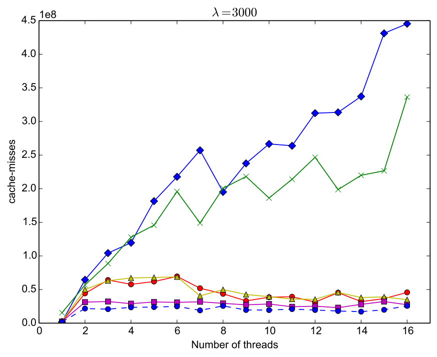


(b) STM

Figure B.6: Number of retries for *grad-hrz*.

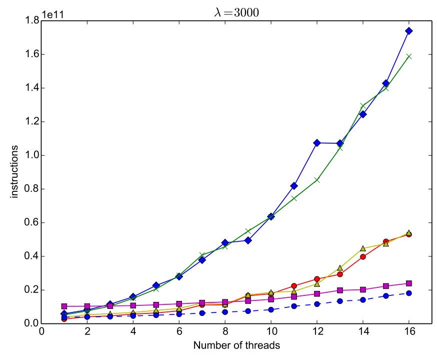


(a) Optimistic

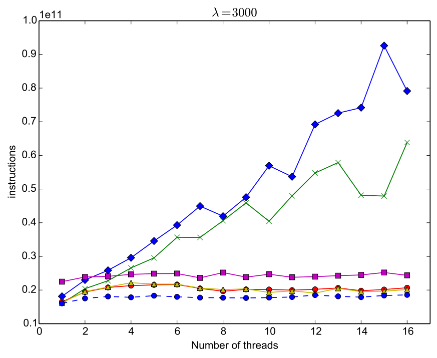


(b) STM

Figure B.7: Number of cache-misses for *grad-hrz*.

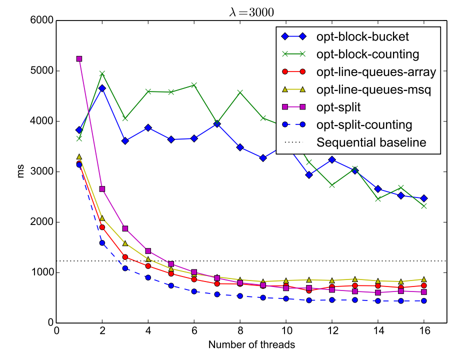


(a) Optimistic

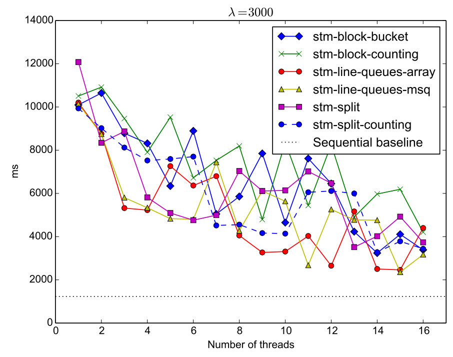


(b) STM

Figure B.8: Number of issued CPU instructions for *grad-hrz*.

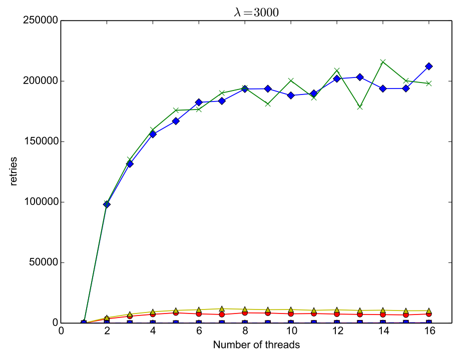


(a) Optimistic

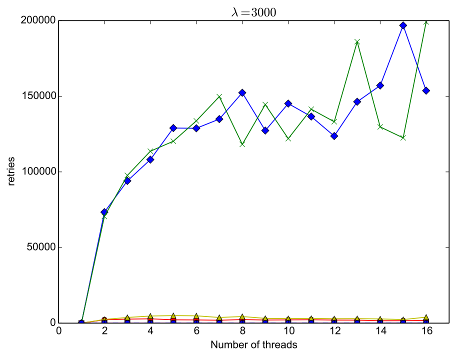


(b) STM

Figure B.9: Execution time in miliseconds for *noise-fine*.

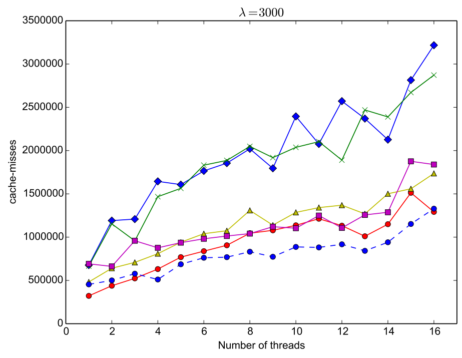


(a) Optimistic

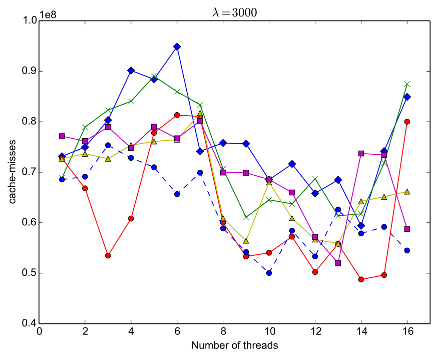


(b) STM

Figure B.10: Number of retries for *noise-fine*.

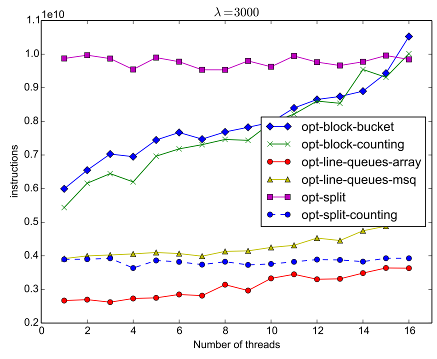


(a) Optimistic

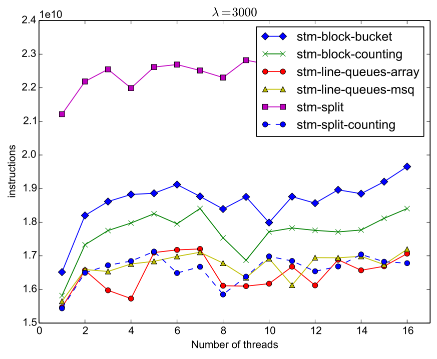


(b) STM

Figure B.11: Number of cache-misses for *noise-fine*.

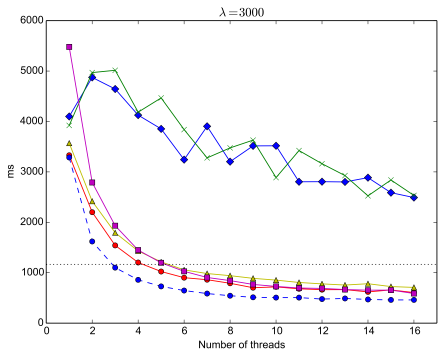


(a) Optimistic

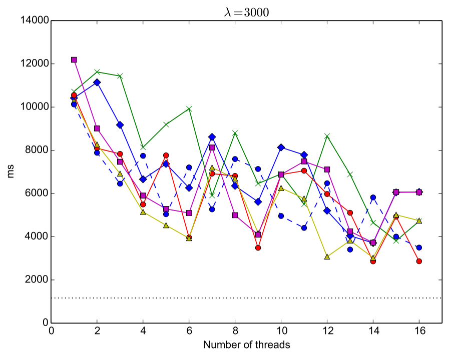


(b) STM

Figure B.12: Number of issued CPU instructions for *noise-fine*.

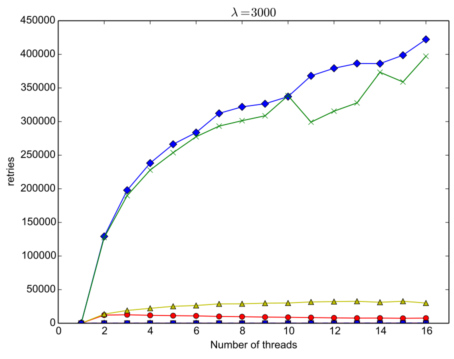


(a) Optimistic

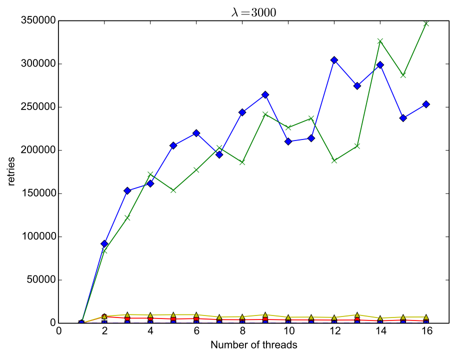


(b) STM

Figure B.13: Execution time in miliseconds for *noise-coarse*.

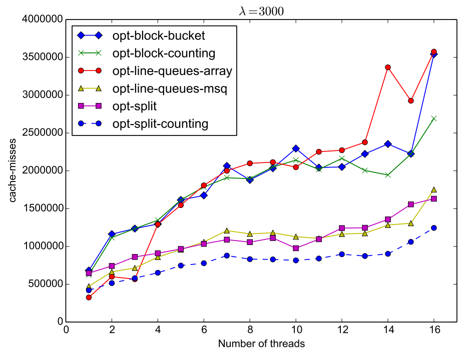


(a) Optimistic

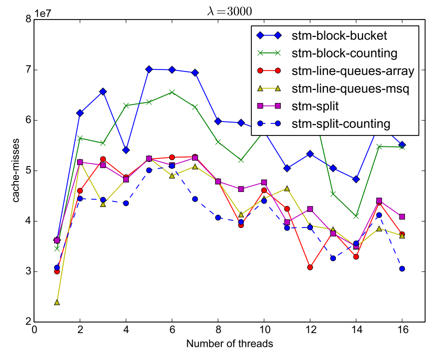


(b) STM

Figure B.14: Number of retries for *noise-coarse*.

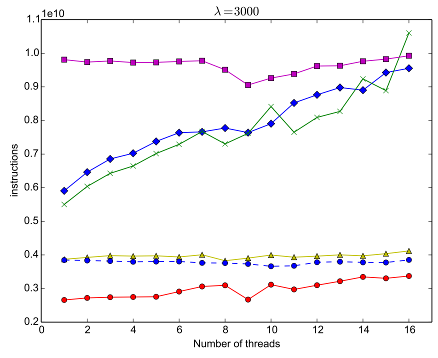


(a) Optimistic

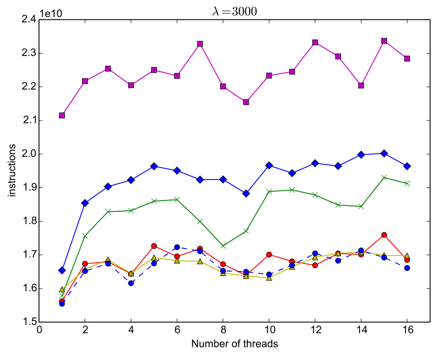


(b) STM

Figure B.15: Number of cache misses for *noise-coarse*.

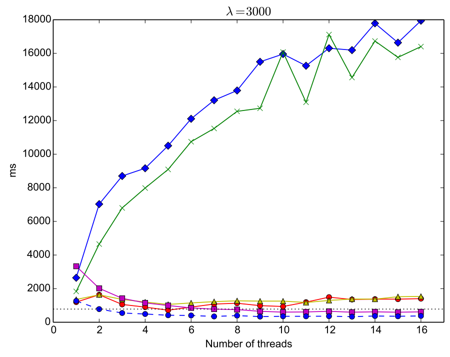


(a) Optimistic

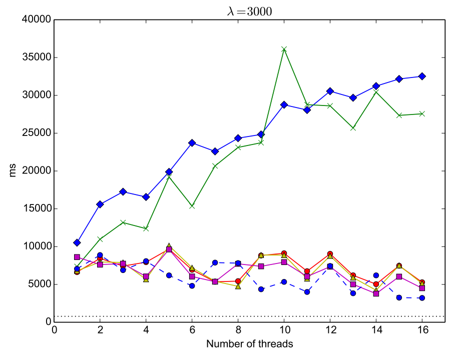


(b) STM

Figure B.16: Number of issued CPU instructions for *noise-coarse*.

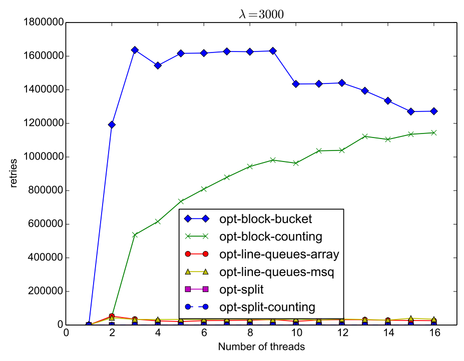


(a) Optimistic

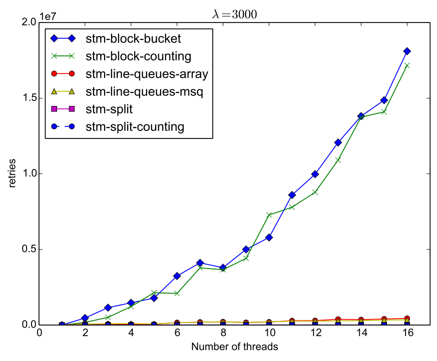


(b) STM

Figure B.17: Execution time in miliseconds for *flat*.

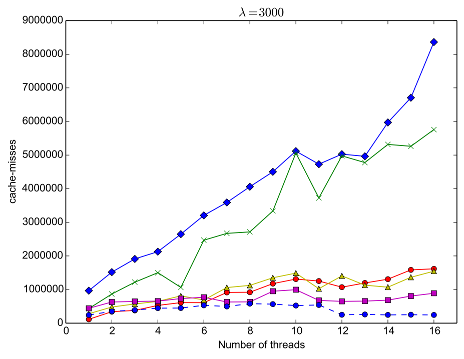


(a) Optimistic

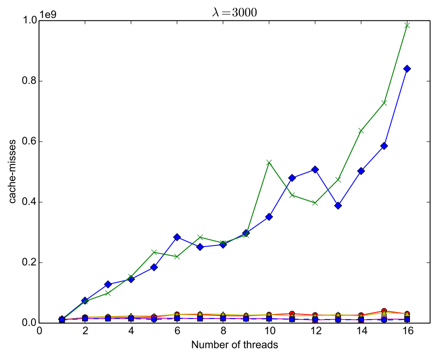


(b) STM

Figure B.18: Number of retries for *flat*.

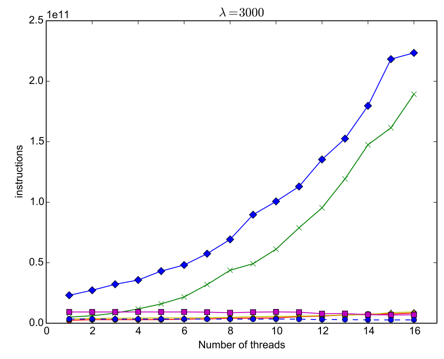


(a) Optimistic

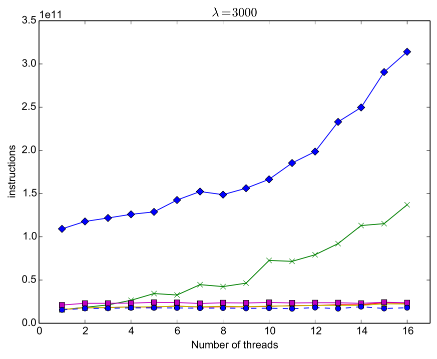


(b) STM

Figure B.19: Number of cache misses for *flat*.

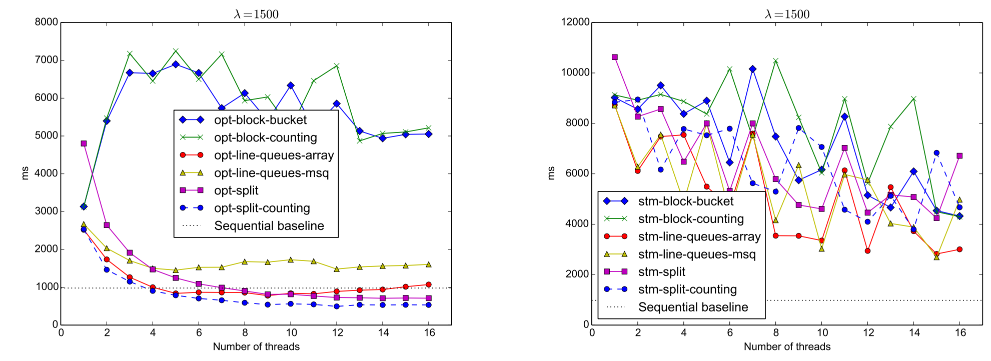


(a) Optimistic



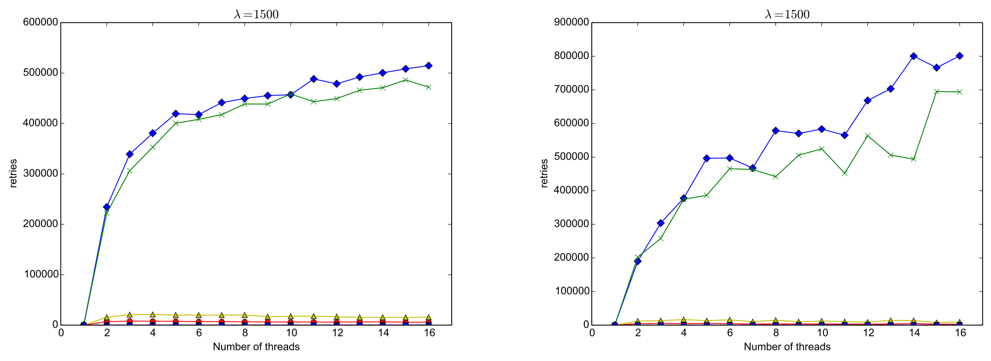
(b) STM

Figure B.20: Number of issued CPU instructions for *flat*.



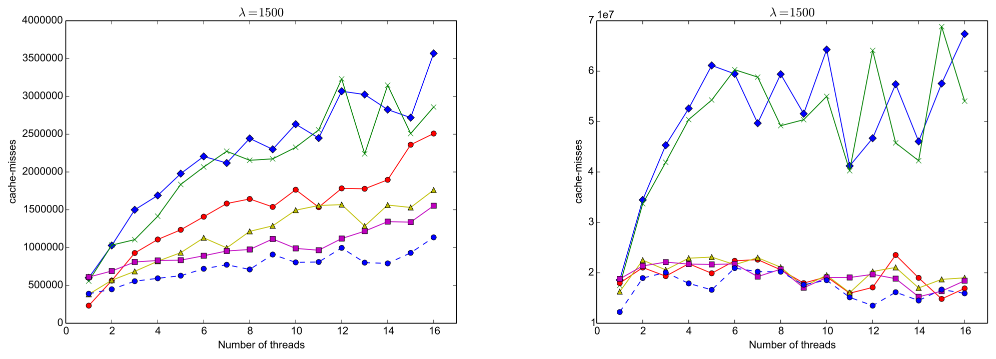
(a) Optimistic (b) STM

Figure B.21: Execution time in miliseconds for *rbc*.



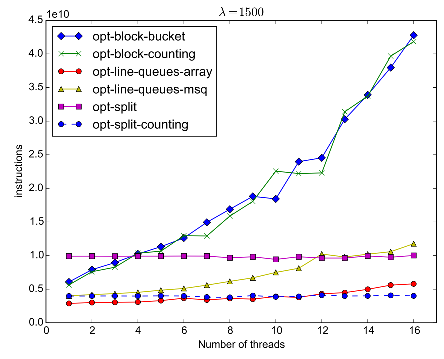
(a) Optimistic (b) STM

Figure B.22: Number of retries for *rbc*.

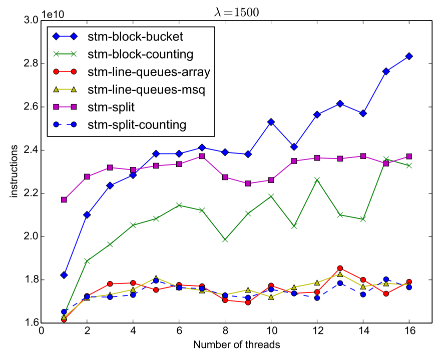


(a) Optimistic (b) STM

Figure B.23: Number of cache misses for *rbc*.

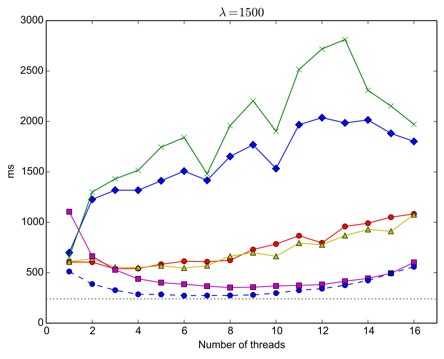


(a) Optimistic

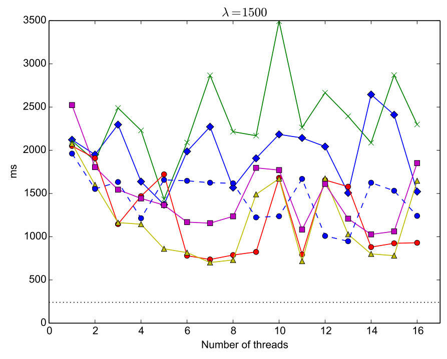


(b) STM

Figure B.24: Number of issued CPU instructions for *rbc*.

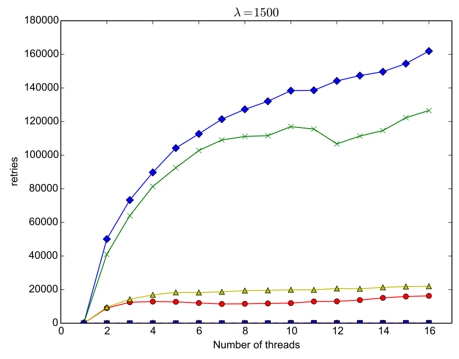


(a) Optimistic

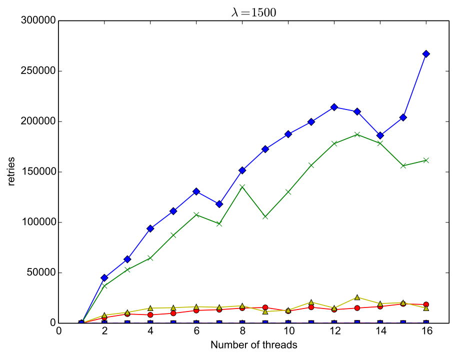


(b) STM

Figure B.25: Execution time in miliseconds for *facade*.

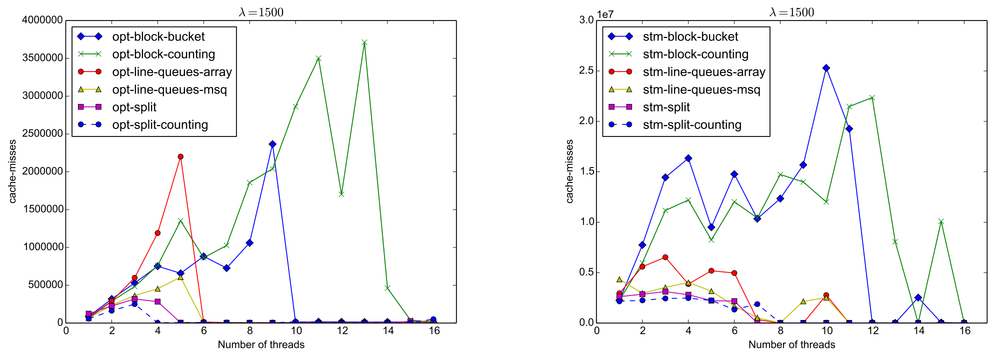


(a) Optimistic



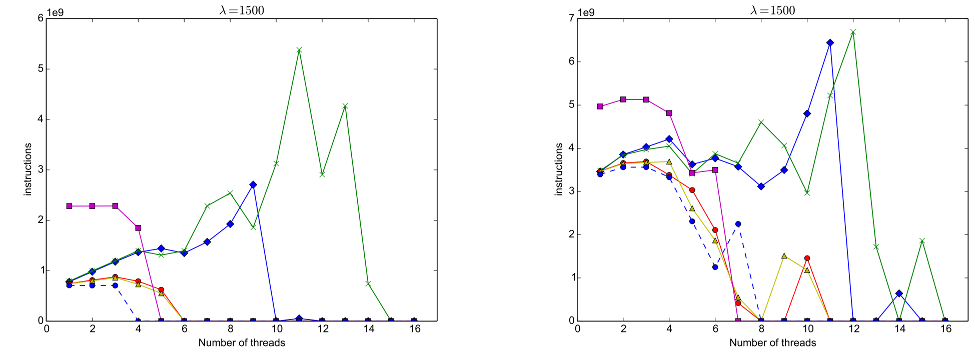
(b) STM

Figure B.26: Number of retries for *facade*.



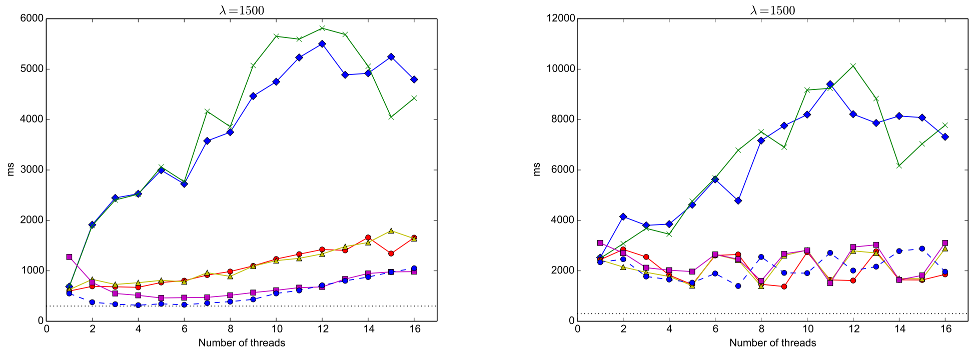
(a) Optimistic (b) STM

Figure B.27: Number of cache misses for *facade*.



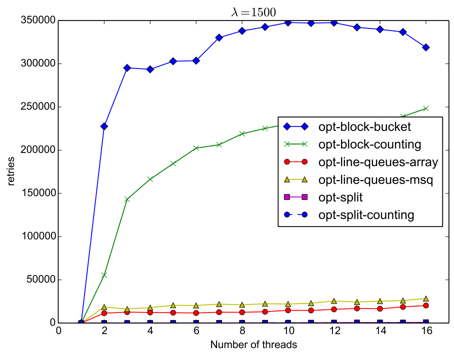
(a) Optimistic (b) STM

Figure B.28: Number of issued CPU instructions for *facade*.

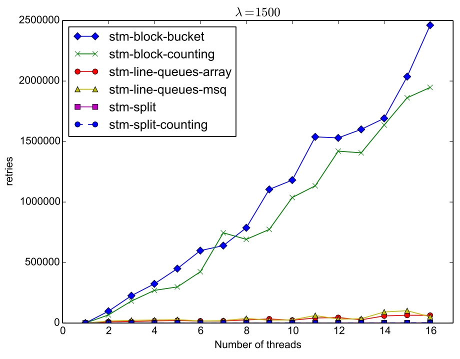


(a) Optimistic (b) STM

Figure B.29: Execution time in miliseconds for *mammo*.

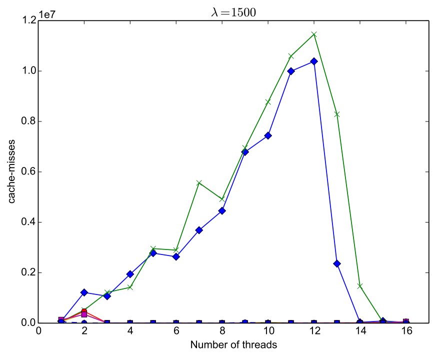


(a) Optimistic

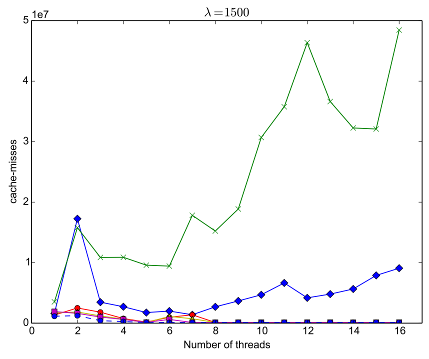


(b) STM

Figure B.30: Number of retries for *mammo*.

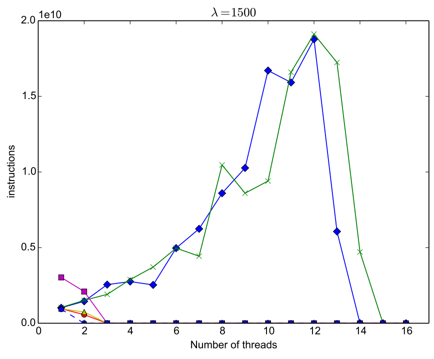


(a) Optimistic

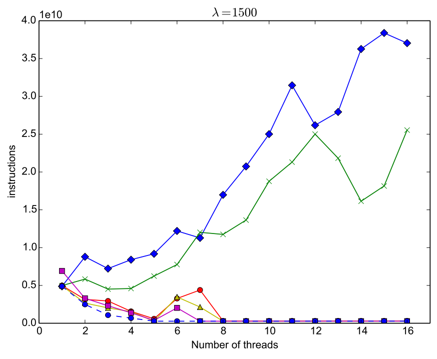


(b) STM

Figure B.31: Number of cache misses for *mammo*.



(a) Optimistic



(b) STM

Figure B.32: Number of issued CPU instructions for *mammo*.