MASTER'S THESIS 2019

Parallelization of Computational Geometry Algortihms

Gustav Kullberg

ISSN 1650-2884
LU-CS-EX 2019-15

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2019-15

Parallelization of Computational Geometry Algortihms

Gustav Kullberg

Parallelization of Computational Geometry Algortihms

Gustav Kullberg
tpi13gku@student.lu.se

June 28, 2019

Master's thesis work carried out at Silvaco Inc.

Supervisors: Dr. Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Björn Wehlin, bjorn.wehlin@silvaco.com

Examiner: Krzysztof Kuchcinski , krzysztof.kuchcinski@cs.lth.se

Abstract

When performing a design rule check(DRC) on an integrated circuit, one is required to perform boolean operations between multiple layers of the IC as part of this process. Boolean operations have become a bottleneck in the process and there is a need to reduce the computing time as number of components on IC:s grow every year.

Computational Geometry algorithms and more specifically the two-dimensional boolean operation of polygons can have troubling time complexities when performing on large amounts of data. In order to speed up the computational-time, parallel computing has been proposed. However, the geometrical dependencies created by polygons stretching across large areas of the 2-D space makes the process of parallelization non-trivial.

The aim of this thesis is to come up, evaluate and propose partitioning-strategies in order to make the problem available to parallel computing, thus speeding up the computing time of the boolean operations: AND, OR and DIF.

The result of the simulations and stress tests done show that there is a great potential in parallelizing these algorithms, with the best results showing a speed-up of approximately 260X on the AND operation. The OR operation was more troublesome in some cases where large dependencies in the output was created, making parallelizing less effective.

Keywords: DRC, boolean, parallelization, partitioning

Acknowledgements

I would like to thank Björn Wehlin for his extraordinary supervisor skills as well as helping out with ideas, feedback and running simulations on the remote machine used for the stress tests, which required much time and effort.

I would also like to thank Silvaco Inc for being able to perform my thesis using their software, problem formulations, and test cases. Their Application Engineers did a great job generating test cases that were used.

A big thanks to my academic supervisor, Dr. Jonas Skeppstedt.

Contents

1 Introduction **7**
1.1 Background 7
1.2 Purpose 8
1.3 Problem Statements 8
1.4 Limitations 9

2 Theory **11**
2.1 Parallel Computing 11
2.1.1 Approach 11
2.1.2 Threads 12
2.1.3 Dependencies 12
2.1.4 Amdahl's law 12
2.2 The Two-dimensional Boolean Operation 13
2.2.1 The Sweep Line Algorithm 14
2.2.2 Line Segment Intersection 15
2.3 Overlay 17
2.3.1 Doubly connected Edge List 17
2.3.2 Computing an Overlay 17
2.4 The Boolean operation 18
2.5 Decomposition: Partition the Boolean Operations 19
2.5.1 Binary Space Partitioning 19
2.5.2 Median-based Partitioning of 2D-space 21
2.5.3 Consequences of Partitioning 22
2.6 Assignment 23
2.6.1 Divide and Conquer 24
2.7 Orchestration 25

3 Method **27**
3.1 Partition Strategy 1 - Separate Layer 27
3.1.1 The bounding box 29

3.1.2 Parallel case . . . 30
3.2 Partition Strategy 2 - Duplication . . . 30
3.2.1 . . . 30
3.2.2 Problems . . . 32
3.3 Partition Strategy 3 - Splitting . . . 33
3.3.1 Problems . . . 34
3.4 Implementation . . . 35
3.4.1 BSP-tree . . . 36
3.4.2 Thread-Scheduling . . . 36

4 Evaluation **39**
4.1 Cases . . . 39
4.1.1 Case A . . . 39
4.1.2 Case B . . . 40
4.1.3 Case C,D . . . 40
4.2 Results . . . 41
4.2.1 Case A - poly, active - AND . . . 42
4.2.2 case A - poly, active - OR . . . 46
4.2.3 Case B, metal1, metal2 - AND . . . 47
4.2.4 Case B, metal1, metal2 - OR . . . 48
4.2.5 Case B, metal1, metal2 - DIF . . . 49
4.2.6 Case C - Metal 2, Metal3 - AND . . . 50
4.2.7 Case C - Metal 2, Metal3 - OR . . . 50
4.2.8 Case C - Metal 2, Metal3 - DIF . . . 51
4.2.9 Case D - Metal 3, Metal 5 - AND . . . 52
4.2.10 Case D - Metal 3, Metal 5 - OR . . . 53
4.2.11 Case D - Metal 3, Metal 5 - DIF . . . 54

5 Discussion **57**
5.1 Overall comments on results . . . 57
5.1.1 Partition Strategy 1 - Separate Layer . . . 57
5.1.2 Partition Strategy 2 - Duplication . . . 57
5.1.3 Partition Strategy 3 - Splitting . . . 59
5.1.4 General comments . . . 60
5.2 The Progress of the Project . . . 61
5.3 Error Sources . . . 61
5.4 Further Improvements . . . 62

6 Conclusion **63**
6.0.1 Great Potential . . . 63
6.0.2 Benefits of Partitioning . . . 63
6.0.3 Output Dependencies . . . 64

Bibliography **65**

Appendix A Detailed Results **69**

Chapter 1
Introduction

A brief introduction to this thesis is given in this chapter. we will cover subjects such as the background, problem formulations, what the purpose of the thesis is and what we hope to get out it. Other subjects covered are limitations.

1.1 Background

In recent years CPU-speeds have risen into such high levels that physical limits, e.g. heat loss and leakage current prevent further speed improvements. Thus has the idea of having a multicore processor with several cores with independent caches become an attractive idea. This, however, does not come without issues. Code generally have to be written in a multicore kind of way. That is special software that partitions the computations on the different processor cores in a proper way. This thesis work considers development of algorithms whose goals are to reach the maximum speed-up of a given algorithm previously run as a single-core program. The work has been carried out together with Silvaco Inc.

Silvaco is designing IoT chips, flat panel displays and microprocessors among many other things. A step in the manufacturing process of these chips is a verification step where various components of the chip which is to be sent to manufacturing must fulfill certain requirements set by the manufacturer. The chips are built in many layers, where each layer has many components. The verification is done in multiple steps. First, a boolean operation which takes one or two layers as input and performs one of the boolean operations OR/XOR/AND/DIF. This will create an output layer. Succeeding this, other operations are performed on this output layer such as a check-operation. This step simply consists of checking various constraints defined by the manufacturer. An example of this is that distance between components must be less than d, defined by the manufacturer. In order

to perform this check operation, the preceding boolean operation must be finished and it now works as bottleneck. It would thereby benefit this verification-process as a whole if we could speed-up the boolean operation.

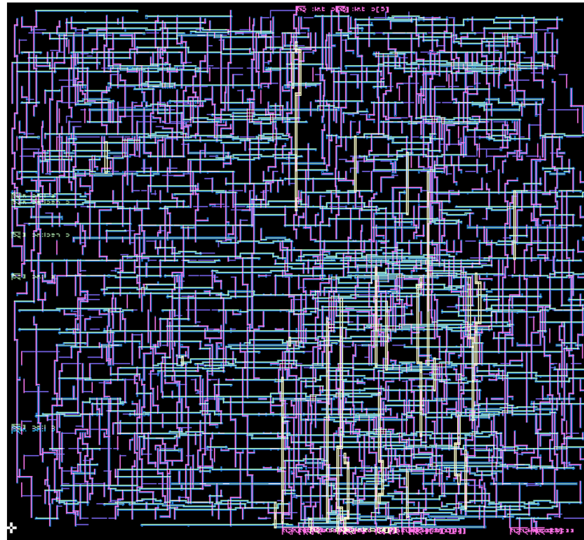


Figure 1.1: Example of a chip, layer in multiple colors

1.2 Purpose

The purpose of this thesis is to explore and evaluate different partitioning strategies and examine the following areas:

- How much parallelization does the partitioning allow us?
- Are there any created dependencies and how severe are they?
- How input-sensitive is the partitioning strategy and in what way?
- How does the partitioning affect the time complexity of the operation?

The final purpose of this report is thus to give a final recommendation regarding the chip design rule check area and the scenarios which occur most frequently there.

1.3 Problem Statements

In preparation for this project, problematic areas that needed to be address were formulated:

1. What dependencies are there between different polygons in the different algorithmic cases? What opportunities and limitations does this confront us with?

2. What is an efficient strategy to partition geometrical figures for parallel execution in a 2-D plane?
3. What is the most efficient way to deal with non-trivial boundaries of these figures?
4. How can different statistical and/or pre-computational analysis of standard chips improve the algorithm further?

1.4 Limitations

Although this thesis is tightly coupled with the practical use of design rule check of computer-chips, this thesis is purely theoretical and no knowledge of the hardware nor electronics is included.

In examining our partitioning, we will primarily consider time complexity. We do not have any ambition of examining space complexity if mentioned, it is secondary. Secondly, we consider the Boolean operation as a black box. It is, of course, necessary to understand how it works, but it has not been implemented from the ground up in this thesis.

Chapter 2
Theory

2.1 Parallel Computing

Definition

"Parallel computing is the use of two or more processors (cores, computers) in combination to solve a single problem"

2.1.1 Approach

When parallelizing a sequential program, there are three major obstacles one must be aware of [6].

1. Load Imbalance: This is the idea of an unbalanced distribution of work between the threads. It leads to some threads not working, slowing down the computation time.

2. Synchronizing: It is vital to protect shared data with locks in order to avoid data races.

3. Cache Behaviour: Coherence cache misses is very important and be taken into consideration.

Four major steps can be taken into consideration in order to counter these problems. These are [6]

1. Decomposition: We divide the work of the sequential program into smaller units called tasks. This will from here forth be referred to as partitioning.

2. Assignment: In the parallel program tasks are assigned to different threads. We may do this in different ways. One example is a fixed assignment, where one thread is responsible for one task and on task only, a row in a matrix for example.

3. Orchestration: The implementation itself, how do we write our source code to optimize the overall program. Things to consider are; How to reduce the cost of synchronization and communication, How to reduce the amount of access to shared data? How to schedule tasks to satisfy dependencies early? A guiding principle is the owner computes rule, where if we design our program in a way such that only one thread can modify some data, other threads only allowed to read it

4. Mapping: Selecting a particular processor for a thread. This is however usually beyond the control of the programmer.

2.1.2 Threads

Creating new threads allows us to do computations concurrently, effectively computing in parallel.

2.1.3 Dependencies

Dependencies between operations play an essential role in how we can partition our data. A pure definition of dependency: Consider two expression evaluations A and B, A carries a dependency to B if

- the value of A is used as an operand of B [6]
- A writes a scalar object or a bit-field in memory location M and B reads from M the value written by A, and A is sequenced before B
- For some evaluation X, A carries a dependence to X and X carries a dependency to B [6]

2.1.4 Amdahl's law

Amdahl's law was formulated by Gene Amdahl 1967 and have since played important roles whenever parallelizing sequential programs [1].

$$S = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.1}$$

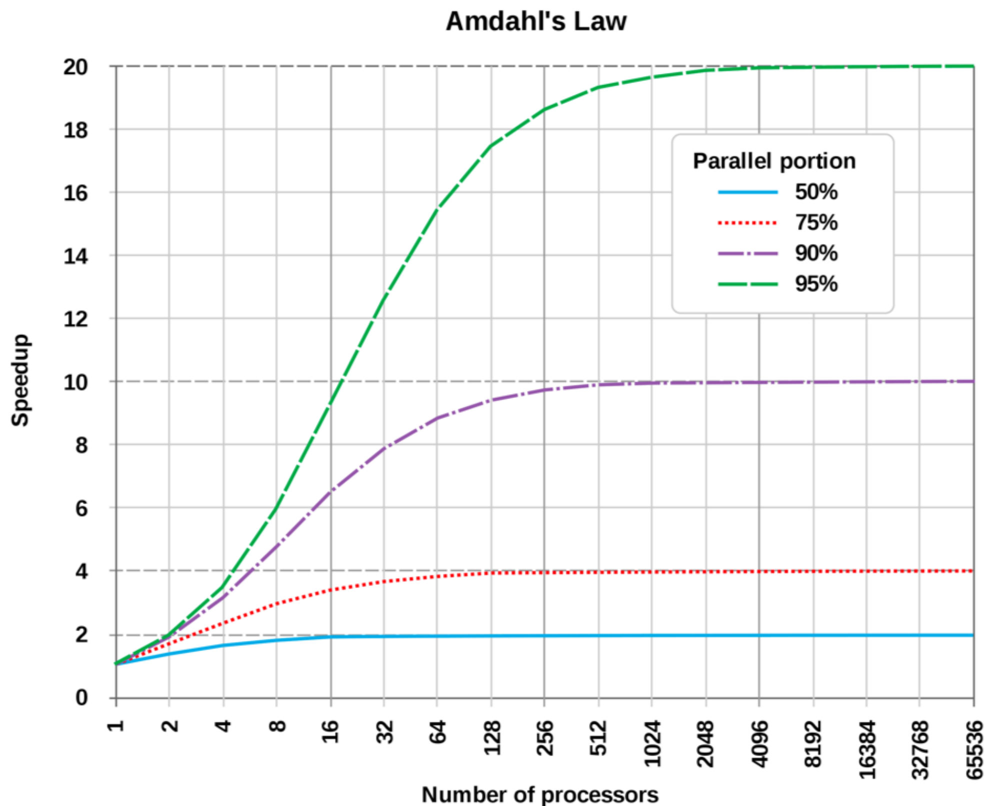
Where,
S is total speed-up of while operation
p is proportion of execution time that the part benefiting from improved resources originally occupied

s is speed-up of the parallelizable part.

Especially if $s \to \infty$ we get

$$S = \frac{1}{1-p}$$

This proves that we theoretically cannot get more speedup due to parallelization than the proportion of our program that is parallelizable. The following graph visualizes how much theoretical speedup which is possible given how parallelizable the program is.



2.2 The Two-dimensional Boolean Operation

In order to properly make good decisions when we decompose, assign and orchestrate our problem, we must first learn about it, its challenges and eventual dependencies. When talking about the Boolean operation in this thesis, we consider 1 or 2 layers of input and 1 layer output. A layer is simply multiple(an array) polygons, which in turn is made up of multiple points, which then, of course, have two properties, X and Y in the two-dimensional space. There are two types of polygons, convex and concave.

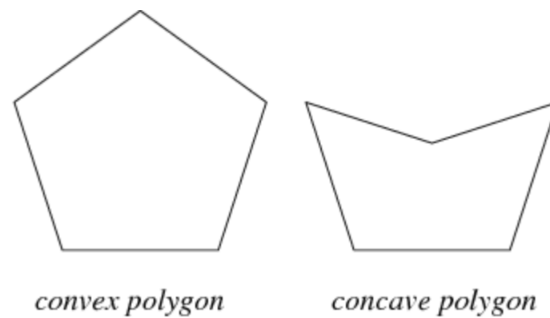


Figure 2.1: Examples of convex and concave polygons

Convex polygons are generally easier to handle. For example, it is impossible to split a convex into more than two parts with a straight line. This property does not hold for concave polygons, and will of course make the task of partitioning polygons into multiple parts more troublesome. This means that if we would divide a concave polygon with a straight line, we could theoretically get more than two polygons as a result of this. In this thesis, the Boolean operations we are considering are the following

- Union $\mathbb{L}_1 \cup \mathbb{L}_2$. Known as OR and consists of $\mathbb{L}_1$, $\mathbb{L}_2$ merged.
- Intersection $\mathbb{L}_1 \cap \mathbb{L}_2$. Known as AND and consists of the area covered by both $\mathbb{L}_1$ and L_2
- Difference $\mathbb{L}_1 - \mathbb{L}_2$. Known as DIF. Consists of the area covered by $\mathbb{L}_1$ subtracted by the area covered by $\mathbb{L}_2$
- Exclusive Or $\mathbb{L}_1 \oplus \mathbb{L}_2$. Known as XOR. Consists of area covered by either $\mathbb{L}_1$ or $\mathbb{L}_2$, but not simultaneously.

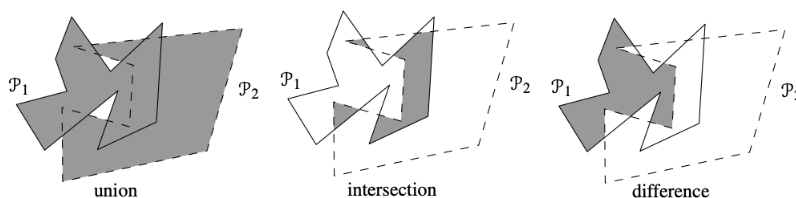


Figure 2.2: Boolean operations on polygons

Most algorithms for computing these operations which have been presented through history depends on finding the actual line segment intersections for all polygons. This has been most effectively been done through the sweep-line algorithm which we will now explain.

2.2.1 The Sweep Line Algorithm

The Sweep Line algorithm is fundamental in Computational Geometry algorithms. The idea of the sweep line algorithm is to imagine a line swept across a 2D plane.

This line stops at so-called **event points** where an operation is done, related to what one is computing of course. This is repeated until the program has traversed all event points.

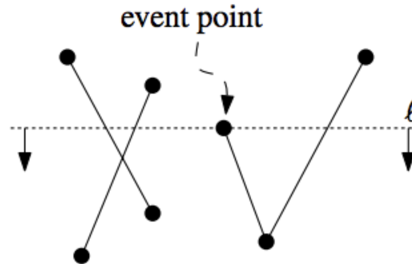


Figure 2.3: Sweep line

2.2.2 Line Segment Intersection

One of the simplest applications and a very useful such is the Line Segment Intersection algorithm. In this case, our inputs consist of segments that have one starting point and one endpoint. We will construct two fundamental data structures for this type of operation. An **event queue** $\mathbb{Q}$ and another structure to maintain the status of the algorithm, we call this $\mathbb{T}$ and this will keep track of what segments which are currently intersecting our sweep line. $\mathbb{Q}$ consists of all the event points in sorted order. For the simplicity of writing this report, we define the order which is used as follows.

Let p and q be two points in the gaussian space $p \neq q$, if p precedes q then $p_y \geq q_y$. In the case of $p_y = q_y$ then $p_x > q_x$.

Visually this means that the sweep line is traversing the plane in a north to south direction. In the case of when two event points have equal y values, the sweep line will momentarily traverse from west to east.

As mentioned above the segments consists of a start point and an endpoint. When our sweep line confronts a start point, it will add the segment to $\mathbb{T}$. When it confronts and end point, it will remove all segments which have an endpoint in $\mathbb{T}$. As will be seen later in the algorithms below, the status structure $\mathbb{T}$ is best implemented as a *self balancing search tree*.

The algorithm to find all intersections given a set **S** of segments looks the following:

Algorithm $FindIntersections(S)$

Input: Set of Line Segments S

Output: Set of Intersection points and corresponding segments.

1. Initialize an empty event queue $\mathbb{Q}$. Next, insert the segment endpoints into $\mathbb{Q}$, when an upper endpoint is inserted, the corresponding segment should be stored with it.

2. Initialize an empty status structure $\mathbb{T}$.

3. **While** $\mathbb{Q}$ is not empty.
   do Determine the next event point
   $HandleEventpoint(p)$

The *HandleEventpoint* algorithm looks as follows

Algorithm $HandleEventpoint(p)$

1. Let $U(p)$ be the set of segments whose upper endpoint is p; these segments are stored with the event point p. (For horizontal segments, the upper endpoint is by definition the left endpoint.)

2. Find all segments stored in $\mathbb{T}$ that contain p; they are adjacent in $\mathbb{T}$. Let L(p) denote the subset of segments found whose lower endpoint is p, and let C(p) denote the subset of segments found that contain p in their interior.

3. **if** $L(p) \cup U(p) \cup C(p)$ contains more than one segment **then**
   Report p as an intersection, together with $L(p)$, $U(p)$, and $C(p)$.
   end if

4. Delete the segments in $L(p) \cup C(p)$ from $\mathbb{T}$

5. Insert the segments in $U(p) \cup C(p)$ into $\mathbb{T}$. The order of the segments in $\mathbb{T}$ should correspond to the order in which they are intersected by a sweep line just below p. If there is a horizontal segment, it comes last among all segments containing p.

6. **if** $U(p) \cup C(p) = \emptyset$ **then**
   Let s_l and s_r be the left and right neighbors of p in $\mathbb{T}$
   $FindNewEvent(s_l, s_r, p)$
   else
   Let s be the leftmost segment of $U(p) \cup C(p)$ in $U(p)$.
   Let s_l be the left neighbor of s in $\mathbb{T}$
   $FindNewEvent(s_l, s, p)$
   Let s'' be the rightmost segment of $U(p) \cup C(p)$ in $\mathbb{T}$
   Let s_r be the right neighbor of s in $\mathbb{T}$
   $FindNewEvent(s, s_r, p)$
   end if

Algorithm $FindNewEvent(s_l, s_r, p)$

if s_l and s_r intersect below the sweep line, or on it and to the right of the current event point p, and the intersection is not yet present as an event in $\mathbb{Q}$ **then**
Insert the intersection point as an event into $\mathbb{Q}$

end if

It can be shown that this algorithm computes the complete set of intersections at a time complexity of $O(n \log n + I \log n)$, where n is the number of line segments and I number of intersections.

2.3 Overlay

We have now seen one of the most basic and fundamental algorithms of computational geometry. The next step towards computing a boolean operation of two sets of segments is to select a data structure that allows us to represent planar subdivisions in a proper way which we, of course, need to do when computing subsets of sets. The Doubly Connected Edge List will allow us to do so.

2.3.1 Doubly connected Edge List

A doubly connected edge list contains a record for each *face*, *edge* and *vertex* of the subdivision [4]. The vertex contains the information of the coordinates, the face record contains a pointer *OuterComponent(f)* which points to a half-edge on its outer boundary. The face record also stores a list *InnerComponents(f)* which for each hole of the face contains a pointer to half-edge on the boundary of the hole. The edge record simply stores pointers to *Prev(e)*, *Next(e)*, *Twin(e)* and a pointer to the face that it bounds *IncidentFace(e)*

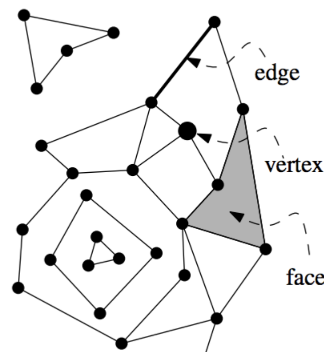


Figure 2.4: Doubly Connected Edge List

2.3.2 Computing an Overlay

Using what we have previously explained, we can now compute an overlay $O(S_1, S_2)$, where S_1 and S_2 are two planar subdivisions.

Theorem:

Let S_1 be a planar subdivision of complexity n_1, let S_2 be a subdivision of complexity n_2, and let $n := n_1 + n_2$. The overlay of S_1 and S_2 can be constructed in $O(n \log n + k \log n)$ time, where k is the complexity of the overlay. [4]

2.4 The Boolean operation

The Boolean operations OR, AND, DIF and XOR is simply a consequence and/or application from the overlay computation above. If we want to compute the union, $S_1 \cup S_2$ we extract all the faces in the overlay which are labeled with either S_1 or S_2. If we want to compute the intersection $S_1 \cap S_2$ we extract all faces labeled with both S_1 and S_2 and so on.

Corollary
Let P_1 be a polygon with n_1 vertices and P_2 a polygon with n_2 vertices, and let $n := n_1 + n_2$. Then $P_1 \cap P_2$, $P_1 \cup P_2$, and $P_1 \setminus P_2$ can each be computed in $O(n \log n + k \log n)$ time, where k is the complexity of the output. [4]

From this, we take away that the algorithm itself is output-dependent. The more intersections there are, the greater the time complexity. We also note that the time complexity is greater than linear, this means that theoretically if we have none or small overhead time. By splitting our operation into multiple parts, we could achieve lower time complexity without even running the program in parallel. Example: If number of line segments n = 20 and number of intersections I = 2, for the original problem we yield:

$$20log(20) + 2log(20) = 28,62$$

If we would theoretically split this into two parts, with 10 line segments in each part and 1 intersection each, we yield:

$$(10 \cdot log(10) + 1 \cdot log(10)) + (10 \cdot log(10) + 1 \cdot log(10)) = 11$$

There are quite a few different 2-D boolean operation algorithms used in the industry as of today. The line segment algorithm above, leading to the boolean operation was presented by Bentley and Ottoman in 1979 [3] which as mentioned, runs at $O(n) = n \log n + k \log n$ time. Improvements have since been made and most significantly by Ivan Balaban in 1995 who presented an algorithm that runs in $O(n) = n \log n + k$ time [2] for any types of polygons. Based on what assumptions one can make about the polygons in use one can even improve the time complexity further. For example, the Sutherland-Hodgman algorithm can be done in constant time given that all polygons are convex [7]. A specific boolean algorithm was not chosen for this thesis, instead the proprietary algorithm developed specifically for chip design rule check by Silvaco Inc. was the aim to use by the start of this thesis. As described later on, tests were instead done using the Clipper c++ library, which implements a version of the Vatti Clipping Algorithm [8], which runs a far more general approach.

2.5 Decomposition: Partition the Boolean Operations

We have now seen that the three different Boolean operations OR, AND and XOR all can be computed with the time complexity of $O(n \log n + k \log n)$. In this section, we examine possible ways to decompose our original, sequential problem into one which can be run in parallel. We will also quickly list consequences and eventual issues we might run into.

2.5.1 Binary Space Partitioning

Binary Space Partitioning is a way of partition multidimensional(N) space using recursive subdividing by hyperplanes. A hyperplane is a plane of dimension (N-1). In fact, Binary Space Partition really is only a generalization of normal binary search trees and can be seen as a geometric version of quicksort, as will explain next. A Binary Search Tree is, of course, a linked data structure made of nodes where each node has a nullable "left" and "right" reference, starting with a non-referable node called the root.

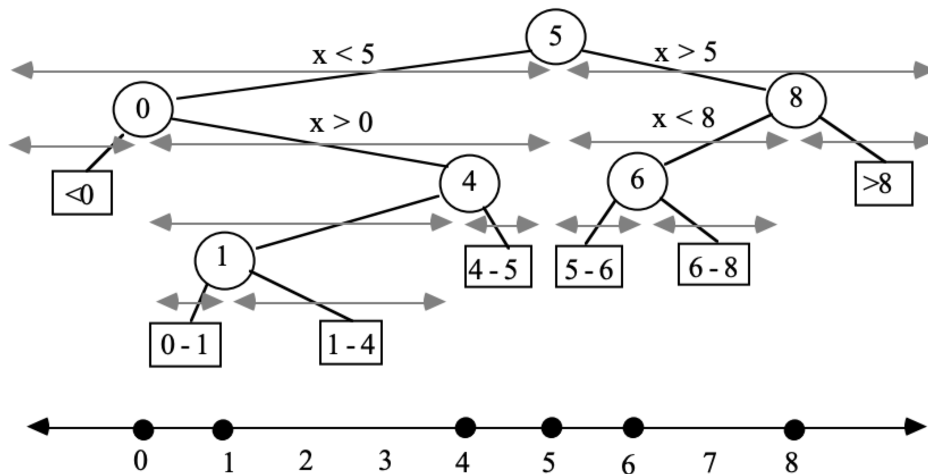


Figure 2.5: An example of a binary search tree in one dimension representing the set of integers $S = \{0, 1, 4, 5, 6, 8\}$

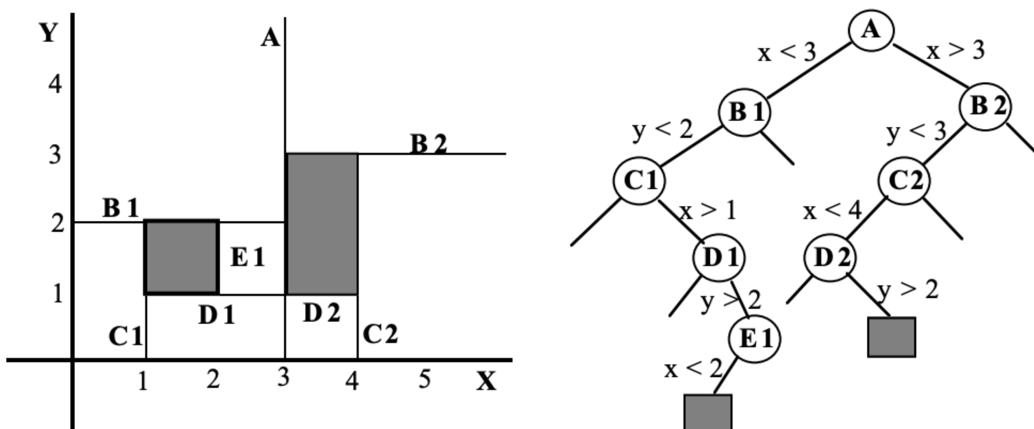
BSP-trees again takes this concept to multiple dimensions. A hyperplane always divides a region into two halfspaces regardless of dimension. In 1-D they are seen as points as seen above in figure 2.5 and in 2-D they are represented by lines. In Binary Space Partition quicksort is used recursively with dimensions alternating throughout the process to decide the position of the hyperplanes, making sure every region which is split contains equal or as close to an equal amount of points as possible. E.g. in two dimensions, one would first sort the given set on points based on e.g. the value of the x-coordinate, select the point p_{x_1}, which has

the median x-value and becomes the first hyperplane. This will also be the value belonging to our root node. As a result, we now have two new sets of points, S_1 which contains all points which $x < x_1$ and S_2 which contains all points $x > x_1$. The process is now repeated on these sets but now instead we sort according to the y-values of the points. The median value of S_1 will then belong to the node pointed to by the "left"-pointer by our root node. Respectively the median value of S_2 will belong to the node pointed to by the "right"-pointer. In pseudo-code, the algorithm looks the following:

Algorithm $BuildBSPTree(points, dim, p_{min})$
if nof points < p_{min} **then**
return
end if
if dimension == x **then**
sortX(points) and dim = y
else
sortY(points) and dim = x
end if

Node n
n → point = median(points)
n → left = BuildBSPTree(points < point, dim, p_{min})
n → right = BuildBSPTree(points < point, dim, p_{min})
return n

The time complexity of building the tree is of course, the same as quick sort due to its similar nature worst case $O(n^2)$ and average $O(nlog(n))$



The other half of the coin is of course, given a point p, find what region p belong to. This Algorithm looks the following: **Algorithm** $FindRegion(p, dim, bspNode)$
if dim == x **then**
if $p_x < bspNode_x$ **then**
if bspNode → left == null **then**
return this region

```
        else
            return FindRegion(p, dim = y, bspNode → left)
        end if
    else
        if bspNode → right == null then
            return this region
        else
            return FindRegion(p, dim = y, bspNode → right)
        end if
    end if
else
    See above but switch x and y.
end if
```

And we can easily see that the time complexity of traversing the tree is $O(n)$

Some points to keep in mind and/or explore are the following

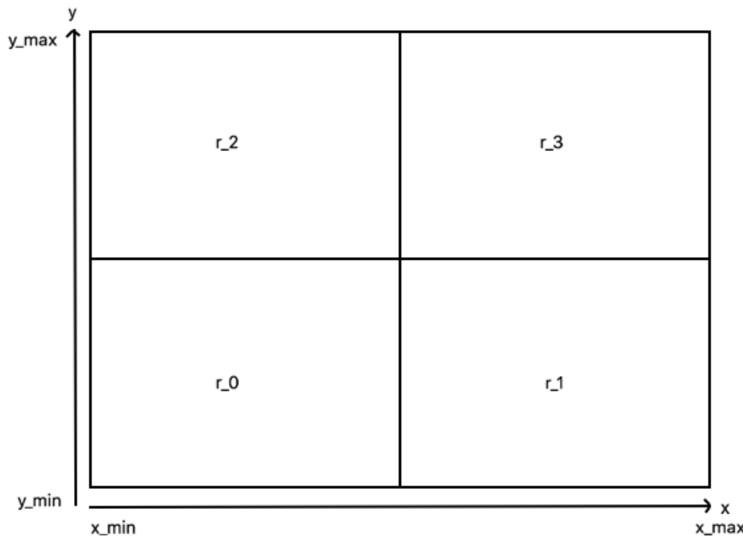
1. Similar to quick-sort we should not completely sort the vector of points when building the tree, only relative to the pivot element.

2. How exact do we need to be when partitioning our region? I.e. Might we reach a point where we "over-partition", wasting too much time on getting regions exact without getting anything in return?

3. To what extent is this process parallelizable?

2.5.2 Median-based Partitioning of 2D-space

Another way to represent regions in the 2-D space is a matrix-like representation such that the index of a region $R = x_i + y_j \cdot x_{dim}$. To calculate to which region a point p belongs to, a binary search tree has been implemented and used. A BST built from a sorted vector retrieved at pre-processing will take $O(n)$ time and will later be used repeatedly. The building time will in other words by no means affect the time complexity of the whole algorithm in the long run. However, this way of partitioning will not guarantee that the polygons are evenly partitioned, or even that a region even has a polygon in it at all.



2.5.3 Consequences of Partitioning

Dividing the 2D-space into N regions geometrically does not come without consequences. These newly created boundaries of our regions are at risk of splitting a polygon, causing different kinds of problems. Let us say N = 4 (as in the figure) and an arbitrary polygon P and list the available scenarios.

1. All points of P are inside the same region R_i.
2. P have points in both regions R_i and R_j
3. P have points in both regions R_i and R_j and intersects polygon P_2 which lies in R_i or/and R_j
4. P have points in both regions R_i and R_j and intersects polygon P_2 which lies in R_i or/and R_j, P_2 does in turn intersect another polygon P_3 which does not intersect P. In theory, this may happen as many times as there are polygons in the region of P_2. We call this the cascading problem

It follows from point 4 above that a polygon which has a point in both region R_i and R_j have dependencies to all polygons in those regions.

Challenges/Issues

Some issues arise when we geometrically divide the space that we must handle. The first issue is the cascading problem explained above. It means that we can never exclude the possibility that a small polygon in the south-west corner of the 2d-space has a dependency to a small polygon in the north-east corner, even if they do not intersect. This is since they might intersect a chain of polygons and

they might end up as the same polygon in the result.

The next issue is the rounding/grid problem. Since we are doing our computations on a grid, built up by Integers, we are prone to rounding problems. There is however a clear difference in having a small rounding problem as a part of our original operation, which is a must when working in the digital world. Let us instead assume we have a polygon with an edge which is at a degree other than 0, 45 or 90 degrees. If we would cut this edge off, we risk changing the input to the Boolean operation itself and thereby risk getting a faulty result.

2.6 Assignment

We need a way to assign tasks to threads. Two general methods were briefly mentioned; Fixed assignment and Dynamic assignment. In a fixed assignment, the tasks a thread will perform is determined at the spawning of the threads, and will not change during the time of computation. An example of a fixed assignment is when a thread is responsible for computing a certain number of rows in a matrix. Dynamic assignment, on the other hand, uses a shared task pool where threads continuously pull its next task, once it is finished with its last one. The advantage of a fixed assignment is that the threads no longer have a shared resource and are, at least during the computation phase completely independent of each other. If the tasks are predictable in computation time, such as operations on a matrix-row, the may very well be the superior technique. This, however, makes one prone to load imbalance. If the tasks receive by thread1 takes longer time than the tasks received by thread2, there is a clear load imbalance and there is nothing we can do on the fly. When using dynamic assignment, we are more resilient to load imbalance. This is due to the fact that if thread1 gets unlucky and receives a very time-consuming task and thread2 finishes ahead of time, thread 2 will pull another task from the task-pool until the task-pool is empty. So by the time thread1 completes its task thread2 may be completing its second or even third task. The problem with dynamic scheduling is of course that we need a shared task-pool where we risk thread starvation. We will next up elaborate on different techniques one may use to build this dynamic assignment task-pool – Thread scheduling.

Thread Scheduling

Thread scheduling is a method to assign units of work to resources that complete the work, examples of resources are treads and/or processors. The main goal of a thread scheduling method is to utilize all the resource to the max, and in other words; to balance out the work to be carried out, i.e. load balancing. Different methods are the following;

- FIFO: First Come First Served
- Priority Scheduling

- Round Robbin Scheduling
- Shortest Remaining Time First

FIFO - First come, first served

The simplest scheduling algorithm. One simply puts the units of work in a queue and then lets the available threads consume jobs from the queue. The main advantage of this algorithm is that the overhead is minimal compared to other algorithms, context switch between threads should only occur when a job is starting or is done. The main disadvantage is that since not prioritization is done between the jobs if there are deadlines there will be a problem meeting these.

Priority Scheduling

A scheduling method mainly designed for meeting deadlines. The unit of work which has the closest deadline is simply put first in the queue and will be executed thereafter.

Round Robbin Scheduling

This method assigns every unit work slices of time in a cyclic fashion. No priority is done between the units of work.

Shortest Remaining Time first

This places the units of work with the shortest (or longest) time first. The main objective is to increase throughput in a scenario where every unit of work gives the same reward. This is, of course, similar to Priority scheduling and requires extensive knowledge of how much time or computing power each job requires to complete.

2.6.1 Divide and Conquer

The algorithmic approach know as divide and conquer is useful when one can break down a large original problem into subproblems which are similar to the original problem, but smaller in size. This approach consists of three steps:

- Divide: Partition the problem into subproblems
- Conquer: Solve the subproblems
- Combine: Assemble the solutions of the subproblems into the solution of the original problem

The key is that when the time complexity of the original problem is larger than linear, one can benefit greatly from splitting it down to subproblems, given that the time of the divide and combine steps does not grow equally as the time reduced by the conquer step. The divide and conquer approach works naturally well together with parallel computing [5].

2.7 Orchestration

The orchestration, i.e. the implementation itself, will be described in the next chapter in more detail. Based on this theory section, implementation details have been made and implemented. Task scheduling, ways of satisfying dependencies and reducing the amount of access to shared data have been the goal. Given this, three partitioning strategies, which meets the problems/issues formulated earlier in this chapter have been proposed and evaluated.

Chapter 3
Method

I this chapter, we present the methods we have used in order to answer the problem formulations. We also present three partition strategies that have been implemented and tested and go into some small detail on how the implementation has been done.

3.1 Partition Strategy 1 - Separate Layer

This strategy aims to avoid non-trivial boundaries of polygons by creating a "middle part" where we put polygons that have segments that cross any border between different parts. Let us say we split all our input into i parts, we then get $i + 1$ different Boolean operations (+1 because of the middle part) which can be computed independently and which are clearly parallelizable. However, the big disadvantage of this strategy is that we while the i different parts are guaranteed to not interfere with each other and can simply be put together at $O(n)$, the "middle part" has to be performed as an argument in a final union together with the rest of the parts to get the final result. Algorithmically the strategy looks the following:

for all InputLayers **do**
  while InputLayer has more polygons **do**
    if All points in polygon p belongs to same part i **then**
      Put p in part i
    else
      put p in middle part
    end if
  end while
end for

for all partitions **do**
  Boolean Operation on partition i. $B_i(L_1, L_2) \rightarrow O_i$
end for
Create a temporary layer $\mathbb{L}$ to put above outputs.
for all partitions **do**
  Put all O_i into $\mathbb{L}$
end for
Boolean Operation on middle partition. $B_m(L_1, L_2) \rightarrow O_m$

Boolean Operation on $\mathbb{L}$ and O_m. $B(\mathbb{L}, O_m) \rightarrow O$
return O

Mathematically, the strategy looks the following

$$O(N \log N + K \log N) \xrightarrow{\text{strat1}} \tag{3.1}$$

$$\sum_{i=1}^{N} O(n_i \log n_i + k_i \log n_i) + O(n_m \log n_m + k_m \log n_m) + O(n_F \log n_F + k_F \log n_F) + OH \tag{3.2}$$

where:
N = number of segments in total $(N_1 + N_2)$
K = Complexity of the output.
n_i = number of segments in the $i:th$ part
k_i = complexity of the output in the i:th part
n_m = number of segments in the middle part
k_m = complexity of the output in the middle part
n_F = number of segments in the final operation.
k_F = complexity of the output of the final operation
OH = Overhead. The time it takes to place polygons into different parts and put them back together.

As mentioned the first and second terms in the equation above are parallelizable. However there exists a dependency between the third term and the first two ones, and therefore the third term must be computed sequentially after the first two ones.

Just to quickly analyze what the computation time of each term depends on we go through them one by one.

$$\sum_{i=1}^{N} O(n_i \log n_i + k_i \log n_i) \tag{3.3}$$

This term simply depends on how many points/polygons we can manage to place in a region. That is exactly the same amount of polygons that do not cross any border between regions. Differences between each part can arise due to bad load balancing. E.g. region 1 contains 100'000 polygons while region 2 contains 1000 polygons, this, however, will have more consequences once we try to parallelize the algorithm.

$$O(n_m \log n_m + k_m \log n_m) \tag{3.4}$$

This term depends on how many points/polygons that do cross any border and are therefore put in the middle region.

$$O(n_F \log n_F + k_F \log n_F) \tag{3.5}$$

The most interesting and impactful term. This term depends on the output of the middle region, combined with all the polygons that risk intersecting the polygons in O_m.

Let us say that we cannot eliminate any polygon $p \in \mathbb{L}$(see algorithm above). n_f is then close to N. It follows that the whole operation $O(n_F \log n_F + k_F \log n_F)$ will be close to $O(N \log N + F \log N)$. We conclude that if we do not make an effort to remove polygons in $\mathbb{L}$ that we can guarantee does not intersect polygons in O_m, there cannot possibly be any speed-up through a parallelization.

3.1.1 The bounding box

The bounding box is a concept to create an area $\mathbb{B}$ that represent the following:

If a polygon p has it's complete set of points laying outside $\mathbb{B}$, we can guarantee that p will not intersect any of the polygons lying completely inside $\mathbb{B}$.

Examples of how to construct this area can be one of the following:

1. A simple square: $x : [\min x, \max x]$, $y : [\min y, \max y]$
2. The convex hull
3. Multiple boxes. Each box representing a border(snitt) between regions.

It is clear that we want to minimize $\mathbb{B}$, thus minimizing (2.5). However we must explore the time complexity of constructing $\mathbb{B}$ and checking however a polygon $p \in \mathbb{B}$.

The first case, the square, it is obvious that constructing the square takes $O(N)$ and to check whether a point $p \in \mathbb{B}$ is also $O(N)$. ($O(4N)$)

The convex hull takes $O(O_m^2)$ time to construct [4] and since we get a polygon with undefined number of edges, we must use a more sophisticated algorithm to

check where p lies inside. Rays casting algorithm is an example and takes $O(n)$ time, where n is number of points we want to check.

The third case is similar to case 1, but with one box per border. This results in that the shape of our boxes are still simple, and the check of each box can still be done in $O(n)$ time.

3.1.2 Parallel case

As mentioned before, looking back at eq. 3.2. There are no dependencies between the terms in (3.3) or the term (3.4) and these parts can therefore parallelized, however we consider the last part non-parallelizable. We can therefore see that this method is heavily sensitive to the computing time of the term 3.5

3.2 Partition Strategy 2 - Duplication

3.2.1

This strategy aims to solve the problem of complex boundaries by duplicating every polygon which simply crosses a border, and thereafter include the polygon in every region it crosses. The issue of cascading polygon-dependencies is solved as well using this strategy. A merge operation will have to take place in order to get the correct solution. This is due to the fact that it is now possible for the result of multiple regions to interfere with each other.

for all InputLayers **do**
  while InputLayer has more polygons **do**
    regions $\leftarrow$ []
    for all points pt in polygon p **do**
      regions add region of pt and all regions crossed between r_{prev} and r
    end for
    Add polygon p to all regions in regions
  end while
end for
Create a temporary layer $\mathbb{L}$ to put above outputs. Create a temporary layer $\mathbb{M}$ to merge polygons needing merge.
for all partitions **do**
  Boolean Operation on partition i. $B_i(L_1, L_2) \rightarrow O_i$
end for

for all partitions **do**
  for all polygons **do**
    if polygon has point in region $\neq i$ **then**
      Put p in $\mathbb{M}$
    else
      Put p in $\mathbb{L}$

end if
end for
end for
Merge/OR Operation on $\mathbb{M}$. $B_M(L_M) \rightarrow O_M$
for all polygons in $\mathbb{M}$ **do**
Add p to $\mathbb{L}$
end for

Mathematically, the strategy looks the following

$$O(N \log N + K \log N) \overset{\text{strat2}}{\Longrightarrow} \tag{3.6}$$

$$\sum_{i=1}^{N} O(n_i \log n_i + k_i \log n_i) + O(n_F \log n_F + k_F \log n_F) + OH \tag{3.7}$$

where:
N = number of segments in total $(N_1 + N_2)$
K = Complexity of the output.
n_i = number of segments in the $i : th$ part
k_i = complexity of the output in the i:th part
n_F = number of segments in the final merge
k_F = complexity of the output of the final merge
OH = Overhead. The time it takes to place polygons into different parts and put them back together.

Similar to the previous strategy the first term

$$\sum_{i=1}^{N} O(n_i \log n_i + k_i \log n_i) \tag{3.8}$$

is similar to partitioning strategy 1, it will however contain marginally more segments due to the fact that we duplicate polygons which crosses multiple regions, compared to removing them from the part as we do in strategy 1. The second term

$$O(n_F \log n_F + k_F \log n_F) \tag{3.9}$$

may be the most interesting part of this strategy. From the outputs of equation 3.8, we check each polygon if it has at least one point outside its supposed region if it has we need to merge it with other polygons with this property. If it has all its points in its supposed regions we put the polygon directly in the solution due to it cannot possibly intersect any other polygon and we can guarantee that it does not intersect any other polygon. The term depends heavily on the following:

1. Number of regions
2. The span/size of each input polygon
3. The output

1. An increase of the number of regions increases the number of borders, thereby increasing the chance of a polygon spanning multiple regions

2. The shape of each individual polygon will, of course, have an impact on how many regions it covers. A polygon p_1 stretching across all of the computing areas will belong to more regions than a polygon p_2 which has its points concentrated close to each other. This is even if p_2 may have many more points.

3. Related to 2, If the result simply is one large polygon. e.g. an OR operation on many overlapping polygons that forms one or a few outputs, there simply are big dependencies between the different regions which one cannot avoid and all the points/polygons must be merged.

3.2.2 Problems

The duplication of polygons creates a vital problem. Due to our limitation of treating a Boolean operation as a complete operation/black box, there arise issues when doing what can be called the asymmetrical operations. Examples are the DIF and XOR operations.
While in the OR and AND cases we only may get duplicate results, e.g. two exact same polygons that we can identify and merge. The XOR and DIF operations may give us incorrect solutions as seen in figure 3.1. Therefore, this partitioning strategy cannot be used with the XOR nor DIF operations.

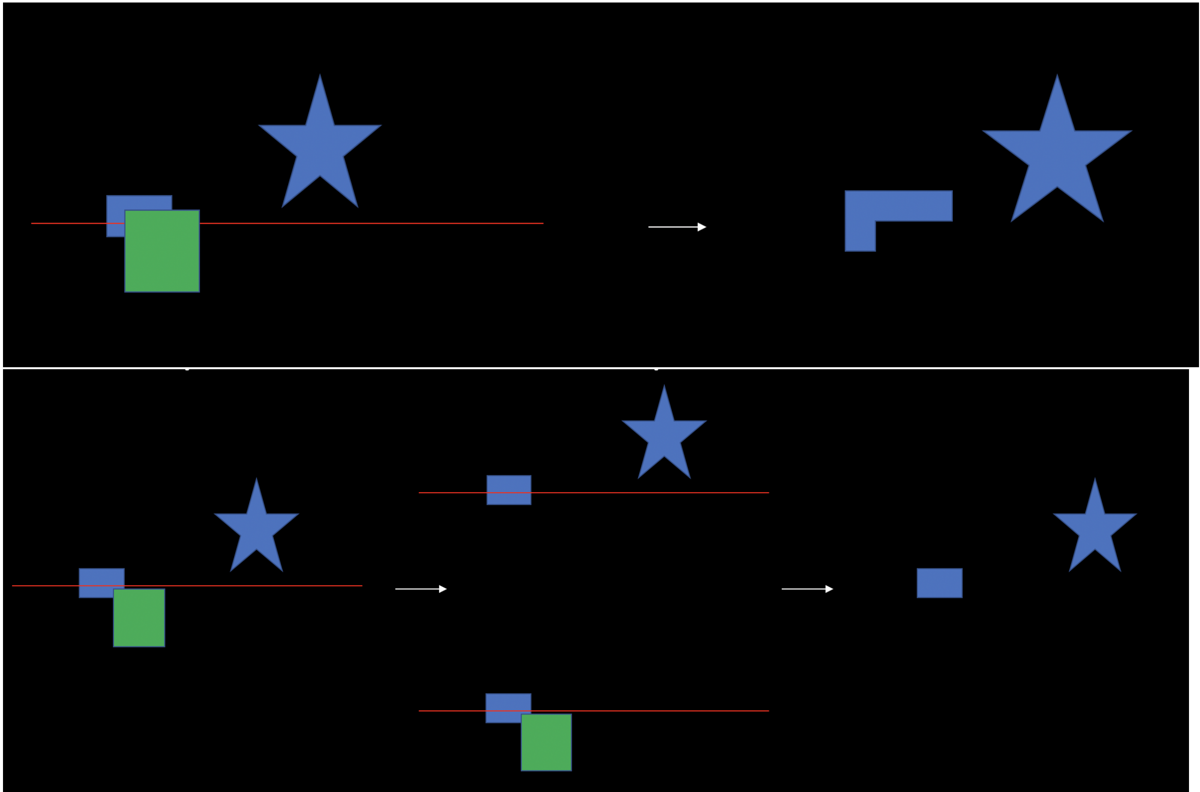


Figure 3.1: Top: Correct DIF operation. Bottom: A faulty DIF operation

3.3 Partition Strategy 3 - Splitting

The third strategy may be conceptually the simplest strategy. We simply split the polygons to multiple parts according to what regions the part is in. This could example be done by performing an AND operation between the polygon and region in question. By splitting the polygon into multiple parts, the resulting sub-polygons will have at least two points on the border of a region which will work as an identifier when we, as in strategy 2, need to merge the different regions to the final result. Since we in this strategy partition our polygons in a geometrical way we will not have the same problem with incorrectness when performing the XOR/DIF operations. We will, however, see that another challenge will arise. Since we are performing our operations on a grid, where polygons are defined as an array of points. By cutting off a line segment, we risk getting a rounding error that will make it such that the input does not exactly equal the input when split up in different parts. This may, of course, impact the result in a severe way in the worst case.

for all InputLayers **do**
  while InputLayer has more polygons **do**
    regions ← []
    for all points *pt* in polygon *p* **do**

regions add region of pt and all regions crossed between $r - prev$ and r
end for
for all r in regions **do**
Add sub-polygon of polygon p which intersects region r to r
end for
end while
end for
Create a temporary layer $\mathbb{L}$ to put above outputs. Create a temporary layer $\mathbb{M}$ to merge polygons needing merge.
for all partitions **do**
Boolean Operation on partition i. $B_i(L_1, L_2) \rightarrow O_i$
end for

for all partitions **do**
for all polygons **do**
if polygon has point on its region's border **then**
Put p in $\mathbb{M}$
else
Put p in $\mathbb{L}$
end if
end for
end for
Merge/OR Operation on $\mathbb{M}$. $B_M(L_M) \rightarrow O_M$
for all polygons in $\mathbb{M}$ **do**
Add p to $\mathbb{L}$
end for

Similar to the previous strategy our time complexity looks the following:

$$O(N \log N + K \log N) \xrightarrow{\text{strat3}} \tag{3.10}$$

$$\sum_{i=1}^{N} O(n_i \log n_i + k_i \log n_i) + O(n_F \log n_F + k_F \log n_F) + OH \tag{3.11}$$

where variables are the same as in strategy 2. The major difference compared to strategy 2 is that there is slightly more overhead when partitioning the polygons and we are forced to actually perform a splitting algorithm on all polygons which belongs to multiple regions. In strategy 2 we only need the information about which regions a polygon belongs to.

Regarding dependencies, they are similar to strategy 2.

3.3.1 Problems

The major shortcoming of this strategy is as mentioned that there may arise rounding problems before that the computation itself takes place which may pro-

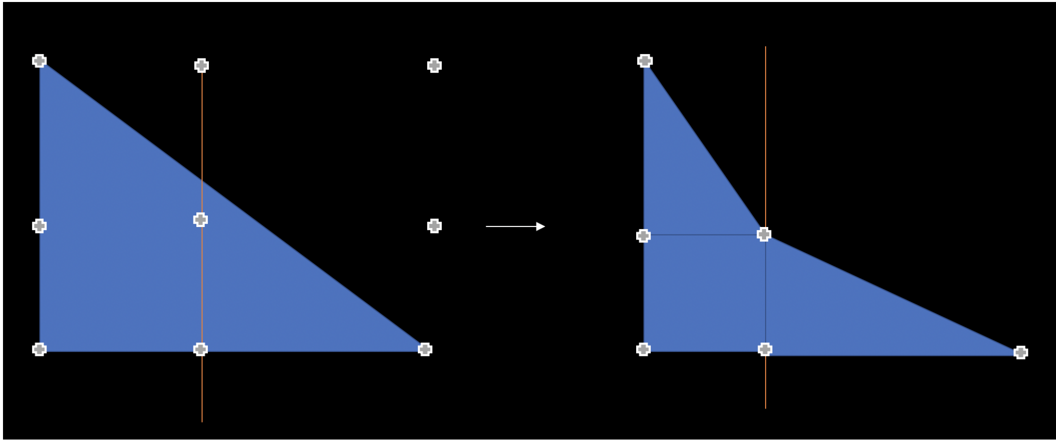


Figure 3.2: A faulty split of a polygon

duce an incorrect result. This scenario may only happen when there is a line segment which has an angle between point p_1 and p_2 which differ from 0, 45 or 90 degrees. The critical problem arises when two polygons that should produce a result, but due to the alteration to one of the polygons because of this rounding error, another result is produced. In figure 2.3 an example of this alteration is shown.

3.4 Implementation

The implementations of the different strategies have been done. Work has been done in the Silvaco product Guardian which is used to perform design rule checks of chip designs. An API to Guardian has been developed by the supervisor Björn Wehlin in order for this project to develop C++ plugins, replacing/overriding the standard boolean operations and allowing us to write our own custom implementations.

The goal at the beginning of this project was to still use boolean methods in Guardian via the API but to use the partition strategies described in the previous chapter and then use Guardians boolean methods in parallel once the partitioning has been done and we can spawn multiple threads. This, however, turned out not possible due to problems with thread safety issues in Guardian.

In order to continue the project, open source library was looked into. First boost geometry, which is a well-known library in the geometry paradigm. The boolean operation of the boost library is based on the Weiler-Atherton [9] algorithm. However, problems did arise again here due to when returning the result to the Guardian API, the result must be on a specific format, basically, the points making up a polygon must be in positive orientation and the holes must be in negative orientation, inserted in the polygon the hole belongs to. The issue with boost geometry was that, while the orientation requirement was fulfilled, one was unable to tell

which polygon each hole belonged to. One could, of course, check manually, however, this was deemed taking to much time computationally and we moved on.

Another open source library eventually used was the Clipper Library. The Clipper library implements the Vatti Clipping algorithm, which in turn is described in the theory section. Here we were also able to receive the holes as child elements of the polygon they belonged to, which made both life easier as well as speeding up the operation.

During the progress of the implementation and due to the nature of the time complexity it was made clear that the algorithm would likely benefit from increasing the number of partitions even though the number of processing cores was limited. This is due to the time complexity of the boolean operation is not linear, interest did also arise whether it was possible to partition the operations enough to run the whole operation on a GPU.

3.4.1 BSP-tree

The binary space partitioning tree was implemented. When sorting the points based on the X dimension, since the next step down in the recursive tree will then sort based on the Y dimension there is no reason to fully sort the points. Since we will split the points into two different parts we only need to make sure all points to the right of the pivot element has an X larger the X of the pivot element, and respectively for the left part. The building time can be greatly decreased this way. An attempt to parallelize the building process of the BSP-tree but due to the shared resource, the tree itself being a bottleneck, this was discontinued. Later on, when doing the partitioning of the polygons itself, due to the fact we always get two independent input layers, we were able to run the partition of their respective polygons in two parallel threads.

3.4.2 Thread-Scheduling

In the theory section, we proposed several thread-scheduling methods. The goal of our thread scheduling is to reduce the time of thread starvation, i.e. the time threads are left unused. The most likely reason for this is when there is one job that takes significantly longer time than other jobs, and one thread is left at the end as the only thread working. It is also desirable to reduce the amount of context switching.

Given that our computing is not related to any deadline of any sort, there is no requirement for a deadline-based priority scheduling. It would be interesting to use priority scheduling, prioritizing based on the longest remaining time left to avoid thread starvation at the end. This, however, requires us to know how much time each job will require on beforehand. It was determined it was more effective to focus on making the partitions equally sized than attempting to estimate time

of jobs. For our implementation, a FIFO-queue was therefore chosen.

Now, if the number of partitions is less than the number of threads used, this will of course lead to thread starvation. If the number of partitions is significantly larger than the number of threads, it is reasonable to assume that the chance of thread starvation at the end will be small. Since we reduce the average time of computation per job, even if the last job accidentally is the largest, the time of these jobs are small in comparison to the overall computation. However, by increasing the number of partitions too much we risk increasing the time of the partitioning itself and the merging operations as previously explained.

Chapter 4
Evaluation

In this chapter, we present results of stress tests done by the end of the time-frame of this thesis as well as insights received during the progress of the work.

4.1 Cases

Test cases were generated by application engineers at Silvaco so that the cases used would be real-life examples. When doing benchmarks, examples reflecting different challenges were selected to gain a broader test surface. To simulate larger inputs, cases have also been stacked together in a matrix-like form. For each case, we will do tests with each of the operations OR, AND, DIF. The correctness of our results have been verified by using layout vs layout(LVL) tool, which can be found in Silvaco's program. This tool compares the result of the parallelized operation with an original operation and simply returns yes or no, depending on the result.

4.1.1 Case A

The first case: Two layers called poly and active as shown in figure 4.1. characteristics of this case are that both input-layers consists of relatively small components which do not stretch across the full chip. The AND operation between these layers is a real case in the industry today and is used in nearly every design rule check. We have on this case therefore done more stress tests than other cases. A test-suite where an 8x8 stacked version of this was run to simulate larger inputs. This was however not very practical due to the total time of this suite consumed over a whopping 13 hours as seen.

- Number of points: 4 736 148

- Number of polygons: 578 351

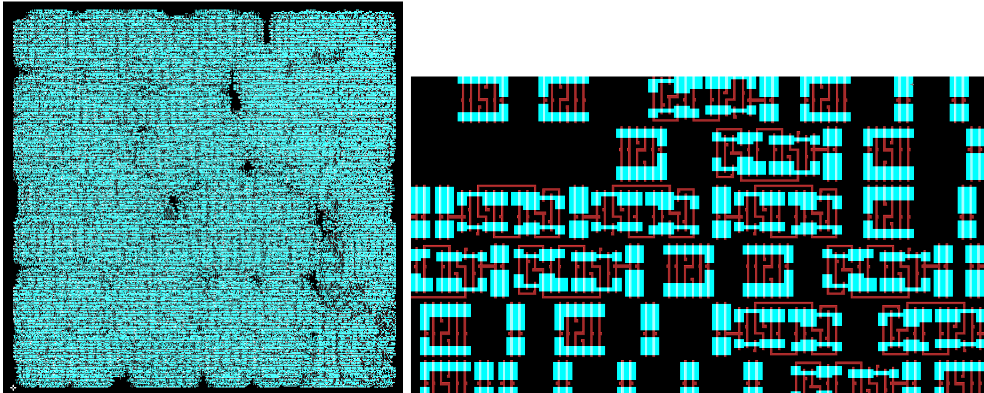


Figure 4.1: Layer1: poly, Layer2: active. Full input left, zoomed in right.

4.1.2 Case B

The second case, shown in figure 4.2. Characteristics of this case, or especially of the layer metal1 is the power rails crossing the full chip. This causes many dependencies across many of the partitions due to these power-rails will likely stretch across multiple regions.

- Number of points: 9 638 190
- Number of polygons: 1 698 577

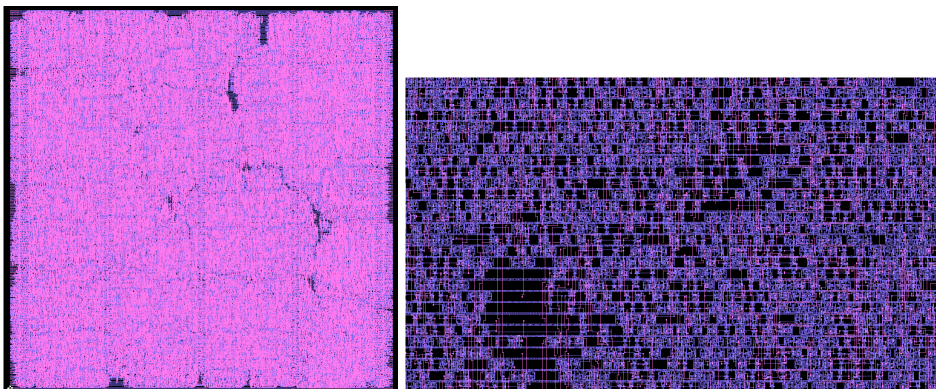


Figure 4.2: Layer1: poly, Layer2: active. Full input left, zoomed in right.

4.1.3 Case C,D

Case C was between metal 2 and metal 3. Characteristics are that metal2 and metal3 are completely perpendicular to each other. Chip-components are large

and often stretch across large parts of the chip. Many intersections of the layers are to be expected.

- Number of points: 4 407 264
- Number of polygons: 1 101 816

Case D was between metal 3 and metal 5. This case is a significantly smaller case than in previous cases. The layers metal3 and metal 5 are generally parallel to each other. Not many intersections are to be expected.

- Number of points: 1 637 588
- Number of polygons: 409 397

4.2 Results

Tests have been done and measurements of time have focused on the algorithm itself, and have therefore omitted the time of importing and exporting the data from Silvaco's program Expert. This as well as the time is taken to convert the format used in Expert to the format used by Clipper. Furthermore, The measurements have then been divided into phases: The time taken to partition the polygons into the different partitions, ready to be executed in separate threads, this phase is tagged TPartition. We then have the time actually spent in threads; TThreads. We then measure the time taken to bring all these independent operations back together to compose the final, correct solution. This phase is called TMerge. Finally we of course measure the total time of the whole algorithm, this is tagged TTotal. These detailed results can be seen in the Appendix due to the vast amount of the results. In this chapter, we choose only to display the overall running times for every combination of cores(marked as C) and partitions(marked as P).

4.2.1 Case A - poly, active - AND

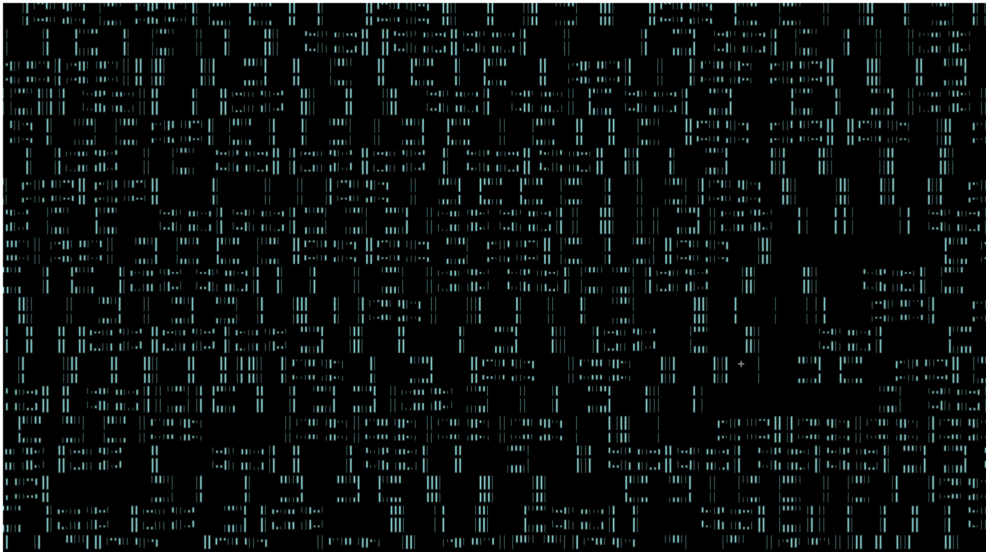


Figure 4.3: Result of AND, case A

The first stress tests were done this case A with the operation AND. The actual results can be seen in the figure above. As we can see, the result consists of many small polygons which logically results in that relatively few polygons need to be merged when sewing the partitions back together.

In table 4.1 we see the results of the AND operation on case A with a normal version of the input. We identify that the best result was received when using 64 partitions and 8 cores with a time of 1,38 seconds. Baseline was done by simply reading the input and performing the AND operation directly on the input without any modifications. Baseline came in at 29.53 seconds. Tests were also done for partitions up to 4096 but have been omitted for clarity since they follow the pattern of getting worse and worse the more partitions used on this input. Figure 4.4 visualizes these same results and how the computation time depends on the number of threads, for every number of partitions. We note that 8 processor-cores seem to be a hot spot. Any number of partitions between 64 and 128 seem to perform well. Tests running on 32 cores seem to generally run slower than 16 cores. Detailed results on how much time was spent in each phase on every combination can be found in Appendix A.1. We can see that the top result of 1,38 seconds, 0,4 seconds were spent partitioning and 0,26 seconds spent merging, this sums to 0,665 seconds and make up around 48,1 % of the total running time and is a variable we will keep track of.

A summary of how the time of partitioning and time of merging depends on the number partitions is shown in figure 4.4. We can see that up to 256 partitions, the time of both partitions and merging grows mildly, which is reflected in table 4.1 as well. After this, at 512 partitions both time of merging and partitioning

takes almost 1 second each. Compared to the top result in this run, 1,38 seconds this, of course, makes it impossible to beat the best result, no matter how fast the computation-phase runs. We conclude that above 256 partitions we have clearly over-partitioned. It is though important to note that we still run significantly faster than the baseline: 29,53 seconds.

Cores	**Partitions**						
	16 P	32 P	64P	128P	256P	512	1028P
1C	5,30	3,82	3,57	3,51	3,98	5,69	10,26
2C	3,01	2,31	2,10	2,07	2,39	3,85	6,72
4C	1,99	1,60	1,48	1,49	1,71	2,83	5,05
8C	2,15	1,55	**1,38**	1,50	1,66	2,27	4,14
16C	3,12	2,14	1,72	1,45	1,56	2,46	4,30
32C	2,92	4,08	2,23	1,98	2,03	2,83	4,40

Table 4.1: AND operation on case A with duplicating partitioning. Baseline execution time: 29,53 seconds

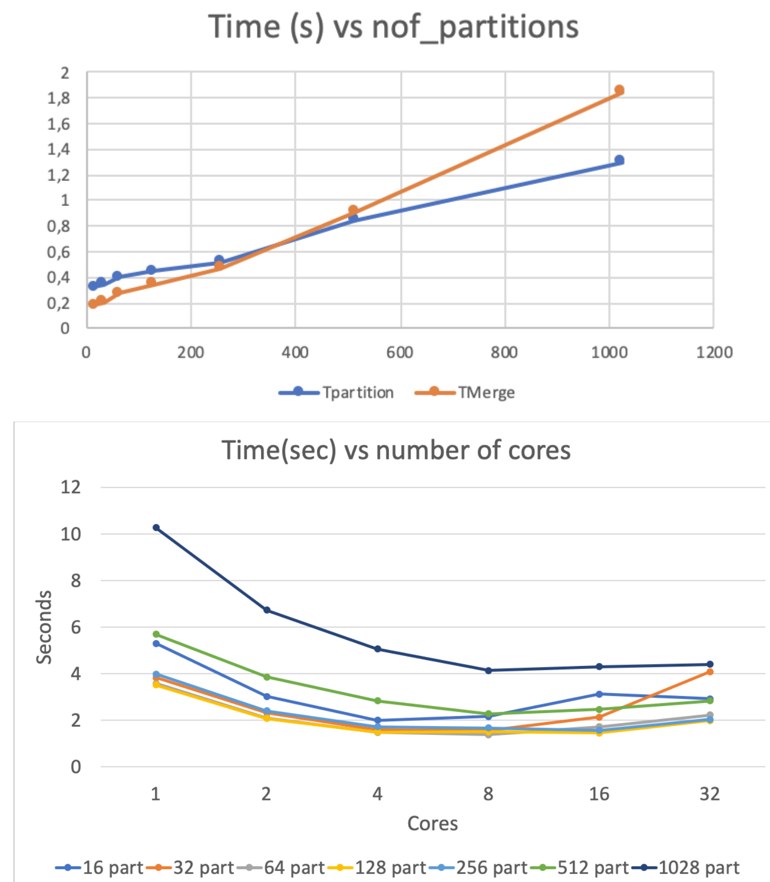


Figure 4.4: Upper: Time of partitioning and Merging depending on number of partitions.
Lower: Total time depending on number of cores. Duplicating Strategy, case A

In table 4.2 we see the results of the AND operation on case A with a large version of the input. We note the top result of 87,54 seconds when running on 16 cores with 512 partitions. Comparing this to the baseline of 22745,37 seconds we can see a speedup of 260X speedup. The difference between the top result and other results which are being partitioned with 512-2048 partitions and running on many cores(>16) becomes negligible compared to the baseline. We can also see that the worst result is taken, running on 1 core with 16 partitions which runs at 4799 seconds still has a speedup of 4,7X compared to the baseline. Detailed results can be shown in Appendix A.2 and A.3 and we can see a similar pattern as in the small version of this test. Once we reach 4096 partitions, we can see that the time of partitioning and merging starts to consume a percentage of time spent on their respective tasks. The top result of 87,54 seconds spends 29,88 seconds partitioning and 21,07 seconds merging. This adds to 50,95 seconds and is 58.2% of the total running time which is the non-parallelizable part.

Cores	**Partitions**								
	16 P	32 P	64P	128P	256P	512P	1028P	2048P	4096P
1C	4799,00	2049,81	1998,37	813,76	743,82	487,96	425,65	336,35	340,12
2C	2649,38	1050,49	1006,98	425,05	377,11	249,89	228,88	190,96	196,50
4C	1347,89	541,52	534,25	229,40	213,11	150,15	144,49	128,33	136,97
8C	757,95	325,15	313,07	146,23	140,05	105,68	105,34	100,04	109,60
16C	531,61	240,48	214,17	112,08	106,68	87, 54	88,07	87,80	103,61
32C	508,89	274,20	196,14	114,82	105,59	93,98	94,19	94,23	106,62

Table 4.2: AND operation on a large version of Case A with duplicating partitioning. Baseline execution time: 22745,37 seconds.

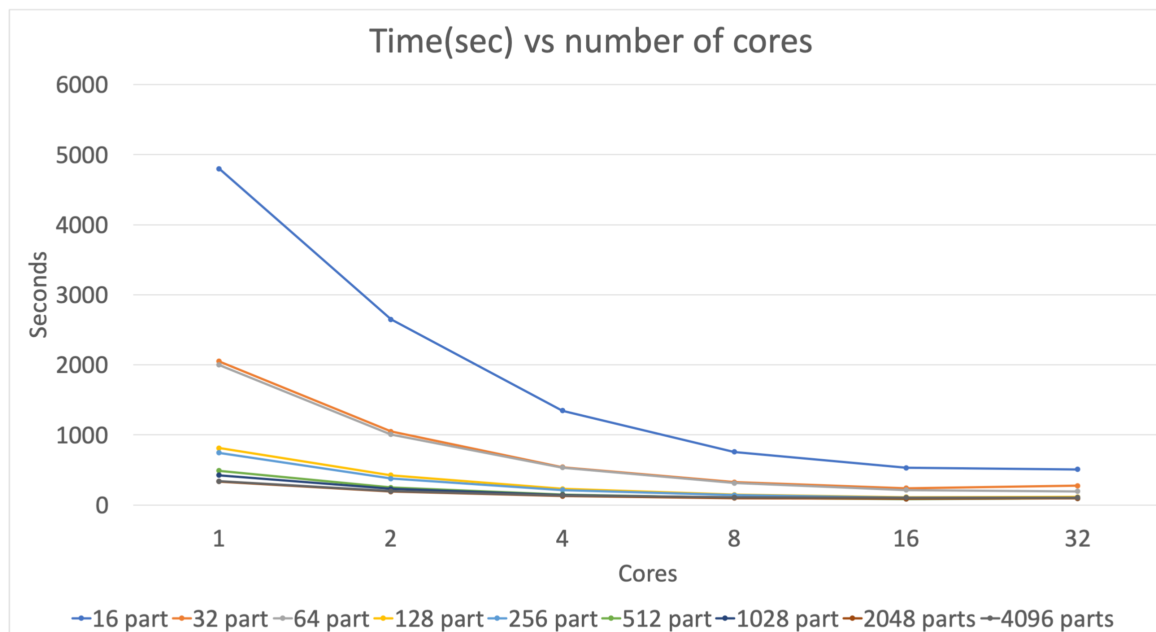


Figure 4.5: Result of AND, case A, large input

Table 4.3 shows total computation times tests done with the splitting partition strategy. Full detailed results can be found in Appendix A4. We can see a similar pattern as in the duplicating strategy. The top result was received with 128 partitions on 8 processor cores, a speed-up of around 19X compared to baseline. Interestingly the combination is different than on the duplicating strategy. Compared to the duplicating strategy, which top-result was 1,38 seconds, this result is slightly higher than the duplicating one. In this top result, we can see in the appendix that this top-result consists of a partitioning time of 0,58 seconds and a merging time of 0,24 seconds which sums to 0,82 seconds. This is 49,3 % of the total running time. This is similar to the 48% which was found on the top result of the duplicating strategy on the same input. Figure 4.6 shows the summary of how merging and partitioning times depends on the number of partitions. The number of partitioning is seen to grow at a slower pace than the time of merging. The time of merging seems to grow at a slower pace before 256 partitions, but more heavily afterwards.

Cores	**Partitions**						
	16 P	32 P	64P	128P	256P	512	1028P
1C	5,29	3,97	3,71	3,22	3,80	4,14	5,24
2C	3,43	2,63	2,36	2,10	2,73	3,15	4,24
4C	2,48	2,00	1,69	1,71	2,28	2,71	3,81
8C	2,83	1,91	1,77	1, 66	2,16	2,59	3,70
16C	3,95	2,71	2,28	1,88	2,34	2,66	4,10
32C	4,03	3,96	3,05	2,55	2,81	2,97	4,09

Table 4.3: AND operation on Case A with slicing partitioning. Baseline execution time: 31,84 seconds

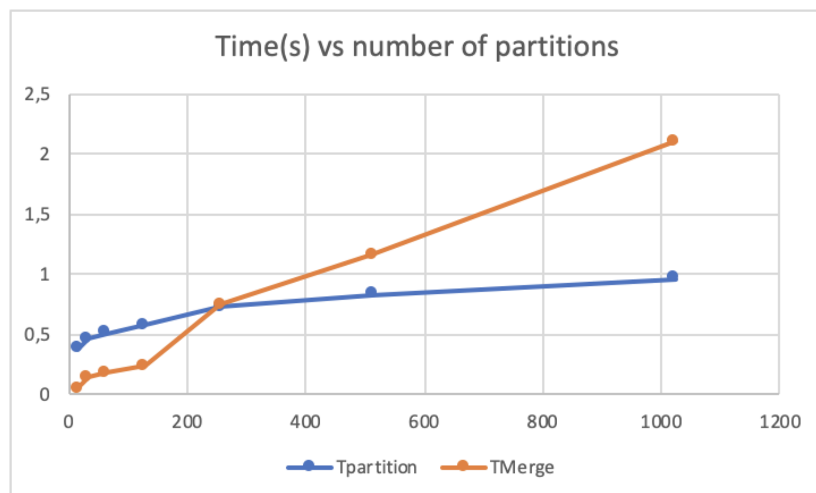


Figure 4.6: AND on case A with splitting strategy. Graph showing how time of merging and partitioning depends on number of partitions

4.2.2 case A - poly, active - OR

The result of the OR operation using the duplicating strategy is shown in table 4.4. The top result of 3,84 seconds was received, compared to the baseline of 75,40 seconds this is a speed-up of 19.3X. In Appendix A.5 we can see that with getting the top result 0,59 seconds were spent partitioning and 1,22 seconds were spent merging. This sums to 47,1 % that was spent on overhead tasks. An interesting point here is that already from the start, time spent merging dominates the time spent partitioning. Compared to the AND-operation on the same input, it is expected that the computational time is larger in the OR operation due to the fact the output produced in the OR case is larger and we know from the theory section that the algorithm is output-dependent.

Cores	**Partitions**					
	16 P	32 P	64 P	128 P	256 P	512 P
2C	8,48	6,93	9,35	11,62	17,63	41,41
4C	5,23	4,597	6,46	8,30	12,72	30,69
8C	4,51	3,84	5,28	6,83	10,74	25,70
16C	5,44	4,83	5,44	6,63	10,03	24,11
32C	5,26	7,00	6,48	7,78	10,62	24,73

Table 4.4: OR operation on Case A with duplicating partitioning. Baseline execution time: 75,40 seconds

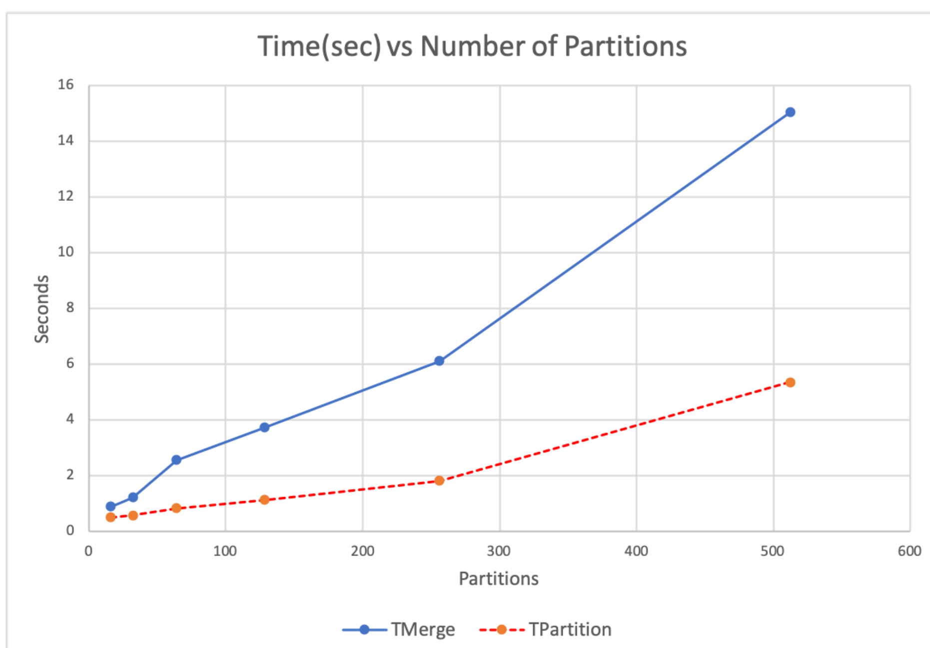


Figure 4.7: OR on case A with duplicating strategy. Graphs showing how time of partitioning and merging depends on number of partitions

4.2.3 Case B, metal1, metal2 - AND

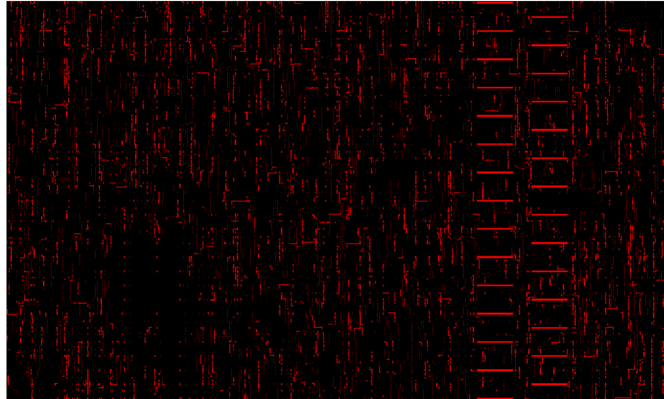


Figure 4.8: Result AND operation.

As seen in the results of the AND operation, the result consists of many, small polygons which are independent of each other. It follows that there likely is not required a lot of merging of polygons that cross multiple regions.

Cores	Partitions					
	16 P	32 P	64 P	128 P	256 P	512 P
1C	25,06	21,10	24,00	26,98	48,30	67,80
2C	13,99	11,75	13,38	15,67	28,60	41,38
4C	8,27	7,21	8,67	9,87	18,98	27,81
8C	6,68	5,40	6,12	7,34	14,13	21,40
16C	7,62	5,93	5,63	6,63	12,33	18,71
32C	7,48	8,033	6,94	7,32	12,03	17,93

Table 4.5: AND operation on Case B with duplicating partitioning. Baseline execution time: 144,04 seconds

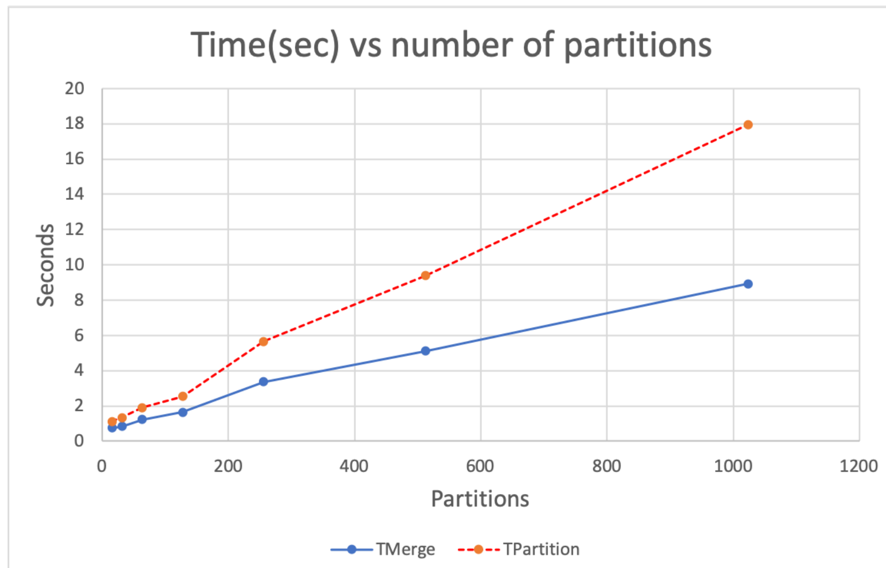


Figure 4.9: AND on case B with duplicating strategy. Graphs showing how time of partitioning and merging depends on number of partitions

4.2.4 Case B, metal1, metal2 - OR

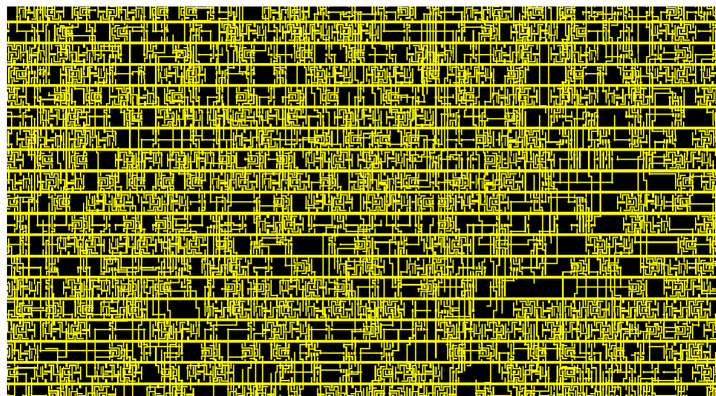


Figure 4.10: Result OR operation.

In figure 4.10 we can see the result of the OR operation. In contrast of the AND operation above, the produces one big polygon with many holes in it. Of course, there are dependencies between the so called independent computations done in the threads. Since these parts must be merged into the same polygon and this merge does contain the sum of all points in the input, large time spent merging is to be expected.

We did run test on this case, but on this case we did not get a successful result because of what was just explained above. Since we have to merge all input data

at the end, we basically redo the whole operation one more time. No speed-up was yielded.

4.2.5 Case B, metal1, metal2 - DIF

In table 4.6 we can see the results of the DIF operation. We note the top result on 64 partitions and 16 processor-cores at 14,46 seconds. Interestingly we see in figure 4.11 that the time of merging is significantly larger than that of partitioning. This is different of what we see in the AND operations. It is however logical since DIF produces larger outputs, which in turn run larger risk of having to be merged.

Cores	**Partitions**					
	16 P	32 P	64 P	128 P	256 P	512 P
1C	112,10	93,36	56,30	48,59	44,90	57,85
2C	57,11	46,69	32,63	30,66	34,013	54,01
4C	33,76	25,62	20,92	22,20	27,56	50,42
8C	24,68	17,01	16,27	18,52	24,87	48,69
16C	22,42	14,85	14,46	17,19	21,42	47,82
32C	22,78	16,04	15,17	17,56	23,31	-

Table 4.6: DIF operation on Case B with splitting strategy partitioning. Baseline execution time:

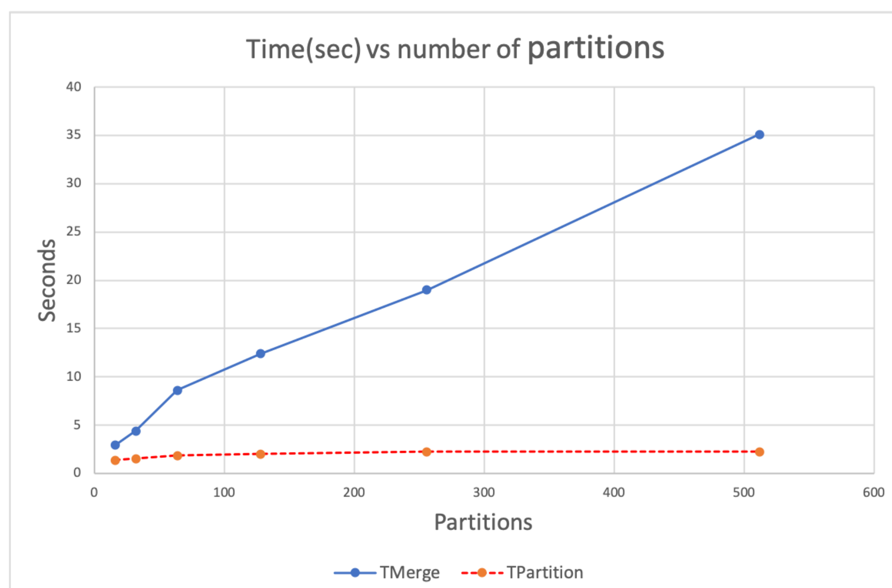


Figure 4.11: DIF on case B with splitting strategy. Graphs showing how time of partitioning and merging depends on number of partitions

4.2.6 Case C - Metal 2, Metal3 - AND

We note the top result of 2,13 seconds running on 32 partitions and 8 cores with a speed-up of 14,7X.

Cores	Partitions					
	16 P	32 P	64 P	128 P	256 P	512 P
1C	7,53	6,74	6,90	9,10	14,13	25,25
2C	4,50	3,88	4,17	5,50	8,88	16,45
4C	2,92	2,58	2,80	3,81	6,38	11,81
8C	2,50	2,13	2,31	2,97	5,23	9,77
16C	3,08	2,29	2,29	2,87	4,90	8,87
32C	3,08	2,95	2,78	3,17	5,03	9,04

Table 4.7: AND operation on Case C with duplicating partitioning. Baseline execution time: 31,41 seconds

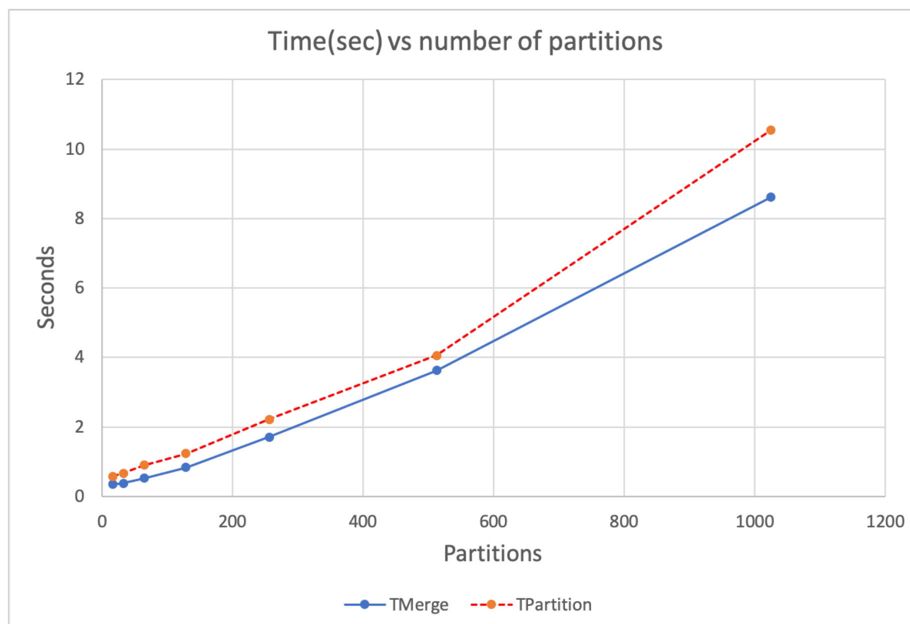


Figure 4.12: AND on case C with duplicating strategy. Graphs showing how time of partitioning and merging depends on number of partitions

4.2.7 Case C - Metal 2, Metal3 - OR

We see that the time of merging is very problematic when the output is one large polygon with many holes in it.

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	1	0,56	142,16	498,51	641,70
32	1	0,64	64,17	479,19	544,69
64	1	0,88	43,19	500,98	546,26
128	1	1,17	30,00	490,78	523,98
256	1	2,15	49,86	512,40	569,12
512	1	3,87	46,74	489,98	548,74
1024	1	10,27	146,70	541,75	716,98

Table 4.8: OR operation on Case B with duplicating partitioning.

4.2.8 Case C - Metal 2, Metal3 - DIF

We note the top result of 3,00 seconds running on 64 partitions and 8 cores. Again, time of merging grows larger and faster than time of partitioning.

Cores	**Partitions**					
	16 P	32 P	64 P	128 P	256 P	512 P
1C	17,53	11,46	8,95	8,31	8,42	11,25
2C	9,09	6,21	5,25	5,40	6,01	9,17
4C	5,28	3,88	3,65	4,00	4,84	8,07
8C	4,12	3,13	3,00	3,38	5,46	7,54
16C	4,92	3,47	3,07	3,44	4,10	7,44
32C	5,23	4,79	3,80	3,84	4,35	7,60

Table 4.9: DIF operation on Case B with splitting partitioning.

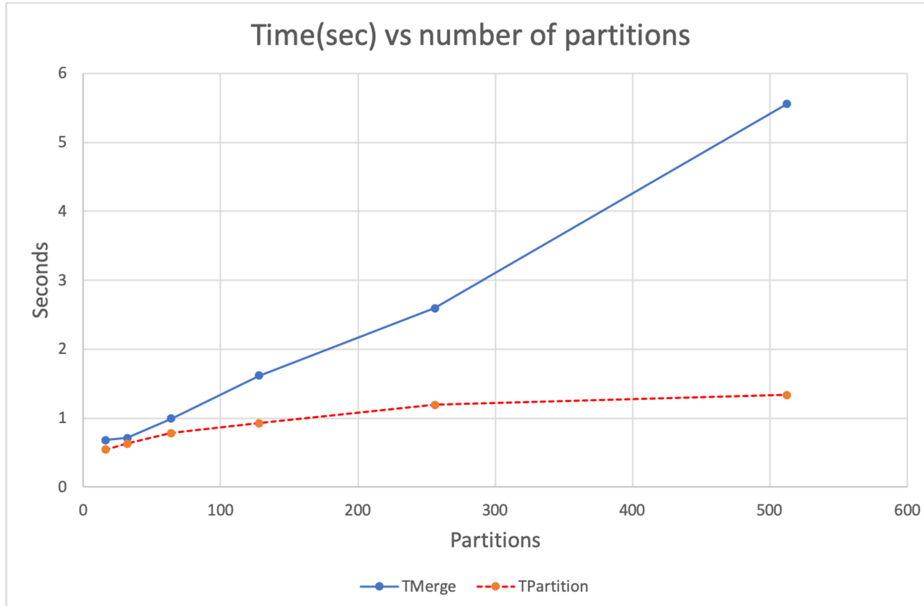


Figure 4.13: DIF on case C with slicing strategy. Graphs showing how time of partitioning and merging depends on number of partitions

4.2.9 Case D - Metal 3, Metal 5 - AND

We note the top result of 0,61 seconds running on 16 partitions and 4 cores. We can see that the small the size of our sample is, the less optimal is it to use many partitions and cores. Top result percentage overhead: 60%

Cores	Partitions				
	16 P	32 P	64 P	128 P	256 P
1C	0,99	1,14	1,69	2,16	2,90
2C	0,72	0,82	1,22	1,51	2,22
4C	0,61	0,66	0,98	1,22	1,75
8C	0,77	0,70	0,91	1,13	1,60
16C	1,09	0,89	0,99	1,20	1,64
32C	1,13	1,16	1,20	1,37	1,70

Table 4.10: AND operation on Case D with duplicating partitioning. Baseline execution time: 1,03 seconds

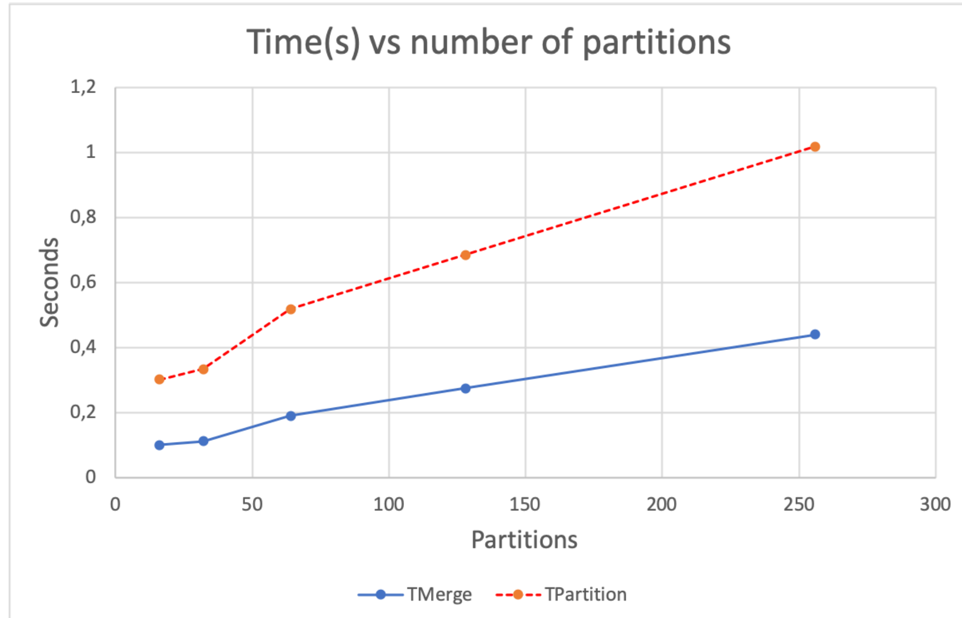


Figure 4.14: AND on case D with duplicating strategy. Graphs showing how time of partitioning and merging depends on number of partitions

4.2.10 Case D - Metal 3, Metal 5 - OR

We note the top result of 1,17 seconds running on 16 partitions and 8 cores. Top result percentage overhead: 47%

Cores	Partitions				
	16 P	32 P	64 P	128 P	256 P
1C	2,31	2,72	4,24	-	-
2C	1,43	1,74	2,82	3,67	5,06
4C	1,19	1,36	2,28	2,87	4,05
8C	1,17	1,27	2,05	2,56	3,66
16C	1,75	1,50	2,17	2,68	3,68
32C	1,62	1,96	2,57	3,00	4,00

Table 4.11: OR operation on Case D with duplicating partitioning. Baseline execution time: 3,06 seconds

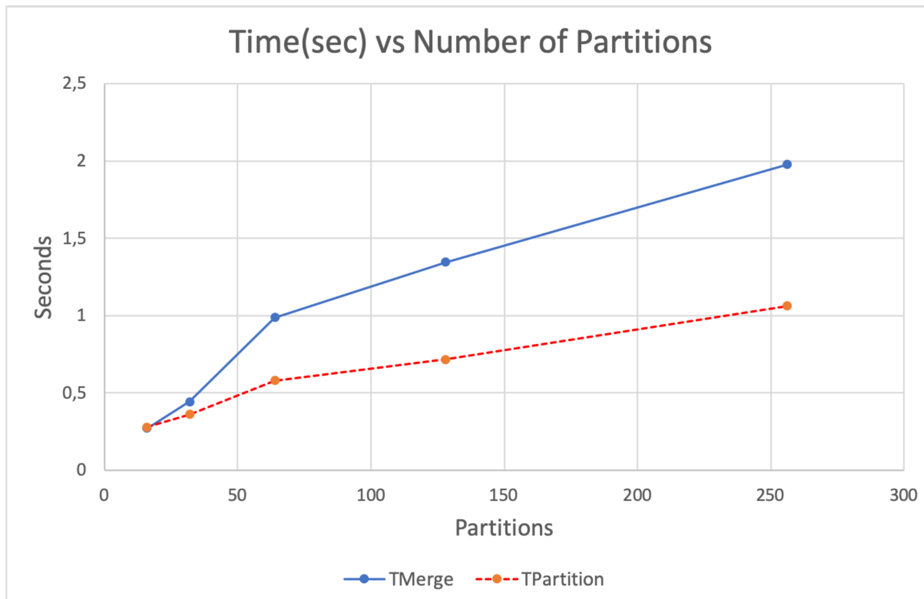


Figure 4.15: OR on case D with duplicating strategy. Graphs showing how time of partitioning and merging depends on number of partitions

4.2.11 Case D - Metal 3, Metal 5 - DIF

We note the top result of 1,08 seconds running on 32 partitions and 8 cores. Top result percentage overhead: 63%

Cores	Partitions				
	16 P	32 P	64 P	128 P	256 P
1C	2.50	2.408	2.571	-	-
2C	1,52	1,57	1,74	2,45	4,31
4C	1,15	1,19	1,38	2,09	3,98
8C	1,25	1,084	1,27	1,94	3,86
16C	1,60	1,28	1,34	1,95	3,78
32C	1,80	1,58	1,74	2,20	3,87

Table 4.12: DIF operation on Case D with splitting partitioning.

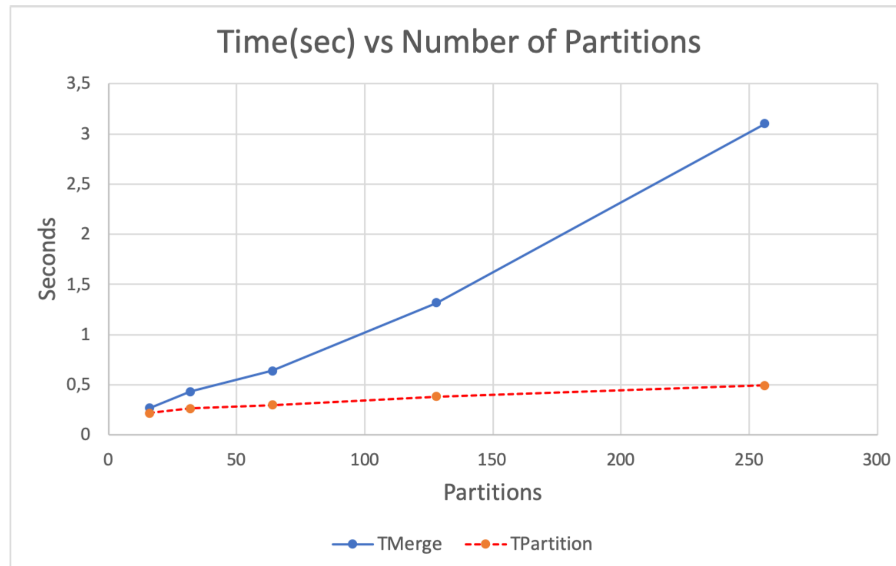


Figure 4.16: DIF on case D with slicing strategy. Graphs showing how time of partitioning and merging depends on number of partitions

Chapter 5
Discussion

In this chapter, we will look back at the results of the evaluation in a greater perspective and relate them to the purpose of this thesis as well as try to answer our problem formulations.

5.1 Overall comments on results

5.1.1 Partition Strategy 1 - Separate Layer

During development of this strategy it became clear that, while the strategy was simple and easy to implement, it was too sensitive to the number of polygons in the separate layer. The result was that large load imbalance happened since the unit of work for the separate layer became much larger than the other units of work. Especially in test cases where the chip had components which covered large amounts of area, the bounding box basically was as large as the chip and no parallelization of significance could take place.

5.1.2 Partition Strategy 2 - Duplication

It is clear from results from the AND operation in all cases that this strategy has the potential of a great speed-up. In the best case, case A with large input, 260X speed-up was achieved. This should not be surprising due to that case A consists of the layers poly and active, which is made up of many, small polygons. The consequence of this is that relatively few polygons are put in different regions, need to be duplicated. It follows that not as many polygons need to be merged. Looking at case B, we can see that we have a more troublesome input where there are many polygons that covers large amounts of the chip. Here we can see that the time of partitioning is generally larger than that of merging, which is likely a

result of that more polygons have to be duplicated. From the result of the AND operation itself we can see that, even though the input covers large amounts of the chip, the output does not. It follows that since we get many small polygons as output, we do not need to spend much time merging. This is likely related to the nature of the AND operation rather than our partitioning strategy. Similar results are seen in the rest of the cases, speed-up is however decreased when the amount of input-data is decreased which of course is logical due to the nature of the algorithm.

The OR operation was also successful when running on case A. In a similar fashion as for the AND operation the input as well output consists of many, small polygons that do not require large amounts of duplication nor merging. A speed-up of around 19X was achieved on normal input. However, when running on other cases, problems were encountered. An output of case B shows that the majority of the output is made up of one large polygon with thousands of holes in it. An operation like this, of course, have many dependencies in it. We were unsuccessful to get a speed-up of notice on these cases. A brief result of case C can be seen to show that the time of merging is what takes large amounts of time.

As explained in the theory section, a big downfall of this strategy is that it is unable to perform the DIF operation(and XOR).

In most cases, there exists a zone of combinations of partitions/cores that perform well. In any practical use, we are not really interested in if an algorithm does have 19.1X speedup or 18.7X speed-up as long we can be sure that it performs above 15X at all times. That we are able to identify zones where we perform well is therefore important.

Regarding the time of the non-parallelizable parts. We have in the result sections taken notes of how big percentage of computation time that is spent on the overhead parts, partitioning and merging on the combinations of partitions and cores that performed the best. These percentages range from 47% to 60%. The input has likely a great impact on this but interestingly we can conclude that around half of the computation time should be spent on doing overhead-tasks. Whenever we are below this mark, we are likely not using enough partitions and if we are above, we should use fewer partitions.

We can conclude that in the tests done that AND has been the most fitting operation for this strategy due to the fact that the output of this operation has fewer dependencies between each other and less time is therefore needed to be spent merging these dependent polygons. In contrast, problems were encountered when performing the OR operation on certain cases that have large dependencies in its output. One can say we are trying to split a problem into different parts that cannot be split up.

5.1.3 Partition Strategy 3 - Splitting

When looking generally at the results of the Splitting strategy, they are quite similar to that of the duplicating strategy. For the AND operation case A the results are slightly slower than the duplicating strategy, but still, a 19X speed-up compared to the 21.8X speed-up with the duplicating strategy.

Looking at the results of the DIF operations we can see that they did perform well. The output of a DIF operation is in general larger than that of a AND operation, and especially so in our test-cases. It is therefore logical that the time of merging is larger than the time of partitioning. The DIF operations do generally perform slower compared to the AND operations as well as a consequence of this. Compared to the OR operation we do however get a successful result. The reason behind this is likely as explained that the result of the OR operation is one or very few polygons, with lots of holes in it which makes the merging consume lots of time. This does not apply to the DIF operation. Given two input layers L_1 and L_2, where the DIF operation is to subtract L_2 from L_1 the output is, of course, the polygons which make up L_1, possible split in some way if they were intersected by L_2. The number of polygons in the output can in other words increase, but the size of each polygon remains the same or decrease. This means that this reduces that chance of us having to merge one mega-polygon together as in the OR operation.

In the test cases used in this report, no problem existed related to the rounding problem described in the method section. In these cases, polygons are mostly orthogonal which describes it. Outside of these test cases, experimenting have been done with non-orthogonal input but with the same result, that there were no problem occurring. This can however be a combination of many thing such as grid size, how polygons mix together, angles of edges etc. It may be so that the theoretical problem exist, but in practical use the problem does not exist. This has however not been proven.

In regards to how the non-parallelizable parts behave, fundamentally this strategy behaves the same as the duplicating strategy. At first, the computation times get lower and lower as we increase the number of partitions. This continues up to a certain point when the pattern turns and either the time of partitioning or the time of merging starts to eat up the total computation time. This strategy does also behave in a similar way when it comes to merging, the merging generally seem to grow in a linear fashion. The partitioning, however, while it starts out with slightly higher computation times on lower amounts of partitions, it seems to grow in a logarithmic way as the number of partitions increase which can be beneficial when running on larger cases where the time of partitioning is the dominating time consumer.

5.1.4 General comments

In all three strategies, we can see that the time of merging is problematic. While the time of partitioning requires its fair amount of time and eventually grows too large to be of use, it grows at a fairly low rate and above all, it is fairly predictable in all three cases since it really only depends on the amount of points and polygons in the input.

On the other hand, the time of merging grows at a larger rate in both the splitting and duplicating strategy. In strategy 1, the time required to merge the polygons back together required many times more time than the baseline operation, and even though it could be effective in certain cases when there are few polygons to merge, this factor made strategy 1 impossible. In the remaining strategies, the time of merging was less predictable than the partitioning time due to the dependency of the polygons in the output. We can say that we are trying to partition something into different pieces and the result is that we have to piece it together again at a greater cost. In cases where we do not have polygons with large output polygons, stretching across many regions, our strategies have been most effective.

We can also see in the tests that the time spent in threads, the time required to compute our separated partitions, does not necessarily increase as the number of threads increase. It is possible that creating too many partitions, consequently creating smaller jobs, combined with more threads resulted in a high rate of context-switching and too small time spent on each job compared to tie spent och context switching and eventually time spent waiting for the mutex lock used to protect the queue of jobs to unlock.

Load balancing generally has not been a problem once we used the binary space partitioning tree which created the regions in such a way that equally many points in every region. On top of that, we used the FIFO thread-queuing method. which made sure there were minimal load balancing issues. When using equally as many partitions as threads, we are more prone to load balancing since we cannot say that the number of points or polygons of a partition has a direct impact on exactly how much time the Boolean operation takes, even though there is a clear relation.

Perhaps the biggest remark on the results generally are that the Boolean operations are heavily output dependent. What the output is decides whether there are any, small or large dependencies between the polygons that we are operating on. This means that we have to treat all polygons as if they have a dependency on any other polygon, this until we can prove that they do not. This, of course, leads us to edge cases which hindered us to make clever solutions. If one can spend more time proving certain situations to get rid of these edge cases, one can perhaps reduce time spent merging.

5.2 The Progress of the Project

It was decided early in the work of this thesis that the Boolean operation would be seen as a black box operation. The plan was quickly mentioned in previous chapters, that to use Silvaco's proprietary Boolean algorithm which is of course more specialized on performing these algorithms on simulated computer chips than a completely general one, such as the one later used. Later on, in this project, it turned out that using Silvaco's algorithm was not a possibility when running computations with multiple cores when running code as a plugin, which we did. We were then forced to look to other possibilities to answer our problem formulation to complete the thesis. The Boolean operation used after this was the one through boost::geometry library. Here we had an issue that the output format was not compatible with the one used by Silvaco's program Guardian, and even if we would be able to make them compatible, a great deal of work would have to be put in. So after spending a significant amount of time testing this library out, we decided to switch to the Clipper Library, which turned out to have a more similar format. Even though some time was spent converting formats from and to clipper format, we were able to make a change in the API used to connect to Guardian to make it as close to seamless as one gets. In summary, much time was spent on testing out libraries that would be compatible and writing scripts for converting formats to each other rather than spending time developing new e.g. merging strategies

The 3 strategies which have been evaluated were proposed from the start and when running the first tests, strategy 2 showed great promise, while strategy 1 did not. Too much computation time was spent on the separate layer, and even though some improvements were made; like having multiple separate layers, there were still clear problems and after this, the tests did no longer include this strategy. Strategy 2 and 3 however performed well and work continued with these strategies and some optimization was done.

5.3 Error Sources

In a project like this, one is always prone to imperfect implementation and one must be aware of the risk this affecting the result. For example; the time of partitioning in the partition strategy grows in a linear fashion. The theory says that it should be able to make this partitioning in logarithmic time complexity.

The time required when running a test-suite has been quite large. In the worst case, case A with large input, if one sums up all the running times the complete test-suite took 49806 seconds or around 13 hours. There is, of course, a risk that the machine is performing better on certain operations than others. It is therefore important not to look at the millisecond difference between cases. One should

rather recognize patterns one is able to observe.

Despite these, we think however we have managed to prove the point of this thesis since one can easily recognize where there are problematic dependencies and that clear speed-ups can be achieved in many cases.

5.4 Further Improvements

In retrospect, some things are clear which could be improved if more time was given. As mentioned in 5.2, we limited ourselves to seeing the Boolean operation as a black box. It is clear that by limiting ourselves this way, we also limit ourselves to the different partitioning strategies we can use. If one instead chooses to implement the boolean operation from the ground up, other more effective partitioning strategies may be exploited.

The partitioning could benefit from a more sophisticated analyzing of the input. Variables taken into account where the number of points and polygons. Perhaps taking shapes into account, a better partitioning with fewer polygons having to be merged could be achieved. This, however, proved difficult since the variation of input received was quite large and it was found difficult to come up with a general input-analysis strategy.

The time of merging grows unbearably large when the output needs large amounts of merging as seen in the DIF operation in case A. How merging is performed may also be improved. A general OR operation of the polygons need merge has been used. A good amount of time was given this task, due to many edge cases we were unable to come up with a merging strategy satisfying all these edge cases.

Chapter 6
Conclusion

We will now conclude this thesis, with three primary points. These are points where, if the reader is to take anything away from this thesis, these three points should be just that.

Results Concluded

The result of the simulations and stress tests done show that there is a great potential in parallelizing these algorithms, with the best results showing a speed-up of around 260X on the AND operation. The OR operation was more troublesome in some cases where large dependencies in the output was created, making parallelizing less attractive. The DIF operation did mostly have good results but not as good as the results seen on AND.

6.0.1 Great Potential

In this thesis, we have shown that there is clearly great potential in parallelizing the boolean operations, with the best result having a speed-up of 260X and one would surely increase that if one were to increase the size of the input data. The potential of the AND operation is also clearly the biggest due to the low risk of having to spend too much time merging an output with many dependencies.

6.0.2 Benefits of Partitioning

Much of the speed-up was gained not only by running the operations on many processor-cores but also due to that we reduced a big non-linear operation to many small ones. In this way, we reduced the computation time greatly without even actually parallelize the operation.

6.0.3 Output Dependencies

The last major conclusion is we are, by the end of the day very dependent on what case we are trying to parallelize and how the output of this case looks like. It was in our experience hard to avoid that sometimes by the end of the operation, we had to produce 1 big polygon. One can describe this as trying to parallelize somethings that do not want to get parallelized.

Final Conclusions

The difference between the duplicating and splitting strategies may be seen as small when looking at the big picture. When compared to the baselines, the difference in results was often minor. They were simply different approaches to solve the same problem and it was really the test-cases together with the type of Boolean operation the dictated the result more than the partitioning strategies themselves. Because of this, the partitioning strategy we would choose to continue with if more time were allowed if we would write production-like code would be the splitting strategy since it is compatible with all operations.

Bibliography

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[2] Ivan J Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the eleventh annual symposium on Computational geometry*, pages 211–219. ACM, 1995.

[3] Jon Louis Bentley and Thomas A Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on computers*, (9):643–647, 1979.

[4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry: algorithms and applications*. Springer-Verlag TELOS, 2008.

[5] T.H. Cormen, T.H. Cormen, C.E. Leiserson, Inc Books24x7, MIT Press, Massachusetts Institute of Technology, R.L. Rivest, McGraw-Hill Publishing Company, and C. Stein. *Introduction To Algorithms*. Introduction to Algorithms. MIT Press, 2001.

[6] J. Skeppstedt and C. Söderberg. *Writing Efficient C Code*. Skeppberg AB, 2016.

[7] Ivan E Sutherland and Gary W Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[8] Bala R Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63, 1992.

[9] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. In *ACM SIGGRAPH computer graphics*, volume 11, pages 214–222. ACM, 1977.

Appendices

Appendix A

Detailed Results

Case A

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	1	0,35	4,75	0,20	5,30
16	2	0,33	2,49	0,18	3,02
16	4	0,32	1,51	0,17	1,99
16	8	0,32	1,64	0,19	2,16
16	16	0,31	2,61	0,19	3,12
16	32	0,32	2,41	0,19	2,93
32	1	0,33	3,27	0,23	3,83
32	2	0,42	1,67	0,22	2,31
32	4	0,33	1,08	0,19	1,60
32	8	0,35	1,02	0,20	1,56
32	16	0,34	1,60	0,20	2,14
32	32	0,35	3,51	0,23	4,09
64	1	0,40	2,91	0,27	3,58
64	2	0,39	1,47	0,25	2,10
64	4	0,39	0,84	0,26	1,49
64	8	0,41	0,71	0,26	1,38
64	16	0,40	0,98	0,34	1,72
64	32	0,38	1,51	0,34	2,23
128	1	0,42	2,73	0,36	3,51
128	2	0,42	1,33	0,32	2,07
128	4	0,43	0,74	0,32	1,50
128	8	0,55	0,61	0,34	1,50
128	16	0,43	0,71	0,32	1,46
128	32	0,43	1,12	0,43	1,98
256	1	0,52	3,03	0,44	3,99
256	2	0,49	1,47	0,44	2,40
256	4	0,48	0,79	0,46	1,72
256	8	0,57	0,62	0,48	1,67
256	16	0,50	0,60	0,47	1,57
256	32	0,47	0,92	0,64	2,03
512	1	0,74	4,03	0,93	5,70
512	2	0,94	1,99	0,92	3,85
512	4	0,91	1,03	0,90	2,84
512	8	0,70	0,69	0,89	2,27
512	16	0,84	0,70	0,92	2,46
512	32	0,75	0,95	1,13	2,84

Table A.1: AND operation on Case A with duplicating partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	1	21,67	4769,40	7,93	4799,40
16	2	21,42	2619,51	8,58	2649,38
16	4	21,74	1317,93	8,22	1347,89
16	8	21,31	728,33	8,31	757,95
16	16	21,18	502,14	8,29	531,61
16	32	21,88	478,61	8,40	508,89
32	1	24,66	2016,78	8,43	2049,82
32	2	24,24	1017,78	8,55	1050,49
32	4	24,80	508,21	8,51	541,52
32	8	24,01	292,74	8,41	325,15
32	16	23,56	208,44	8,48	240,48
32	32	23,48	242,207	8,52	274,20
64	1	26,07	1963,51	8,80	1998,38
64	2	26,68	971,52	8,79	1006,98
64	4	26,09	499,43	8,73	534,25
64	8	26,04	278,28	8,79	313,07
64	16	26,51	178,89	8,77	214,17
64	32	27,25	160,04	8,84	196,14
128	1	27,70	774,68	11,38	813,76
128	2	27,20	386,53	11,28	425,05
128	4	27,97	190,31	11,20	229,40
128	8	27,16	107,69	11,38	146,24
128	16	28,09	72,87	11,12	112,08
128	32	27,15	76,33	11,34	114,82
256	1	27,40	697,89	18,51	743,82
256	2	27,69	330,69	18,73	377,11
256	4	27,95	166,15	19,01	213,12
256	8	28,90	91,92	19,23	140,05
256	16	27,70	59,97	19,01	106,68
256	32	28,02	58,52	19,05	105,60

Table A.2: AND operation on large version of Case A with duplicating partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds. Part 1 displaying partitions 16-256.(see next page for part 2)

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
512	1	29,49	437,36	21,11	487,96
512	2	29,83	198,19	21,87	249,89
512	4	29,25	99,05	21,60	150,15
512	8	29,29	55,37	21,03	105,68
512	16	29,88	36,59	21,05	87,54
512	32	29,62	42,31	22,04	93,98
1024	1	30,69	369,71	25,24	425,65
1024	2	30,29	173,92	24,67	228,88
1024	4	30,64	89,59	24,26	144,49
1024	8	30,40	49,32	25,62	105,34
1024	16	30,06	32,46	25,55	88,07
1024	32	30,75	37,89	25,55	94,19
2048	1	32,16	276,30	27,89	336,35
2048	2	31,73	130,44	28,79	190,96
2048	4	32,48	67,40	28,45	128.33
2048	8	32,97	39,06	27,97	100,05
2048	16	31,78	27,64	28,39	87,81
2048	32	32,23	33,06	28,95	94,24
4096	1	35,84	266,12	38,16	340,13
4096	2	36,03	121,91	38,56	196,51
4096	4	37,09	61,08	38,80	136,98
4096	8	36,19	35,92	37,50	109,61
4096	16	36,47	27,62	39,53	103,62
4096	32	35,69	33,33	37,60	106,62

Table A.3: AND operation on Case A with duplicating partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds. Part 2 displaying partitions 512-4096.

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	1	0,39	4,85	0,06	5,29
16	2	0,38	2,99	0,06	3,43
16	4	0,39	2,05	0,04	2,48
16	8	0,38	2,39	0,06	2,83
16	16	0,38	3,52	0,04	3,95
16	32	0,38	3,59	0,06	4,03
32	1	0,47	3,35	0,15	3,97
32	2	0,47	2,02	0,14	2,64
32	4	0,47	1,42	0,12	2,00
32	8	0,47	1,30	0,15	1,91
32	16	0,47	2,08	0,17	2,72
32	32	0,47	3,32	0,18	3,96
64	1	0,52	3,04	0,16	3,72
64	2	0,52	1,70	0,14	2,36
64	4	0,51	1,04	0,14	1,70
64	8	0,52	1,07	0,19	1,78
64	16	0,52	1,60	0,16	2,28
64	32	0,51	2,33	0,21	3,05
128	1	0,58	2,40	0,24	3,22
128	2	0,58	1,29	0,24	2,10
128	4	0,58	0,90	0,24	1,71
128	8	0,58	0,84	0,24	1,66
128	16	0,58	1,05	0,26	1,89
128	32	0,58	1,61	0,36	2,55
256	1	0,73	2,30	0,77	3,80
256	2	0,73	1,24	0,77	2,74
256	4	0,73	0,80	0,75	2,28
256	8	0,73	0,67	0,76	2,16
256	16	0,75	0,80	0,79	2,35
256	32	0,73	1,07	1,02	2,82
512	1	0,83	2,16	1,15	4,15
512	2	0,82	1,18	1,16	3,16
512	4	0,85	0,72	1,15	2,71
512	8	0,83	0,62	1,16	2,60
512	16	0,83	0,67	1,16	2,66
512	32	0,86	0,84	1,28	2,98
1024	1	0,96	2,18	2,11	5,25
1024	2	0,97	1,18	2,09	4,24
1024	4	0,96	0,78	2,08	3,82
1024	8	0,96	0,64	2,11	3,70
1024	16	0,98	0,87	2,25	4,10
1024	32	0,97	0,79	2,35	4,10

Table A.4: AND operation on Case A with splitting partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds.

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	2	0,47	7,14	0,87	8,48
16	4	0,47	3,87	0,89	5,23
16	8	0,50	3,12	0,88	4,51
16	16	0,49	4,08	0,87	5,44
16	32	0,50	3,90	0,86	5,26
32	2	0,56	5,17	1,20	6,93
32	4	0,57	2,81	1,21	4,60
32	8	0,60	2,02	1,22	3,84
32	16	0,54	3,06	1,23	4,83
32	32	0,58	5,21	1,21	7,00
64	2	0,83	5,97	2,56	9,35
64	4	0,82	3,12	2,53	6,46
64	8	0,83	1,90	2,56	5,28
64	16	0,83	2,02	2,59	5,44
64	32	0,83	3,13	2,53	6,49
128	2	1,13	6,75	3,74	11,62
128	4	1,13	3,43	3,74	8,30
128	8	1,08	2,01	3,74	6,83
128	16	1,11	1,81	3,70	6,63
128	32	1,13	2,96	3,69	7,78
256	2	1,78	9,74	6,10	17,63
256	4	1,78	4,87	6,07	12,73
256	8	1,84	2,75	6,15	10,74
256	16	1,85	2,05	6,13	10,03
256	32	1,79	2,79	6,05	10,62
512	2	5,31	21,04	15,06	41,41
512	4	5,39	10,29	15,01	30,69
512	8	5,32	5,43	14,95	25,70
512	16	5,34	3,71	15,05	24,11
512	32	5,33	4,33	15,07	24,73

Table A.5: OR operation on Case A with duplicating partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds.

Case B

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	2	1,12	12,12	0,76	14,00
16	4	1,10	6,41	0,75	8,27
16	8	1,11	4,84	0,73	6,68
16	16	1,10	5,79	0,74	7,62
16	32	1,16	5,59	0,73	7,48
32	2	1,34	9,55	0,85	11,75
32	4	1,34	5,02	0,85	7,21
32	8	1,34	3,22	0,84	5,40
32	16	1,35	3,76	0,82	5,93
32	32	1,35	5,88	0,80	8,03
64	2	1,93	10,24	1,22	13,38
64	4	2,08	5,33	1,27	8,67
64	8	1,82	3,07	1,23	6,12
64	16	1,83	2,56	1,24	5,63
64	32	1,90	3,82	1,22	6,94
128	2	2,54	11,51	1,62	15,67
128	4	2,48	5,77	1,62	9,87
128	8	2,49	3,21	1,64	7,34
128	16	2,57	2,43	1,64	6,63
128	32	2,55	3,11	1,66	7,32
256	2	5,65	19,61	3,34	28,60
256	4	5,78	9,88	3,32	18,98
256	8	5,58	5,22	3,34	14,13
256	16	5,65	3,22	3,46	12,33
256	32	5,61	3,09	3,33	12,03
512	2	9,47	26,82	5,08	41,38
512	4	9,39	13,34	5,07	27,81
512	8	9,24	7,04	5,12	21,40
512	16	9,38	4,09	5,25	18,71
512	32	9,45	3,39	5,08	17,92
1024	2	17,98	44,06	8,88	70,92
1024	4	17,90	22,08	9,03	49,01
1024	8	17,92	11,50	8,90	38,32
1024	16	18,09	6,31	8,92	33,32
1024	32	17,89	4,44	8,89	31,22

Table A.6: AND operation on Case B with duplicating partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds.

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	2	1,40	52,78	2,94	57,11
16	4	1,40	29,40	2,97	33,77
16	8	1,40	20,35	2,93	24,68
16	16	1,39	18,10	2,94	22,42
16	32	1,40	18,46	2,92	22,78
32	2	1,51	41,25	3,93	46,69
32	4	1,53	19,56	4,53	25,62
32	8	1,53	10,97	4,52	17,02
32	16	1,51	8,80	4,55	14,85
32	32	1,51	10,05	4,49	16,04
64	2	1,85	22,18	8,60	32,63
64	4	1,85	10,47	8,61	20,93
64	8	1,85	5,73	8,70	16,28
64	16	1,87	4,08	8,52	14,46
64	32	1,85	4,72	8,61	15,18
128	2	2,01	16,28	12,37	30,66
128	4	2,02	7,75	12,43	22,20
128	8	2,02	4,09	12,41	18,52
128	16	2,02	2,78	12,39	17,19
128	32	2,00	3,14	12,42	17,56
256	2	2,25	12,18	19,58	34,01
256	4	2,25	5,71	19,61	27,56
256	8	2,26	3,02	19,60	24,88
256	16	2,25	1,93	17,24	21,42
256	32	2,27	2,14	18,90	23,31
512	2	2,79	7,31	43,92	54,02
512	4	2,80	3,65	43,97	50,42
512	8	2,79	1,94	43,97	48,70
512	16	2,79	1,32	43,72	47,8

Table A.7: DIF operation on Case B with splitting partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds.

Case C

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	2	0,62	3,54	0,34	4,50
16	4	0,59	2,00	0,33	2,92
16	8	0,55	1,59	0,35	2,50
16	16	0,55	2,20	0,34	3,08
16	32	0,57	2,17	0,34	3,08
32	2	0,67	2,83	0,38	3,88
32	4	0,68	1,53	0,37	2,58
32	8	0,67	1,09	0,37	2,13
32	16	0,65	1,25	0,39	2,29
32	32	0,64	1,92	0,38	2,95
64	2	0,93	2,73	0,51	4,17
64	4	0,86	1,42	0,52	2,79
64	8	0,89	0,90	0,52	2,31
64	16	0,89	0,87	0,52	2,29
64	32	0,91	1,35	0,52	2,78
128	2	1,23	3,45	0,83	5,50
128	4	1,26	1,72	0,84	3,81
128	8	1,19	0,95	0,83	2,97
128	16	1,22	0,83	0,83	2,87
128	32	1,24	1,10	0,83	3,18
256	2	2,28	4,91	1,69	8,88
256	4	2,20	2,46	1,72	6,38
256	8	2,22	1,31	1,70	5,23
256	16	2,22	0,95	1,73	4,90
256	32	2,19	1,15	1,70	5,03
512	2	4,37	8,48	3,61	16,45
512	4	4,04	4,12	3,66	11,81
512	8	3,99	2,16	3,61	9,77
512	16	3,90	1,35	3,62	8,87
512	32	3,98	1,44	3,62	9,04
1024	2	10,75	17,00	8,62	36,37
1024	4	10,19	8,25	8,61	27,05
1024	8	10,26	4,21	8,61	23,07
1024	16	10,69	2,61	8,60	21,90
1024	32	10,80	2,49	8,60	21,89

Table A.8: AND operation on Case C with duplicating partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds.

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	2	0,55	7,87	0,68	9,09
16	4	0,55	4,05	0,69	5,28
16	8	0,54	2,89	0,69	4,12
16	16	0,55	3,70	0,68	4,92
16	32	0,54	4,01	0,67	5,23
32	2	0,64	4,86	0,72	6,22
32	4	0,62	2,55	0,71	3,88
32	8	0,63	1,79	0,71	3,13
32	16	0,63	2,14	0,70	3,47
32	32	0,63	3,45	0,72	4,79
64	2	0,78	3,51	0,97	5,25
64	4	0,78	1,87	1,01	3,65
64	8	0,78	1,23	0,99	3,00
64	16	0,79	1,29	0,99	3,07
64	32	0,78	2,00	1,02	3,80
128	2	0,93	2,88	1,59	5,40
128	4	0,93	1,46	1,61	4,00
128	8	0,93	0,85	1,60	3,38
128	16	0,93	0,86	1,64	3,44
128	32	0,93	1,26	1,66	3,84
256	2	1,14	2,44	2,44	6,01
256	4	1,13	1,25	2,46	4,84
256	8	1,46	0,71	3,29	5,46
256	16	1,12	0,58	2,40	4,10
256	32	1,13	0,83	2,39	4,35
512	2	1,33	2,28	5,56	9,17
512	4	1,34	1,19	5,54	8,07
512	8	1,34	0,66	5,54	7,55
512	16	1,34	0,53	5,58	7,44
512	32	1,33	0,68	5,58	7,60

Table A.9: DIF operation on Case C with splitting partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds.

Case D

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	2	0,28	0,34	0,10	0,72
16	4	0,28	0,24	0,10	0,61
16	8	0,27	0,40	0,10	0,77
16	16	0,27	0,71	0,10	1,09
16	32	0,30	0,73	0,10	1,13
32	2	0,35	0,37	0,11	0,82
32	4	0,35	0,21	0,10	0,66
32	8	0,35	0,24	0,11	0,70
32	16	0,39	0,39	0,11	0,89
32	32	0,35	0,69	0,11	1,16
64	2	0,57	0,50	0,16	1,22
64	4	0,55	0,26	0,16	0,98
64	8	0,56	0,20	0,16	0,91
64	16	0,56	0,27	0,16	0,99
64	32	0,56	0,47	0,16	1,20
128	2	0,72	0,59	0,20	1,51
128	4	0,71	0,30	0,21	1,22
128	8	0,71	0,21	0,21	1,13
128	16	0,73	0,26	0,20	1,20
128	32	0,73	0,43	0,21	1,37
256	2	1,17	0,78	0,27	2,22
256	4	1,08	0,39	0,27	1,75
256	8	1,08	0,24	0,28	1,60
256	16	1,06	0,28	0,29	1,64
256	32	1,05	0,38	0,27	1,70
512	2	2,88	1,65	0,64	5,16
512	4	2,89	0,83	0,63	4,35
512	8	2,93	0,46	0,63	4,02
512	16	2,89	0,43	0,62	3,93
512	32	2,91	0,57	0,62	4,11
1024	2	8,72	3,83	1,52	14,07
1024	4	8,74	1,93	1,51	12,18
1024	8	8,74	1,06	1,54	11,34
1024	16	8,65	1,03	1,54	11,22
1024	32	8,69	1,12	1,53	11,34

Table A.10: AND operation on Case D with duplicating partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds.

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
0	1				3,10
16	2	0,29	0,89	0,26	1,43
16	4	0,27	0,64	0,28	1,19
16	8	0,28	0,62	0,27	1,17
16	16	0,29	1,19	0,28	1,75
16	32	0,27	1,08	0,27	1,62
32	2	0,34	0,96	0,44	1,74
32	4	0,37	0,54	0,45	1,36
32	8	0,38	0,46	0,43	1,27
32	16	0,37	0,68	0,45	1,50
32	32	0,34	1,17	0,44	1,96
64	2	0,56	1,29	0,97	2,82
64	4	0,62	0,68	0,97	2,28
64	8	0,61	0,47	0,97	2,05
64	16	0,56	0,62	0,99	2,17
64	32	0,56	0,98	1,03	2,57
128	2	0,71	1,59	1,37	3,67
128	4	0,72	0,79	1,36	2,87
128	8	0,71	0,53	1,32	2,56
128	16	0,71	0,62	1,35	2,68
128	32	0,73	0,94	1,32	2,99
256	2	1,06	2,05	1,94	5,06
256	4	1,06	1,04	1,95	4,05
256	8	1,07	0,62	1,97	3,66
256	16	1,05	0,61	2,01	3,68
256	32	1,07	0,92	2,02	4,01
512	2	2,89	4,69	5,11	12,69
512	4	2,89	2,29	5,12	10,31
512	8	2,95	1,30	4,92	9,17
512	16	2,91	1,21	4,91	9,03
512	32	2,89	1,50	5,07	9,46

Table A.11: OR operation on Case D with duplicating partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds.

Partitions	Threads	TPartition	TThreads	TMerge	TTotal
16	2	0,22	1,04	0,26	1,52
16	4	0,22	0,67	0,26	1,15
16	8	0,22	0,76	0,28	1,25
16	16	0,22	1,12	0,27	1,60
16	32	0,22	1,31	0,27	1,80
32	2	0,26	0,87	0,43	1,57
32	4	0,27	0,49	0,43	1,19
32	8	0,26	0,40	0,43	1,08
32	16	0,27	0,57	0,44	1,28
32	32	0,26	0,89	0,43	1,59
64	2	0,30	0,81	0,63	1,74
64	4	0,30	0,43	0,65	1,38
64	8	0,29	0,32	0,65	1,27
64	16	0,30	0,41	0,64	1,34
64	32	0,29	0,81	0,64	1,74
128	2	0,38	0,76	1,31	2,45
128	4	0,38	0,39	1,32	2,09
128	8	0,38	0,24	1,32	1,94
128	16	0,38	0,26	1,31	1,95
128	32	0,39	0,48	1,33	2,20
256	2	0,50	0,72	3,10	4,31
256	4	0,49	0,38	3,10	3,97
256	8	0,49	0,26	3,11	3,86
256	16	0,49	0,19	3,10	3,78
256	32	0,49	0,26	3,11	3,87
512	2	0,53	0,72	3,81	5,06
512	4	0,53	0,36	3,80	4,69
512	8	0,53	0,23	3,81	4,57
512	16	0,54	0,18	3,80	4,52
512	32	0,53	0,22	3,80	4,55

Table A.12: DIF operation on Case D with splitting partitioning including time of partitioning and time of parallel execution in threads. Runtime in seconds.

EXAMENSARBETE Parallelization of Computational Geometry Algorithms
STUDENT Gustav Kullberg
HANDLEDARE Jonas Skeppstedt (LTH), Björn Wehlin (Silvaco Inc.)
EXAMINATOR Krzysztof Kuchcinski (LTH)

Parallelisering av beräkningsgeometriska algoritmer

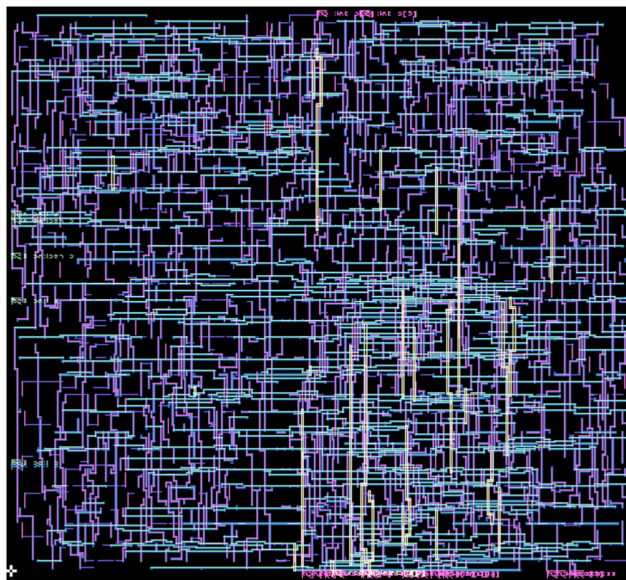
POPULÄRVETENSKAPLIG SAMMANFATTNING **Gustav Kullberg**

Verifikationsprocessen av designade datorchips tar lång tid på grund av boolean operationer. Detta arbete har undersökt möjligheten, eventuella svårigheter samt potentialen i att parallisera dessa operationer.

När företag idag designar datorchips krävs att en verifikationsprocess, så kallad Design Rule Check godkänns innan man fysiskt konstruerar det. Det finns vissa operationer som tar mer tid en andra och fungerar därmed som flaskhalsar. En av dessa operationer är den tvådimensionella Boolean-operationen. Genom att göra beräkningar på ett flertal olika processorkärnor samtidigt, det vill säga att paralllelisera beräkningen kan man ofta reducera beräkningstiden. Problematiken i fallet med den tvådimensionella boolean-operationen är hur man hanterar chip-komponenter som sträcker sig över stora delar av chippet. Då skapas det beroenden mellan olika paralleliserade operationerna vilket måste hanteras.

I mitt examensarbete har designat tre partitionerings-metodiker för att dela upp ett chip olika regioner, vilket möjliggör parallelisering. Dessa tre metodiker drabbas olika hårt av dessa beroenden. Tester har sedan utförts på olika testfall, både realistiska och andra något teoretiska. I dessa tester har olika varitioner av antal kärnor, och olika antal partitioner körts för att skapa en helhetsbild.

Resultatet visar på tre intressanta punkter. Den första är att det finns stor potential i en parallelisering. I bästa fall nåddes en speed-up av 260X



jämfört med samma beräkning på 1 kärna och 1 partition. Den andra slutsatsen är att en stor del av prestanda-vinsten fanns i själva uppdelningen av beräkningen i flertal mindre delar, en stor speed-up nåddes därmed, utan att egentligen köra på flera processor-kärnor. Anledningen till detta är att beräkningstiden av en boolean-operation växer snabbare än vad storleken på input växer. Den sista slutsatsen är att boolean operationerna är beroende av vad resultatet är, d.v.s. output-