POLITECNICO DI TORINO

Master degree
in
Computer Science - Software Engineering

Improving the scalability of modern applications by applying operating system optimizations and parallel paradigms

Supervisor
Stefano Quer
Associate Professor

Candidate
Andrea Calabrese
Matr. 265340

A.A. 2019/2020

Abstract

While technology advances, CPUs are developed with a multi-core approach in mind. General purpose computing on graphic processing units (GPGPU) is becoming a viable approach for massive parallelization approaches, exploiting a higher number of cores and the many-core approach.

The work of thesis focuses two distinct problems: a flying-probes test optimization algorithm, exploiting the capabilities of the GPU computation using CUDA, and an EVCD file analyzer, which exploits the power of multi-threading on the CPU along with operating system level optimizations.

Testing on PCBs (printed circuit boards) can be achieved by using flying probes instrumentation or the bed-of-nails technique; while the latter is very fast, it is dependent on the board layout; instead, flying probes machines can perform tests on different boards, providing that they receive the correct test information; the instrumentation has constraints, in particular probe movement is not completely free and depends on other probes, and it is imperative being able to optimize tests in such a way that their duration is minimized, by moving the probes as least as possible. We divided the problem in three sections: a first prefetch where we calculate possible collision of probes with zones that do not allow movement on them, then the problem of the calculation of the optimal test sequence, end eventually the path finding from one probe positioning to another one.

Results show, for the first problem, that the GPU is not only an available approach, but provides up to 13 times faster computation without in-depth optimizations, only bound by the CPU and the data structures used in order to memorize collisions; by using a more speed-optimized data structure, it is possible to perform 73 times faster computations on the GPU compared with the CPU speed.

Regarding the second problem, a simplified approach has been developed, using, for simplicity, the Microsoft PPL (Parallel Patterns Library), taking advantage of the parallel for in order to compute the distance between as many test points as possible in parallel, using, depending on internal performance metrics, either the CPU or the GPU; using the PPL library provides a significant advantage in terms of computation, between 2 and 4 times using a CPU with 8 threads available. Work is still in development in order to improve the current status, having less simplifications and providing a possible optimization that can be implemented in the future.

Regarding the last problem, a small proof of concept has been developed, yet it is still a work to be discussed; due to its nature to be a potentially expensive calculation, we propose a solution that can be implemented through triangulation of the board, simplifying the problem, yet possibly providing a useful help also for the research of the optimal test sequence.

Simulations are becoming more and more important for safety-related applications and the computation of failure modes. The outputs of the simulations are in the EVCD format (Extended Value Change Dump), representing a list of value changes in the circuit at given times. The nature of the file is sequential, divided into three main parts: first, there is the declaration section; withing this section, we can find declarations of buses and gates of a board; a gate configuration consists in a single state memorized, while a bus configuration consists in the combination of a generic amount of states, where a state consists in a logical 0, a logical 1, a high impedance state (z) or a don't care value (x); a signal is a generalization of a gate, while buses are generalized as agglomerates of signals; the second part of the file is the dump of the initial values of the signals; the third part is a list of changes grouped for each timestamp.

While the first two parts of the EVCD file do not represent an issue for the elaboration time, the third part is the longest one in the file, consisting in a large amount of changes; this means that the number of signals, that can vary between each case, is much lower than the number of changes. In order to optimize the program, we first optimized the file reading in order for it not to be an issue, using binary reading and a double buffering technique in order not to wait for the reading to be complete, already having part of the file ready to perform additional parsing.

Other optimizations are using the least amount of heap possible or, if used, ensuring that the memory is local enough to be cached together. Smaller optimizations to standard functions have also been implemented, saving smaller amounts of time that become significant on the long term. These operating system dependent optimization can speed up the elaboration up to 5 to 6 times, while using the least amount of pointers by using dynamic arrays instead of linked lists saves gigabytes of RAM.

After those optimizations have been developed, we implemented a 1 approach aiming to parse the file and insert the values in the correct structure, in order to perform different types of analysis. While multithreading results are generally good during the parallel part of the analysis (parsing and calculating the coverage of the simulation), the overall process is slowed down by a necessary sequential part, needed to store data in a precise order.

An improved approach is already being developed, where order is automatically guaranteed. Also, different approaches are being experimented in order to improve the analysis time.

Acknowledgement

This work of thesis has been taking a lot of time, thinking and work, but it also gave me quite many satisfactions.

Thanks to my supervisor, Stefano Quer, for providing many useful advices and sharing his experience, also providing always different points of view worth of notice.

Thanks to Paolo Bernardi and Giovanni Squillero, providing the practical applications to work for and helping me understand the problems they provided.

Thanks to my family, who always supported and born me and the discussions for all those years, helping me when I was in the need.

Thanks to my friends, supporting me, exchanging ideas; we shared precious slices of life within each other, memories to keep with us.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increase in number of cores in modern CPUs leads to an overall performance improvement in many applications; yet, most of the algorithmic thinking is done in a sequential fashion, not taking advantage of the ability of the current hardware to implement parallel computation techniques.

Moreover, GPUs are becoming fully programmable, with the main specialization being floating point operations to be performed taking advantage of a massive parallelism.

In this thesis, we chose to work on two problems, showing different ways to implement a parallel computation within the testing domain; in particular, one of the projects makes mainly use of the GPU computation techniques, while the other one focuses more on multithreading and on operating system dependent optimization techniques.

In chapter 2 there is an explanation of the technology and the mathematics that will be used; in particular, we describe some of the advantages of C++ that we exploit in order to have a working program, we talk about parallelism and its different software implementations, choosing the one available on the systems we work on, also providing an overview of the Amdahl's law, very well known in computer science, and its limitations; there also is the mathematics needed to understand the proposed solutions: in particular, a collision detection algorithm, needed for the flying probes test optimizer, and the triangulation process, which can be useful in order to improve the current work.

In chapter 3 we discuss about the GPU parallelism, how it is imple-

mented and why it can provide great advantages in terms of computation performance, thanks to its huge number of cores, even increasing in number in the most recent NVIDIA GPUs; in this chapter we discuss about threads, what slows down the GPU computation, how to perform parallel operations on both GPU and CPU and how its memory is managed inside CUDA, the programming language designed by NVIDIA to perform general-purpose calculations on their own GPUs.

In chapter 4 we describe the problem of the flying probes test optimizer, how test on PCBs (Printed Circuit Boards) are performed today and how the GPU can help in calculating an acceptable test path; tests performed using flying probes machines require a significant amount of path optimization in order to be performed as fast as possible, knowing all the points that the board can be tested on; however, this optimization costs a significant amount of time due to the quantity of data to be processed; we think that the data can be processed in parallel and that geometry concepts such as triangulation can help in solving the whole problem.

In chapter 5 we discuss about a parallel implementation of the elaboration of a EVCD (Extended Value Change Dump) file, that provides information about stress tests performed on a simulated board; EVCD files can vary in size from few megabytes to hundreds of gigabytes, that can be time consuming in terms of elaboration; because of this, operating system dependent optimizations are being used for the current solution in conjunction with multithreading techniques.

The work of this thesis is, however, only the beginning of a path of improvements that are going to be implemented and discussed. Results are already available, yet there is still work to do in order to define a new state of the art.

Chapter 2

Technical background

In this chapter we will discuss about some notions that are used in order to understand the work done later. In particular, chapter 3, the collision detection algorithm described in section 2.4 and the geometry concepts described in section 2.5 are used for chapter 4; in particular, collision detection is used in order to detect if a probe is going to pass through a no-fly zone or a no-touch zone, the geometry section is useful for a possible future work while the CUDA part explains how a GPU works programmatically, what are the possible optimizations and helps understand why the collision detection algorithm scales very well on it (this is also discussed in section 4.3.3).

Amdahl's law and the concept of span are used in chapter 5; in particular, Amdahl's law provides the theoretical maximum speed-up of an algorithm, coupled with the concept of the algorithm span.

The multithreading introduction is common to all the works, providing a general overview of what is the current technology with the parallelism by a programmatic point of view.

The language used mainly in both works is C++; in the flying probes test optimizer, it is also coupled with CUDA, with which it is interoperable.

2.1 C++

C++ is a programming language aiming to be the successor to the C programming language; it expands the C library and toolset by providing a standard library for common data structures, an OOP (Object Oriented Programming) paradigm and becoming more and more operating system independent.

In both works, it is used for many reasons:

- its `std::vector` structure, a very well optimized dynamic array wrapper, allowing the programmer to forget about pointers and automatically resizing when needed;
- the thread library, providing standard thread function definitions for every operating system;
- since it has many standard structures, it is easy to pass from one to the other and compute the advantages of one over the other in practical cases;
- standard structures help in avoiding memory overflows by having an exception system and boundary checking.

2.2 Concurrency and parallelism

Definition 1 CONCURRENCY *is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.*

Hardware level concurrency is mainly achieved by having multiple cores on the same CPU, leading to parallelism, which is achieved when instructions to achieve the final result run at the same time.

Software level concurrency is achieved by operating systems by using three main paradigms:

- **process-based parallelism**: parallelism is achieved by dividing the process in two, effectively cloning the current running program. This

approach is expensive in terms of time and resources, since the whole process is copied in RAM once again and memory is not shared between processes;

- **thread-based parallelism**: parallelism is achieved by mapping the hardware threads (usually corresponding to the number of cores on the CPU) to a larger number of software threads. This approach provides lower, yet still significant, overhead. In this case, memory is shared between threads and the program is not loaded again into memory; those features make this approach compatible with GPUs, which also work using the thread model; This is the paradigm that will be used during the work of this thesis.

- **task-based parallelism**: it is a relatively new approach; tasks are thread-independent abstractions, assigned to threads at runtime. The low overhead of tasks makes them theoretically preferable to threads in some cases, but they lack many synchronization primitives such as semaphores.

2.2.1 Microsoft PPL

In chapter 4 Microsoft PPL (Parallel Patterns Library) is used in order to reduce the work to achieve a *parallel for* loop. PPL provides an imperative programming model that promotes scalability and ease-of-use for developing concurrent applications [10].

2.3 Amdahl's law

Amdahl's law provides a theoretical limit to the speed-up of the program given the number of cores and the ratio of the program that runs in parallel. It is formulated in the following way:

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

where:

- **S** is the speed-up that the computation will receive;
- **N** is the speed-up of the part of the task that benefits from the parallelism; in this case, it roughly represents the number of threads;
- **p** is the proportion of the execution time that benefits from the parallelism.

A theoretical superior limit is achievable by letting N tend to infinity. In this case, the formula can be written as follows:

$$S = \frac{1}{1-p} \tag{2.2}$$

While Amdahl's law is a theoretical superior limits, it does not take into account technical details about the overhead of threads and synchronization.

Since Amdahl's law is a general concept, due to its theoretical nature it does not take into account many relevant factors about the GPU; this is why it is not used to calculate the optimal speed-up on GPU algorithm, even though it is a general concept.

2.3.1 Span

The concept of SPAN completes to Amdahl's law, defined as follows. The span is the time it takes to perform the longest chain of tasks that must be performed sequentially [9]. We can identify more *spans* in a program, taking into account only the ones being interesting to our purpose.

2.4 Collision detection

Collision detection is a geometrical problem that aims to uncover if two or more geometrical objects intersect each other. Many algorithms have been developed in order to solve this problem, some of them being approximated in order to save computational power, since a precise collision detection can take a significant amount of time if many geometrical objects are involved.

In chapter 4, collision detection is used to find incompatible points (see section 4.1.4). Since paths between points are modeled as rectangles and

No-fly zones are generic polygons, a simple solution is applying a segment-segment collision detection and reproduce it four times the number of segments composing the polygon, each time on one side of the rectangle. While optimizations can be done, those are not considered in this section.

2.4.1 Segment-segment collision detection algorithm

Data structures

The data structure needed is the line. Each line contains two points: **start** and **finish**.

Each point is identified by two coordinates: x and y. This allows the algorithm to retrieve informations on the parametric line passing through the two points.

Overview

A generic line is defined by the following mathematical formula:

$$Ax + By + C = 0 \tag{2.3}$$

This is called the IMPLICIT FORM of the line. It is preferrable to the explicit form since it allows the representation of vertical lines without recurring to limits theory.

In the first part of the algorithm parameters A, B and C are retrieved from the line segment endpoints coordinates. Then, the denominator represents both a useful value and a check to detect if the line segments are parallel. If they are parallel, they do not touch (or they touch infinitely, but this case is ignored).

If the line segments are not parallel, we find the x and y coordinates of the collision. Since we are interested in line segments instead of lines, even if the segments are not parallel they may not collide. The part with the checks ensures this is respected by calculating the ratio of the distance between the collision point over the one using the extremes. If this ratio is not between 1 and 0, this means that the segments do not collide; otherwise, they do collide.

In the final results, we are not interested in the collision point, so a simple boolean is returned as result of the check.

The algorithm

Algorithm 1 SEGMENT TO SEGMENT COLLISION DETECTION

```
1:  function lineToLineCollisionDetection(firstLine, secondLine)
2:      A_1 ← firstLine.finish.y − firstLine.start.y
3:      B_1 ← firstLine.start.x − firstLine.finish.x
4:      C_1 ← A_1 ∗ firstLine.start.x + B_1 firstLine.start.y
5:      A_2 ← secondLine.finish.y − secondLine.start.y
6:      B_2 ← secondLine.start.x − secondLine.finish.x
7:      C_2 ← A_2 ∗ secondLine.start.x + B_2 secondLine.start.y
8:      denominator ← A_1 ∗ B_2 − A_2 ∗ B_1
9:      if abs(denominator) == 0 then return false
10:     x ← (B_2 ∗ C_1 − B_1 ∗ C_2)/denominator
11:     y ← (A_1 ∗ C_2 − A_2 ∗ C_1)/denominator
12:     check_0 ← firstLine.finish.x! = firstLine.start.x
13:     check_1 ← firstLine.finish.y! = firstLine.start.y
14:     check_2 ← secondLine.finish.x! = secondLine.start.x
15:     check_3 ← secondLine.finish.y! = secondLine.start.y
16:     if check_0 then
17:         r ← (x − firstLine.start.x)/ − B_1
18:         check_0 ← r >= 0 and r <= 1
19:     if check_1 then
20:         r ← (y − firstLine.start.y)/A_1
21:         check_1 ← r >= 0 and r <= 1
22:     if check_2 then
23:         r ← (x − secondLine.start.x)/ − B_2
24:         check_0 ← r >= 0 and r <= 1
25:     if check_3 then
26:         r ← (x − secondLine.start.y)/A_2
27:         check_0 ← r >= 0 and r <= 1
        return (check_0 or check_1) and (check_2 or check_3)
```

2.5 Triangulation

As for a possible future work, an approach o the problem described in chapter 4 requires a triangulation process of a polygon. Triangulation is the operation of dividing a polygon into triangular sections.

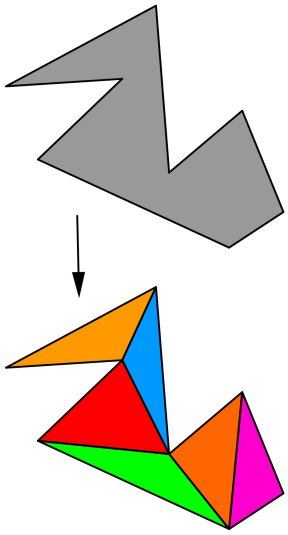

Figure 2.1: The triangulation process

Many algorithms have been developed for triangulation, some using optimizations for special cases, such as convex polygons, polygons without holes or monotone polygons.

Monotone polygons are polygons that, given a line L belonging to it, any line orthogonal to L intersects the polygon at most twice. Our polygon is not convex neither monotone and can have many holes, so it does not fall into such particular cases. Yet, non-monotone polygons can be split into many monotone ones, providing a possible divide and conquer approach between the left side and the right side.

Using a sweep line algorithm in 2 dimensions, it is possible to divide the polygon into monotone parts; the sweep line approach consists in having a sort of scan line going from one direction to the other in 2D space (from left to right or the opposite). It is possible to split the polygon in two halves using a vertical line and compute the triangulation on each half polygon. This may result in a small overhead of triangles, but would improve the splitting algorithm.

Chapter 3

NVIDIA CUDA

NVIDIA CUDA (Compute Unified Device Architecture) is a technology providing a set of APIs (Application Programming Interfaces) for general purpose programming on NVIDIA GPUs (Graphics Processing Units). It was first released in 2007 and it is frequently updated; it is designed to work along with Fortran, Python, MATLAB, C and C++ programming languages [12]. In this thesis, we will use CUDA version 10.2 along with C++.

3.1 GPGPU history

GPUs are becoming more and more powerful in terms of performance every year; In the late 2000s, the industry was moving from a GPU architecture with limited programming capabilities to a completely programmable one; the paradigm for programming on GPU is called GPGPU (General Purpose computing on Graphic Processing Units). Libraries such as DirectX and OpenGL require a good amount of experience in order to be used, thus OpenCL was born in order to solve those issues. OpenCL was one of the first GPGPU languages. Since it is usable on all types of GPU supporting them, they are not optimized for NVIDIA GPUs. NVIDIA created their own language, CUDA, providing an easy way to enable GPGPU in a simple way on their GPUs.

3.2 About GPUs

GPUs have been used for graphics processing for a long time. Graphics processing requires single operations to be performed on a large set of data repeatedly; being the single operations all the same in every stage, it made sense to think to GPUs as a SIMD (Single Instruction on Multiple Data) accelerator for general purpose applications. The SIMD paradigm allows a single instruction to be performed on a large set of data. GPUs have been optimized during the years for managing large sets of floating points data, thus their computation power is measured in FLOPS (FLoating point Operations Per Second), in later years increasing the capabilities to the order of the TeraFLOPS (10^{12} FLOPS) even in home computing GPUs.

3.3 Threads, blocks, grids and warps

Figure 3.1 shows the architecture of the Pascal micro-architecture, available in the NVIDIA GTX 10xx series. In the figure, each green square is a CUDA core of the SM (Streaming Multi-processor). CUDA, through its high-level APIs, allows the programmer to abstract the number of CUDA cores from the hardware, implementing SOFTWARE THREADS for each core, similarly to how a CPU is used by operating systems. While the number of software threads is limited, it is large enough not to be an issue.

The CUDA APIs group software threads in BLOCKS, which are grouped in a GRID. The maximum number of threads in a block and blocks in a grid is also implementation dependent, yet again it is large enough not to be an issue with the vast majority of the applications.

Due to the micro-architecture, threads are grouped in a small hardware matrix containing 32 of them. This matrix is called a WARP. Each call to a GPU function spawns automatically a number of threads multiple of the warp size. If a single thread is issued by the programmer, a warp is allocated for the task, leaving the remaining 31 threads occupied but not working. Each thread warp, moreover, reaches its optimal computational speed when no decisional branching code paths are present (no *if*s) or when the branching path is uniformly chosen, leaving the other branch unused for the warp; if the decisional path between the warp diverges, branching

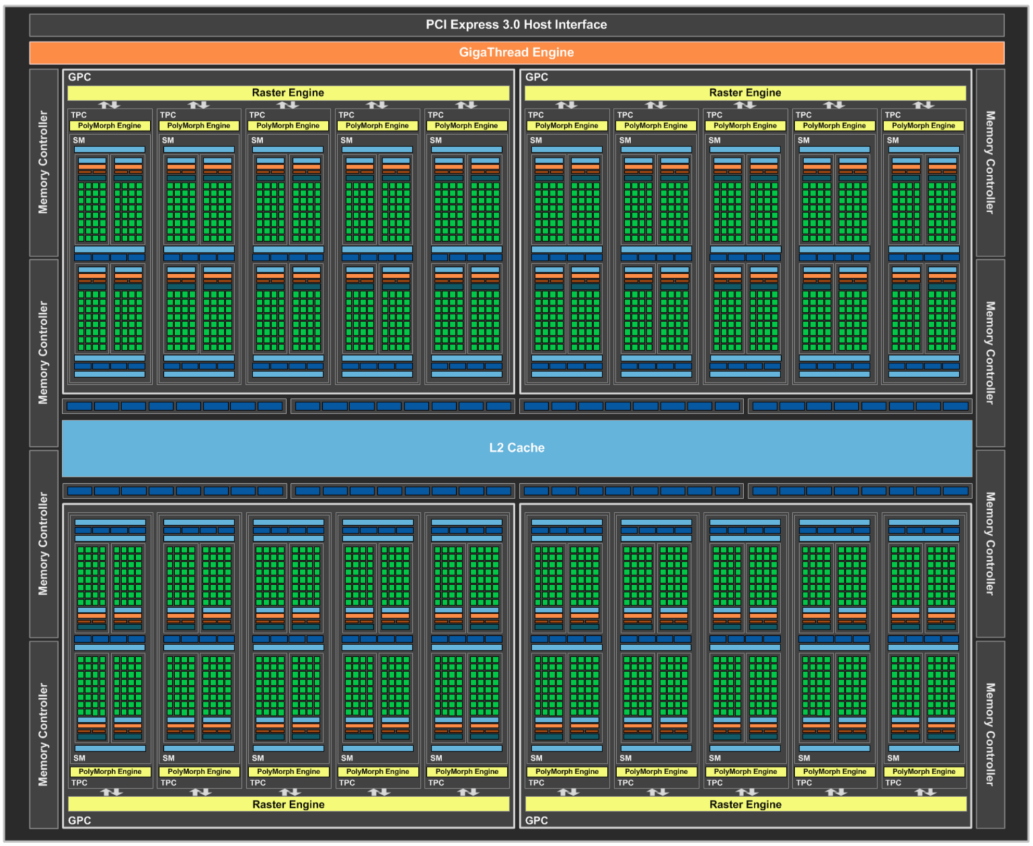

Figure 3.1: NVIDIA PASCAL micro-architecture

instructions are executed sequentially in order; if a thread does not need to execute the instruction, the code for that thread is filled with `nop` assembly instructions[11].

Blocks limit the working threads in a warp: the active threads in more blocks are not optimally distributed in warps. This means that if the number of threads in a block is not multiple of the warp size, then the number of threads used is the warp size multiplied by the number of blocks, while the active threads are only the ones declared in the block size.

3.4 Kernels

GPU functions that can be launched through API calls from the CPU are called KERNELS. A kernel can be executed in parallel on an arbitrary number

of threads, providing an easy way to perform parallel computations. Kernels can accept an arbitrary number of parameters, such as functions in C, and need two mandatory parameters: the number of blocks used and the number of threads for each block. Kernels are declared similarly to functions, using the `__global__` declaration specifier. A kernel can be called by using the `<<<...>>>` execution syntax, similar to the following example:

```
1    // Declaration
2    __global__ myKernel(param1, param2, ...);
3    // Usage
4    myKernel<<<n_blocks, n_threads>>>(param1, param2, ...);
```

When calling *myKernel*, *n_blocks* is the number of thread blocks allocated on the device and *n_threads* is the number of threads for each block. As a rule of thumb, it is advisable to dimension the blocks in such a way that their dimension is a multiple of the block size, as mentioned above.

3.5 Memory

GPUs have their own RAM, called V-RAM (Video-RAM), separated from the main memory of the CPU. V-RAM is physically closer to the GPU cores than RAM is to the CPU, reducing latency, and is usually faster than the current RAM generation for CPUs.

Memory in CUDA is divided into sections: GLOBAL memory, LOCAL memory, CONSTANT memory, SHARED memory and TEXTURE and SURFACE memory. Since the application described in chapter 4 is not a graphical one, texture and surface memory are not used.

3.5.1 Global memory

Global memory is the slowest one on the GPU, since it can contain all kind of data. In order to improve computational speed, data must be well aligned to 32, 64 or 128 byte segments. In global memory resides everything memorized through `cudaMalloc()` calls, which are the equivalent for `malloc()` in C. CPU can not operate on this memory directly, except for the operations of allocating it and copying the values from and to it, using CUDA functions.

Since it is not the fastest memory available, it is advisable to use different types of memory whenever possible.

3.5.2 Local memory

Local memory occurs when data is explicitly declared as shared, device or constant. It is mainly used when the compiler does not know if an array is indexed with constant quantities or when data structures would not fit into the register space.

3.5.3 Constant memory

Constant memory resides in the device memory and is usually cached in the constant cache, in order to speed up their access. Constant memory is useful when non mutable data needs to be used frequently, providing an increased throughput.

3.5.4 Shared memory

Shared memory is an on-chip memory, closer to the actual cores. It is shared within a block, with the lifetime of the block itself. It is useful when we have partial data to collect or frequent operations on those data. Shared memory is also synchronous, meaning that threads are automatically stalled if one is writing on a particular memory address. A simple trick in order to have independent data is allocating a shared array on which each thread can work.

3.6 Asynchronous operation execution

CUDA also provides a set of API used for asynchronous operation execution. This means that it is possible to perform an asynchronous memory copy coupled with an asynchronous kernel execution, allowing the parallel usage of both GPU and CPU for computation. This is headache via the CUDA STREAMS. Using a stream and passing it through operations that need to be performed in order (such as memory copying and the execution of a kernel accessing that memory) allows the programmer to order operations naturally as they are be called in code.

Chapter 4

Flying probes test optimization

4.1 The problem

Manufacturing PCBs (Printed Circuit Boards) is a complex process requiring extensive testing in order to be acceptable. Test phases are often implemented starting at intermediate steps during the manufacturing process [3].

Final tests focus on finding the defect introduced during the board assembly process, and many approaches are used in order to define the defects introduced by this process by providing scores for the defect coverage[6]. Board testing using flying probes is one of the most used methodology in order to verify the correct operation of PCBs. While it is not the only methodology, its advantages reside in the possibility of re-using the same hardware to test different boards. The main disadvantage, compared to the other most used method, the bed of nails, is that testing is much slower, due to the lower number of points tested simultaneously.

Some work has already been done in the test optimization techniques[5][1]. This work aims to extend this work in a many-core environment, exploiting the advantages of GPGPU.

4.1.1 The testing machine

We modeled the testing machine thinking of one having 4 arms on each side, on which there is a probe. The arms move, in a first approximation, in a 2D

space. Each arm can move on the x axis. On each arm is attached a probe, with the possibility of moving on the y axis with a different velocity than the x axis, being generally faster.

When probes are moved, there are 2 possible height values, being *low* or *high*; the height cannot be specified for one probe only, but is common to all of them. Raising a probe to the *high* value takes a considerable amount of time, and might be used to reach points on the board avoiding lower components.

Each side of the machine is theoretically independent, but tests may not be: for this reason, tests must be optimized on both sides of the board.

While the number of probes has an impact on the test timing, it is not the only factor that plays a significant role today: minimizing the number of tests has a large impact on the time spent testing[14], but also optimization .

4.1.2 The tests

We modeled the test in such a way that is possible to have equivalent tests in different points. First, we defined what is an *atomic point*:

Definition 2 *An* Atomic Point *P is a point in a 2D space. It is also simply referred as a point.*

Then, we defined a *test point*:

Definition 3 Equivalent Points *are a set of equivalent atomic points such that there is no difference on having a probe located in one atomic point or another one in the same set in order to perform a test.*

Definition 4 *A* Test Point *can be an atomic point or a set of equivalent points.*

A complete test requires the positioning of two or more probes in different test points simultaneously. Tests are grouped in *test blocks*, with the following characteristics:

Definition 5 *A* Test Block or Test Group *is an ordered set of tests points to be reached simultaneously. The number of test points in each test block can not exceed the number of probes.*

4.1.3 The probes

Each probe can move in a 3D space following the rules here described:

- Movement on the z axis (vertical axis) is always applied to all probes;
- Movement on x and y axis are independent;
- Probes can not change their order in the x axis; this means, for instance, that the leftmost probe will always be the leftmost probe;
- Probes do not move in a straight line; the worst case scenario of movement is a rectangle, which models their ability to move from a point to another.

Since probes have an order, we can label them from left to right as p_0, p_1... With N probes, we can define a probe configuration as a set of points having (x, y) coordinates, where each point represents the position of the probes in order according to their labeling. Using those rules, a configuration is defined as follows:

Definition 6 *A* Probe configuration *is a set of coordinates defining the positioning of all probes.*

A valid configuration is a configuration that follows the rules listed above.

4.1.4 No-touch zones and No-fly zones

On each board, it is likely to find zones where the probes would touch a component they should not. If the zone presents an high component, the probe can not pass over it, leaving a circumnavigation as only solution to avoid a collision with it.

Definition 7 *A* No-touch zone *is a zone where probes need to be raised higher than usual in order to avoid it.*

Definition 8 *A* No-fly zone *is a zone where probes can not fly over.*

Because of the No-fly zones, we may find with incompatible points.

Definition 9 *A couple of points is* INCOMPATIBLE *if the path leading from one to the other intersects a No-fly zone.*

While it is possible defining also the equivalent of incompatible points for No-touch zones, these zones are an easier problem to manage, since the worst case scenario requires raising all the probes.

An obvious consideration is that no-fly zones and no-touch zones cannot contain test points.

4.2 Board generation

In order to be able to get some significant results, it is necessary to have or to generate a board taking into consideration all of what has been defined above.

4.2.1 First board generator

Approach

The first solution was generating the board starting from a generic file, randomly choosing atomic points, equivalent points and test blocks, while modeling no-fly zones and no-touch zones as rectangles.

The board has some user-defined areas, within which there is a uniform distribution of atomic points; each area can be more or less populated by setting a weight parameter.

Among the various signals, ground was considered the most significant in order to have a good number of equivalent points; for this reasons, parameters such as the ground frequency, indicating the number of equivalent ground points, and the special test point named GND were designed to manage this special case.

Equivalent points are generated by joining random points. A parameter defines the probability of having Equivalent points with 2, 3 or 4 equivalent atomic points. In order to generate random test blocks, test points are shuffled within the array and then grouped in blocks. Blocks are generated considering equivalent points from first to last, while randomly choosing the ground for some of them.

Advantages and disadvantages

An advantage of this approach is the ability to generate a board with few parameters. This allows a fast generation of different types of boards.

However, it presents the following disadvantages:

- there is no direct control over the generation of equivalent points, leading to a high probability to have meaningless ones; ground is the most controllable point, and it can have meaningless points too;
- an issue is the modeling of no-fly zones and no-touch zones as rectangles; in the second version of the generator, this has been taken care of;
- atomic points are distributed randomly but their distribution may have no meaning in a real case.

4.2.2 Second board generator

In order to overcome the disadvantages, a second, more precise board generator was later developed. This aims to give more control to the user generating the board, by effectively providing a layout description of the board. Some of the old parameters were deprecated and some were changed, in order to have a board with meaningful test points.

In order to ease the process of generating the board, a board viewer has been developed using Python, depicting the most important components.

Approach

In order to give more meaning to the generated point, the abstraction of a component has been created; a component can represent anything from a full system on chip to a simple resistor. Components are modeled as rectangles, with the number of pins defined for each side; each pin represents an atomic point and it is given a unique ID in order to retrieve it.

the generation also allows for user-defined point to be created with a custom identifier; this way, points are stored within a `std::map` and can be retrieved by their IDs.

To generate equivalent points, nets can be defined by users, providing strands joining each atomic point. This way, the ground signal does not become a special case, but falls in the general case, being particular only because of the larger amount of connections. A net optimizer has been developed when generating the board, joining nets if they have points in common before generating the board. This provides the effective number of meaningful equivalent points.

Test blocks are generated randomly; they test each atomic point in a net, leaving the user the rest of the test groups definition.

No-fly zones and no-touch zones are now modeled as polygons; to define the polygon, the program uses a list of (x, y) coordinates. In order to speed-up the generation of the board, now the generator does not check if a point is inside a no-fly zone or no-touch zone. This can be done by inspection using the board viewer.

Given the input, the output is translated in a similar file, lacking the information of the components, presenting the information of all generated points and with joined nets. IDs for all test points and nets are represented by positive integer numbers instead of strings. The generator assigns each point an integer ID and all the string references contained in test groups and nets are translated into integer references.

Advantages and disadvantages

Advantages of this approach are:

- users have the control of where to put test points, aided by the definition of components
- components give meaning to atomic points
- nets help with the generation of equivalent points; ground is not a particular case

The main disadvantage is having to work longer in order to generate a board, by placing components and verifying by inspection to not violate constraints for the test points. A program with a GUI can be used in order to simplify the generation of the file, but it is out of the scope of this thesis.

Board viewer example

On the top, we have the code that generates the board. On the bottom, there is the view of the board itself.

```
# Board name
.name fake_board.txt
.width 1200
.height 800
# .component, name, x_left, y_bottom, x_right, y_top, pinTop,
                    pinBottom, pinLeft, pinRight
.component CPU 500 500 600 600 20 20 20 20
.component el0 300 100 400 150 8 8 2 2
.component el1 300 200 400 250 8 8 2 2
.component el2 300 300 400 350 8 8 2 2
.component el3 300 400 400 450 8 8 2 2
.component el4 300 500 400 550 8 8 2 2
.component el5 300 600 400 650 8 8 2 2
.component debug 700 550 750 600 8 8 8 8
.component R0 440 300 470 310 0 0 1 1
.component I0 440 400 500 420 0 0 1 1
.component stick0 800 340 840 380 1 1 1 0
.component stick1 800 400 840 440 1 1 1 0
# .point, name, x, y
.point GND_point 600 750
.point CPU_to_gnd0 600 700
.point el5_to_GND 300 750
.point stick_to_GND 820 700
#.net, name, point1, point2...
.net CPU_GND_net GND_point CPU_to_gnd0 CPU.bottom.0
.net el_GND_net GND_point el5_to_GND el5.bottom.7 el5.top.7
                    el4.bottom.7 el4.top.7 el3.bottom.7
                    el3.top.7 el2.bottom.7 el2.top.7
                    el1.bottom.7 el1.top.7 el0.bottom.7
.net debug_GND_net debug.bottom.0 CPU_to_gnd0
#debugger-CPU nets
.net CPU_dbg0 CPU.right.0 debug.left.0
.net CPU_dbg1 CPU.right.2 debug.left.1
.net CPU_dbg2 CPU.right.19 debug.left.7
.net CPU_dbg3 CPU.right.18 debug.left.6
.net CPU_dbg4 CPU.bottom.19 debug.bottom.7
.net CPU_dbg5 CPU.bottom.18 debug.bottom.6
.net el2_R0 el2.right.0 R0.left.0
```

```
.net R0_CPU R0.right.0 CPU.top.0
.net GND_lnk1 CPU_to_gnd0 stick_to_GND
.net stick0_GND stick0.bottom.0 stick_to_GND
.net stick1_GND stick1.bottom.0 stick0.bottom.0
.net CPU_to_stick0 stick0.left.0 CPU.top.16
.net CPU_to_stick1 stick1.left.0 CPU.top.18
.net CPU_to_stick0_fb stick0.top.0 CPU.top.17
.net CPU_to_stick1_fb stick1.top.0 CPU.top.19
.net I0_to_CPU I0.right.0 CPU.top.1
.net I0_to_el3 el3.right.0 I0.left.0
#each couple of values is an extreme of a segment, each line is
                    a no fly zone
.nofly nfzI0 450 390 490 390 490 430 450 430 450 390
.nofly nfzStick0 805 360 820 375 835 360 820 345 805 360
.nofly nfzStick1 805 420 820 435 835 420 820 405 805 420
```

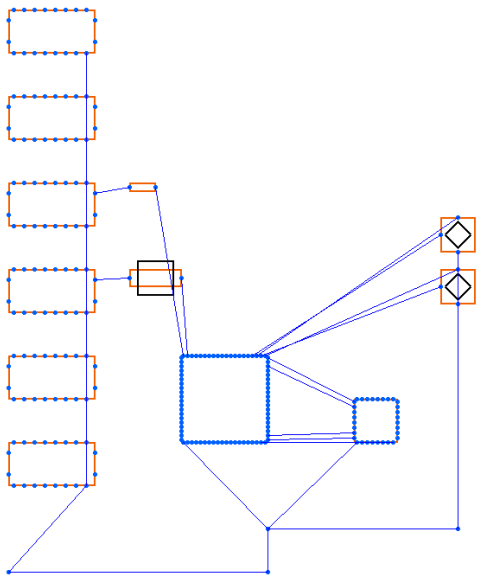

Figure 4.1: Example of generated board

Legend

- Blue dots are test points
- Orange rectangles are components
- Blue lines are nets
- Black lines are no-fly zones

4.3 First approach to the problem

Using the first board generator shown in section 4.2.1, boards could be generated in a very fast way, given few parameters; the boards generated with this approach vary in size and in number of atomic points. The most used boards are the ones with 100, 10000 and 30000 test points, and the latter has two versions: one with heavy focus on having collisions, the other having less collisions. The board with the most collisions is an invalid one, since some atomic points were generated inside the no-fly zones; the check was added later, but it can show some speed-ups in a worse case than the one using valid boards.

4.3.1 The test optimizer

In order to simplify the problem, a tester with only one probe has been selected; this is because our intention is to apply the leader/follower pattern, where the probes on the left shall be the leaders and the probes on the right act as followers, in such a way that the leader decides what is his part of the next configuration, and the followers adapt to the situation finding their positioning based on the points that they can reach.

A first hypothesis was also that the board could be split into N parts, where N is the number of probes, and each probe would be assigned to its part; since a hard rule would not work due to test blocks having the possibility of contacting near points, this should have been a *soft* rule, where the zone should have been preferable.

Since only one probe is present, test groups are, in this case, managed similarly to test equivalent points. It is not distant from their true meaning,

since the leader has to decide which is the nearest point and the follower should adapt in this paradigm.

4.3.2 Data structures

Math library Since 2D space concepts are frequently used, a mathematical library is needed in order to implement them. In the mathematical library are implemented the concept of a 3 dimensions vector, the concept of POINT (which is also used as a 2 dimensions vector), the RECTANGLE and the LINE segment. The math library is also where the GPU is mainly used, since it also manages the collision detections.

Test library It contains the TESTATOMICPOINT class, the TESTPOINT class, which is used for managing equivalent points and atomic points themselves (if a TestPoint is related only to one atomic point, it is effectively an atomic point; if it is related to more points, it manages equivalent points) and the TESTGROUP class. The TestPoint and the TestGroup have a string ID identifying them, while the TestAtomicPoint has a numerical ID representing its position in an array of atomic points.

The TestAtomicPoint class also keeps the information of incompatible points and points that can be reached by touching no-touch zones. Later, the information of what no-touch zone is involved in the incompatibility has been memorized.

Board The BOARD class has the information about the probes, the no-touch zones, the no-fly zones and the test points; it also manages the initial collision detection between the probe paths and the no-fly and no-touch zones and the high-level view for the probe movement.

Since it manages the collisions through the no-fly and no-touch zones, it contains also the pointers to the data on the GPU, managing their high-level allocation.

4.3.3 Collision detection

Since the collision detection algorithm has very few branching paths, it is reasonable thinking of an algorithm for the collision detection working on

GPU. While the algorithm stays the same, a wrapper has been built around it in order to exploit the SIMD nature of GPU computation. In order to do this, a simple way to achieve significant improvements is calculating each couple of points on each thread. This solution is more simple, selecting only one atomic point at the time and calculating the couples formed between that atomic point and the others on the board. The connection between unreachable points can be modeled as bidirectional edges on a graph, thus if the first point is incompatible with the second, we can derive that the second is incompatible with the first; because of this, a couple of points can be tested once and the information about incompatibility is complete; the board function managing the collision detection ensures that computation of incompatibility is performed only once per couple of atomic points, saving a significant amount of time. The CPU ensures, through frequent synchronization, that the GPU threads calculate the collision one at the time. This simplifies the wrapper function, yet it offloads the weight of a for loop from the GPU, which has a higher impact than on the CPU because of the lack of a branch prediction unit.

Algorithm overview and implementation

The implementation for the wrapper uses a boolean array for results. this function was used in the first implementation of the algorithm, that did not track which no-fly zone or no-touch zone was involved with the collision.

points and rects are arrays of 2D points, representing atomic points, and rectangles; rectangles are divided into two arrays, one of no-fly zones and one of no-touch zones; by having rectangles as inputs, the kernel can be used for both no-fly and no-touch zones.

In line 10 we retrieve the index of the second point of the couple, while the index for the first one is `currentPoint`. Since it does not make sense to calculate the incompatibility between the same atomic point, it is not considered in calculations, along with already calculated incompatibilities.

In this version of the algorithm, the path between points is modeled as a line, while no-fly and no-touch zones are modeled as rectangles, due to missing requirements.

Since every point is outside of every rectangle, it is easy to demonstrate

by method of exhaustion that it is sufficient to test the collision with the diagonals of the rectangle in order to establish if there is one, except for the particular case in which they are on the same side along the rectangle positions; in this case, the collision is not detected, but it is a borderline case that never happens due to the no-fly zones constraints.

The function `segmentToSegmentCollision` is the one described by the algorithm 1.

```
1   __global__ void GPU::segmentToRectangleCollision(
2       bool* results,
3       const Point* points,
4       const Rectangle* rects,
5       const size_t currentPoint,
6       const size_t currentRectangle,
7       const size_t max
8   )
9   {
10      int index = threadIdx.x + blockIdx.x * blockDim.x +
                    currentPoint + 1;
11      if(results[index] || index >= max) return;
12      Rectangle rect = rects[currentRectangle];
13      Line currentLine, l1, l2;
14      currentLine = Line(points[currentPoint], points[index]);
15      l1 = Line(Point(rect.x, rect.y), Point(rect.x + rect.w,
                    rect.y + rect.h));
16      l2 = Line(Point(rect.x, rect.y + rect.h), Point(rect.x
                    + rect.w, rect.y));
17      results[index] = segmentToSegmentCollision(currentLine,
                    l1);
18      if(!results[index]) results[index] =
                    segmentToSegmentCollision(currentLine, l2);
19  }
```

4.3.4 Experimental results in collision detection

Tests have been performed using a computer with the following specifications:

- CPU: Intel i7-9700k (8 cores, 8 threads)
- RAM: 32 GB DDR4

- GPU: NVIDIA GeForce GTX 1060

Results may vary with different configurations.

Tests have been performed with boards having 12 no-fly zones, with collisions calculated using the algorithm described above.

As we can see in table 4.1, improvemens are significant with 10000 atomic points, while we start gaining an advantage using the parallel approach already with 1000 points. Since incompatible points need to be saved in lists, however, those insertion must be atomic operation.

Because of the large number of collisions, the 30000 invalid file takes longer in the multithreaded approach, and the performance gain becomes less significant. This is due to the data structure used to represent collisions: instead of a graph represented by a matrix, in order to save memory, the representation is done thrugh a vector of indexes. The algorithm on the GPU returns the results in a static array, providing the information about the presence of the collision. Since collisions must be added in a vector, the algorithm gets slown down.

Number of atomic points	**Single thread [s]**	**Multithreaded [s]**	Speed-up	**GPU [s]**	Speed-up
100	0.004	0.006	0.67x	0.019	0.21x
1000	0.300	0.06	4.69x	0.148	2.03x
10000	17.44	9.07	1.92x	4.15	4.20x
30000	267.30	37.87	7.06x	19.43	13.76x
30000 invalid	191.14	59.75	3.20x	46.51	4.11x

Table 4.1: Empirical time measurements and comparisons between different implementations of collision detection

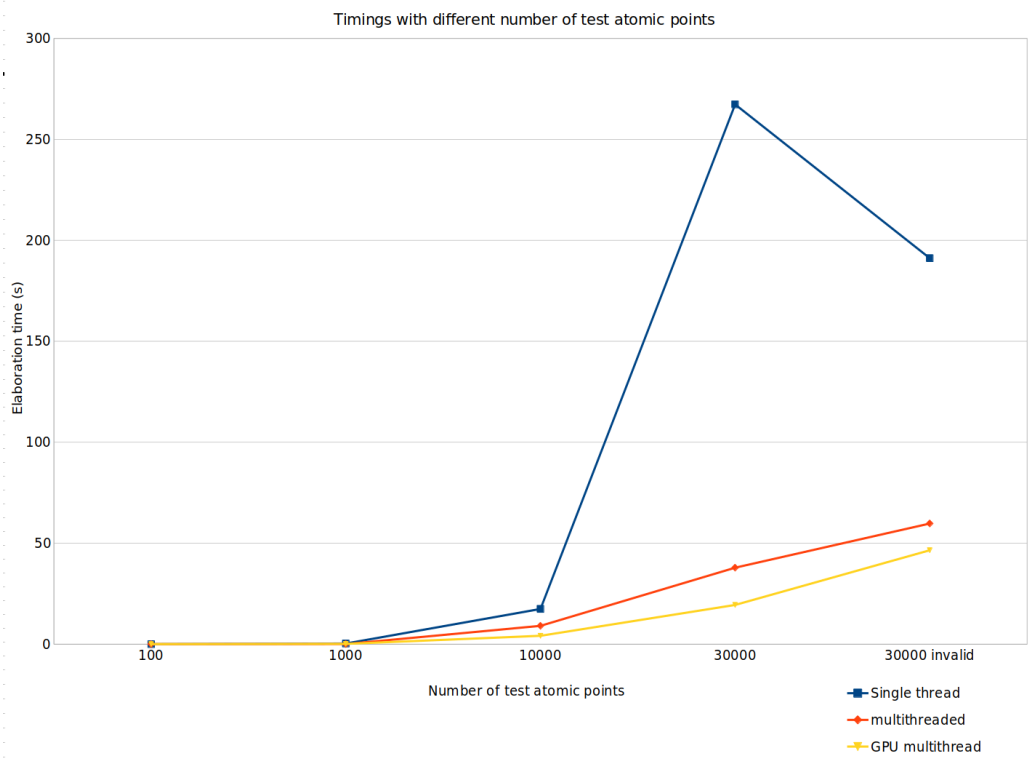

Figure 4.2: Collision detection elaboration time comparison with increasing number of atomic points

A simulation can be performed only calculating collisions and copying the results back, simulating a matrix-like graph. This version of the algorithm shows a significant improvement both over the single-threaded version and over the GPU multithreaded one. Since it does not require CPU computation, the only part that slows down the computation is the memory copying time between VRAM and RAM.

Table 4.2 shows the amount of those improvements; those improvements can also be useful in the test planning performed later, since the complexity of the search for an incompatible point becomes $O(1)$ instead of $O(N)$ because of the nature of a data matrix.

Number of atomic points	**Single thread** [s]	**GPU** [s]	**GPU using matrix** [s]	GPU speed-up compared to single thread	GPU speed-up compared with current multithreaded
100	0.004	0.019	0.008	0.50x	2.38x
1000	0.300	0.148	0.071	4.23x	2.08x
10000	17.44	4.15	0.86	20.78x	4.83x
30000	267.30	19.43	3.63	73.64x	5.35x
30000 invalid	191.14	46.51	3.61	52.95x	12.88x

Table 4.2: Comparison between single threaded approach, GPU approach and GPU using matrix approach

4.3.5 Test planning

The first approach to the test planning is simplified. In order to make a test plan, a first approximation of the probes behavior would be the leader-follower pattern. Each probe would be bound to the choice that the previous probe made; this way, p_0 becomes the most important probe to manage.

By modeling the problem this way, the first thing to do was deciding the path for one probe. In order to complete the tests, the probe should contact one point of each test block.

In this version, in order to emulate the differences between the x and y speed of the probe, they have a factor assigned to their speed. Movement has been modeled as linear both on the x axis and on the y axis in a first approximation, since this is not the path finding algorithm.

In order to find the best path, we have to define the distance metric between the test blocks considering the point of view of a single probe.

Definition 10 *The* DISTANCE BETWEEN TWO ATOMIC POINTS *is defined as the distance between two points in the Cartesian plane, with different factors between x and y axes.*

Definition 11 *The* DISTANCE BETWEEN TWO EQUIVALENT POINTS *is the minimum distance of atomic points considering each of the atomic points in the first group of equivalent ones with each one of the second group.*

Definition 12 *The* DISTANCE BETWEEN TWO TEST GROUPS *is the minimum distance of equivalent points considering each of the equivalent points in the first test group with each one of the second group.*

Having multiple probes creates a slightly different definition for the distance between equivalent points, taking into account the maximum distance for each of the probes in order to reach it.

Distance calculation is the slowest operation of the procedure, so it makes sense to optimize it in order to have a reasonable speed-up.

Possible parallel implementations

Distance between atomic points A first approach is parallelizing the distance between atomic points; CPU parallelization does not give significant performance improvements, and GPU parallelization gives 1.1x worse results on average on a single calculation.

Distance between equivalent points This approach gives rise to more waiting times, due to the cycling between the test points. Results are very similar to the previous point.

Distance between test groups Using the parallel patterns library, parallelizing the distance between test groups improves performance by a significant amount of time.

Since the program is supposed to run on Windows, in order to lose less amount of time for the context switching, any mutex is implemented by the `CRITICAL_SECTION` structure, since it does not operate in kernel space.

4.3.6 Empirical results in test planning

Depending on the number of atomic points and test blocks, given 12 collision rectangles, in table 4.3 are reported the elapsed times in order for one probe to touch every test block, using the distances defined above.

In this implementation of the algorithm, atomic point distance between incompatible points is set as infinite, in order to spend less time on the calculation of less likely paths; as a first approximation, paths are modeled as straight lines; in the path finding algorithm, it will be necessary having the probes moving using the right logic.

Number of test blocks	**Single thread [s]**	**Multithreaded with parallel patterns [s]**	speed-up
22	0.138	0.196	0.70x
160	0.042	0.046	0.91x
2156	43.33	18.95	2.29x
5046	15.52	3.53	4.40x
6507 (invalid file)	1084.26	447.26	2.42x

Table 4.3: Empirical time measurements and comparisons between single threaded and parallel pattern implementations of the test planning.

4.4 An improved approach to the problem

As a work in progress, a second approach has been taken into consideration for the whole planning task. This approach follows a more in-depth scenario, where no-fly zones and no-touch zones are modeled as generic polygons instead of rectangles, while paths between points are modeled as rectangles.

4.4.1 Data structures

Data structures remain almost the same. No-fly zones and no-touch zones are represented by a `std::vector` of line segments, while paths are calculated at runtime using a temporary rectangle structure. The declaration of polygons is a simplified version of the SVG polygon declaration format.

Configuration In path finding, an important class is the configuration, which replaces the old vector of points to be contacted in order to execute the tests. Another use of this class is for path finding, where collisions have to be avoided and the path is colliding with a no-fly or no-touch zone, providing a safe path for all the probes. A valid configuration respects the constraints given in section 4.1.3. Probes can only move to one valid configuration to another valid configuration. Intermediate results are stored in a vector, also used to compute distance more precisely.

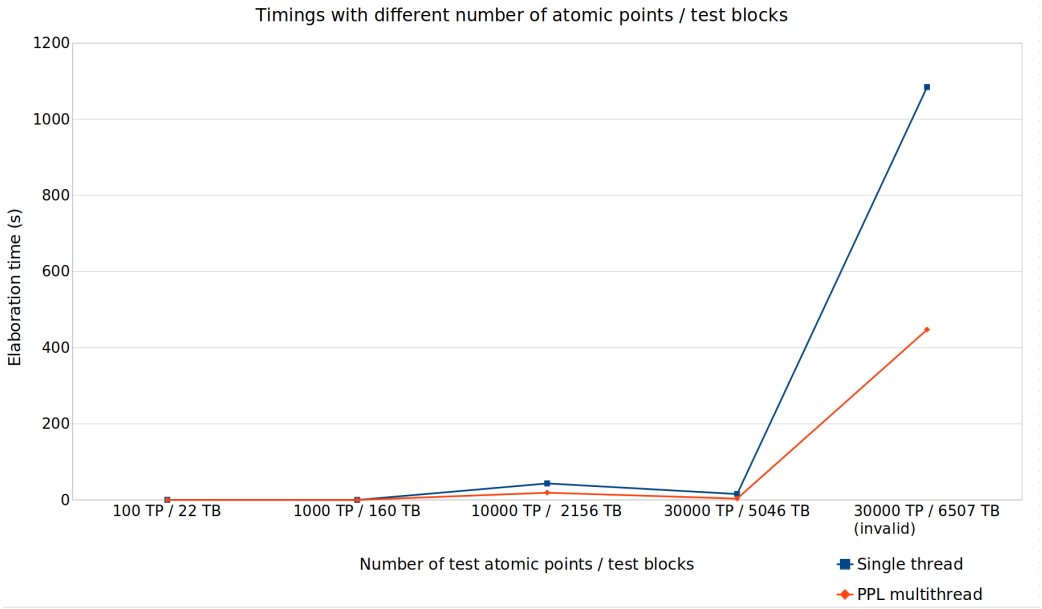

Figure 4.3: Graph calculations timings with increasing test blocks

4.4.2 Path finding

The problem of path finding is, given a valid starting configuration and a valid final configuration, finding, if there are any, all the intermediate configurations in order to pass from the first one to the second one.

Considering the valid configurations, we individuated three main categories of paths that can be taken:

- a direct path, with no intermediate step. This is the easiest case, where probes can move directly from one configuration to another;
- a path colliding with a no-fly zone that can be solved by splitting the line segment joining the 2D points in more segments. This allows the probe to form smaller rectangles, providing a more accurate control of the probe;
- a path which collides with a no-fly zone. In this case, a collision avoidance algorithm is needed in order to move the probes;

For no-touch zones, the path finding is easier, since in the worst case scenario the probes only need to be raised higher than usual.

Collision avoidance

Since collisions may happen in various, an algorithm is needed in order to avoid no-fly zones. Regarding the no-fly zones avoidance, we identified two main situations that can occur:

- probes can avoid the no-fly zone without moving in the opposite direction with respect to each other, but the path needs to be split in more valid configurations;
- at least one probe is moving in the opposite direction with respect to the others.

Taken separately, these cases are not difficult to manage separately; yet, they intersect if a new collision may happen when passing from one configuration to another, or if one of the possible destination points is inside a no-fly zone. In order to solve those cases, we are likely to need computational power; in this case, a runtime collision detection using the GPU is likely to save time; it is also possible using a CPU + GPU multithreaded approach, where each thread of the CPU manages one probe.

4.5 An alternative approach

4.5.1 About dividing the free space

A possible approach may come from focusing on the space that the probe is allowed to move freely. Within those *free zones*, it is possible for the probe to move without having to worry about collisions; those zones should also share a link between them, in order for the probe to be able to pass from one to the other.

By using this approach, there would be no need for collision detection between paths and no-fly zones, since it is implicit that two linked areas would not cause a collision through any no-fly zone. Since no collision can occur through no-fly zones, any path linking two or more zones is a valid one, and only constraints between probes need to be applied.

The advantage of automatic collision avoidance comes with a disadvantage: if a probe passes through more than one zone, it will likely stop more

than once, slowing down the test execution. This situation may rise when few points are left to be tested, requiring long movements by the probes. In order to solve this problem, it might be possible to use different techniques:

- collision detection might play a role, but instead of testing every single point it would be limited on testing few key points in zones;
- simulating ray casting in order to determine the reachable area at runtime;
- increasing the number of links between zones, creating an undirected graph linking different zones. This would require an initial computation in order to build the graph, but it would ensure a faster runtime computation.

4.5.2 About free zones

Free zones should have a fixed shape in order not to create computational overhead. A rectangle would seem a good solution, since it is an easy shape to manage, but since the no-fly zones have generic shapes it is unlikely to be the best option in order to approximate free spaces without colliding areas; the solution may come from the graphical field, where triangles are used to cover entire sections of space without leaving gaps. Triangles are more difficult to manage, but it is possible to cover the whole free space on the board by using them.

4.5.3 Dividing the free space

Free space needs to be divided into triangles. This process is called TRIANGULATION, and many algorithms have been developed for this purpose, from generic ones to ones that take advantage of special situations. As we do not know the shape of the free space with the no-fly zones, the polygon is generically non-monotone. In order to triangulate the board, the sweep-line approach can be used in order to divide it into monotone sub-polygons.

4.6 Conclusions and future work

Even an unoptimized GPU approach can provide a significant speed-up of the collision detection, providing possibly a viable runtime calculation of collisions instead of a pre-processed one. This way it would be possible to save a large amount of memory and have only local data providing info about collisions. The path finding problem can also benefit from a multithreaded approach, independently moving each one of the probes with due synchronization.

Some future work can be implemented starting from the theory described in section 4.5, which we think is a viable approach. Improvements and optimizations can also be performed on the original collision detection.

Chapter 5

EVCD analyzer

5.1 The problem

Chips are increasing in size and software simulations are a major concern for testing and reliability. While simulation tools are on the market, results are analyzed separately. Simulation tools, as output, write an EVCD file (Extended Value Change Dump), which is an extension of the VCD[7] format (Value Change Dump). EVCD files proved to be more reliable than ATPG (Automated Test Pattern Generation) engines in order to reach the desired accuracy of the analysis; thus, simulation dump analysis is a more used strategy in the field [13][8].

Tools exist and have been developed for analysis of EVCD files; a tool from professor Paolo Bernardi is available for this type of analysis; the request is building a new tool which should use the least RAM possible while exploiting the power of the multi-core approach to speed up the different types of analysis. The proposed approach is a multithreaded computation, which would decrease the time needed to perform both full and statistical analysis.

5.2 File structure

5.2.1 EVCD files

An EVCD file is formatted as follows:

- the header section with generic information;
- the declaration section, where all the relevant gates, buses and modules are declared and assigned to an ID represented by an increasing number;
- the initial values assignment section;
- the simulation section

5.2.2 Header

```
$date
    Date of creation
$end
$version
    Tool used
$end
$timescale
    Time scale
$end
```

While the importance of those information is low, the timescale can be useful in future analyses.

Gates and modules declarations

Each gate lives within a module. Modules represent parts of the chip in a hierarchical structure. Modules can have a parent-child relationship or a sibling relationship. The declaration follows the pattern:

```
$scope module module_name $end
$scope module child_module_name $end
...
$upscope $end
$scope module child_sibiling_module_name $end
...
$upscope $end
...
$upscope $end
```

Within each scope, gates and buses are declared. The difference between a bus and a gate consists in how many signals define its state: a gate state is defined by only one signal, while bus states contain many signals.

A gate declaration has the following syntax:

```
$var port     1 <ID   gate_name  $end
```

A bus declaration has the following syntax:

```
$var port     [first:last] <ID   bus_name  $end
```

FIRST and LAST are integer numbers defining the range internal bus IDs. To discern the internal identifiers of a bus from what we call IDs, those will be called SUB-IDs.

5.2.3 Assignments and values

A GATE value assignment has the following syntax:

```
Value <ID
```

A BUS value assignment has the following syntax:

$\texttt{Value}_{subID_0}\texttt{Value}_{subID_1}\texttt{...}$ <ID

Allowed values for both buses and gates are:

- **0**: logical value;
- **1**: logical value;
- **x**: don't care value; it means it can be either 1 or 0;
- **z**: high impedance value.

Initial values

```
$enddefinitions $end
#time
$dumpports
assignment
assignment
...
$end
```

where *time* is a positive integer number. This section must contain assignments for each gate and bus.

Simulation

```
#time
assignment
assignment
...
#time
assignment
...
```

The simulation section is similar to a repeated assignment section, while having the following rules:

- times are strictly increasing;
- IDs and values only appear if the state changed in the current time; bus assignments might show repeated sub-ID values when they appear.

5.3 Definition: full analysis

A full analysis of a simulation result, given a EVCD input file, transforms the time to list of changes relationship grouping, instead, changes with the respective bus or gate; this simplifies the study of the full toggle activity on a single gate or bus. The main problem of the process is its duration with files that have the tendency to grow in size; dimensions of the files can vary from few megabytes to hundreds of gigabytes.

The output states can be F (falling edge) if a bus sub-ID or gate toggles from any value except 0 to 0 or from any value to x, or R (raising edge) if the toggle is from any value except 1 to 1.

The output format has the following syntax for buses:

```
bus_path <ID-sub_ID time state
```

and the following syntax for gates:

```
gate_path <ID--1 time state
```

Bus path and gate path are the modules of the chip, written as a Unix-like file system path.

5.4 Definition: statistical analysis

A statistical analysis of a simulation results gives partial information about the toggle activity; more precisely, it tells if the signal value changes follow the pattern 1-0-1 or 0-1-0. If the full pattern is matched, then the coverage of the toggle analysis is 100% of the possible significant changes; if only half of the pattern is matched, the coverage is 50% of the possible changes; otherwise, the coverage is 0%.

we do not consider toggle from a value to x or z in the statistics, since ideally they would appear only as initial values and should not appear when the system is fully operational. Cases in which those values appear as initial values are managed by substituting them with the first change found.

5.5 General approach

The first approach to the problem was developing a new single-threaded version of the program. Starting from professor Bernardi original tool, in order to ease the development and avoid memory leaks wherever possible, the first version of the new tool is a simple C++ port of the old parser written in C.

5.5.1 Original approach

The original approach uses an hash map in order to memorize and retrieve signals, while each change is put in a linked list structure. The advantage of the hash map is the capability of being general purpose and working with strings, while providing access time of O(1) complexity; while this is a good general solution, in practice the signals found on the EVCD files are identified by increasing numbers. Moreover, signal references are inserted in a hierarchical structure provided by an N-branches tree, which requires a significant amount of RAM with more and more signals being added. In section 5.5.3 an overview of the data structures used is shown.

A remarkable memory issue happens when changes get pushed to the lists contained in the signal structure, the new state is represented by a character dynamically allocated, in order to have the possibility of having a NULL value

as an invalid change.

5.5.2 General optimizations

File parsing: sscanf

General details While sscanf is a powerful function, its nature of being general-purpose function based on formatting leads to slower performance. A first optimization was the removal of sscanf, replacing it with the correspondent `std::strtoull` function calls. A basic unoptimized version of the algorithm substituting the `sscanf` call is 5 times faster on average, as shown in figure 5.1.

```
Strtoull time:  0.000001 s
Sscanf time:    0.000006 s
Strtoull time:  0.000001 s
Sscanf time:    0.000005 s
Strtoull time:  0.000001 s
Sscanf time:    0.000006 s
Strtoull time:  0.000001 s
Sscanf time:    0.000005 s
```

Figure 5.1: Empirical durations of sscanf compared to strtoull on the same data

Technical implementation Below, we can see a short example of the code changes between the usage of `sscanf` and the usage of `std::strtoull`.

`sscanf` code:

```
1  int32_t index_start, index_stop;
2  sscanf(line[3], "[%d:%d]", &index_start, &index_stop);
```

`std::strtoull` code:

```
1  int32_t index_start, index_stop, i, wordIndex;
2  for(i = 1, wordIndex = 0; line[3][i] != ':'; ++i) {
3      word[wordIndex++] = line[3][i];
4  }
5  index_start = std::stoull(word);
```

```
6       for(i = i + 1, wordIndex = 0; line[3][i] != ']'; ++i) {
7           word[wordIndex++] = line[3][i];
8       }
9       index_stop = std::stoull(word);
```

File reading

General idea While `fscanf` is already a well working function to read files, the formatting required using `sscanf` can take time to be executed. Two approaches have been explored: the usage of **memory mapping** and the usage of the `read` Unix system call.

Memory mapping Memory mapping directly maps an arbitrary number of bytes of the file in RAM. This allows the operating system to automatically manage the file loading in memory, accessible as an array of bytes. This approach proved to be slightly faster than the `sscanf` function, yet it comes with the downside of having a maximum number of file mappings available per process; this means that large enough files can never be read completely, providing only partial information. Another downside of the memory mapping is that the system call performs lazy loading of the file, moving the data into RAM only when requested; while lazy loading is something we can avoid by requesting the data soon after the `mmap` call, the issue above still remains.

Binary file reading Using this approach, each byte is copied in memory when `read` is called. This call is faster than the memory mapping, since it does not use lazy loading, and can be called as many times as the programmer wants. The absence of lazy loading is also useful to use a double buffering technique, similar to the one used on GPUs: while a single part of the file is being elaborated, the next one is being loaded into memory. This way, the file loading does not slow down the computation. This is achieved through having two FileIterator structures, one for each file part that has to be elaborated. By swapping the FileIterators when necessary, the elaboration works without stopping. The speed-up achieved by this approach is hardware-dependent; testing the double buffering on an SSD with the 6 GB file, the process is about 3 seconds faster. On a traditional hard drive,

however, more than 10 seconds of speed-up have been achieved by using this technique.

Smaller optimizations

String length whenever needed is offered by the file reader structure; this way it does not need to be calculated later, since the value is already available when words are copied in memory.

In order to further optimize the program, `std::stoull` has been replaced with a custom function that does not require the calculation of the string length.

5.5.3 Main data structures

Signals Each gate or bus has been represented by a class called SIGNAL, identified by an ID and a sub-ID. During the reading of Signals in the declaration section, each Signal is pushed in a standard vector; a parallel structure is also filled, where the position of the first signal with a particular ID is memorized in order to decrease the computational cost of the Signal search from **O(N)** to **O(1)**.

Changes Each Signal contains a dynamically allocated array of CHANGES, which represent the state of a signal at a given time. The timing reason for not using the list is having all the values in cache when writing the output to file; while the insertion might be slower, due to the copy of the vector when reallocating it, it is noticeably faster when writing the results to an output file, since an array always exploits cache memory, due to it being a contiguous location in RAM.

File iterator Since the standard C function for file I/O are slow, a FILEITERATOR class has been created, using the native Unix I/O system calls. Since the C functions have been replaced, the class takes care of reading **N** bytes of the input EVCD file; after the reading, each operation is done directly in RAM.

RAM optimizations

Data structure optimization Data structures are declared taking into account memory alignment and optimizing smaller char values. More optimizations can be performed by using bitwise operations in conjunction with union structures, yet this would increase the effort for the programmer in order to be able to insert and retrieve change values.

Dynamic memory allocation The RAM cost of dynamic memory allocating a state change is of 8 bytes for the pointer + 1 byte for the character + 7 bytes of alignment, making it 16 bytes; moreover, the dynamic RAM allocation costs time due to its need to request RAM from the OS. A small but effective optimization for RAM consumption is using a statically allocated character, using in total 8 bytes due to alignment, and assigning an unused value as invalid.

Hierarchical structure The hierarchical structure has been substituted by a dynamically allocated array of signals coupled with another one of integers due to the offset generated by the SUB-ID presence. Since the ID is the reference for each signal in the EVCD file, the hierarchy is preserved only through memorization of the full hierarchy as a string resembling a Unix path.

Avoiding lists As mentioned in section 5.5.3, the data structure of choice for memorizing changes is an array. Another advantage of the array over the list is the RAM usage. A single-linked list uses 8 byte per each change in order to point to the next one. Since each change contains 16 bytes of data, the RAM usage of the link would be 33% of the CHANGE structure. Since changes are many more than signals, the RAM usage would drastically increase.

5.6 Full analysis

5.6.1 The sequential approach

The sequential approach follows the old C program parsing rules, while adding directly in the right Signal structure each Change found already in the format of the output.

5.6.2 The producer-consumer approach

A first implementation of the parallel version of the algorithm was using a producer-consumer approach, where the producer is the file reader, sends the line to the consumer through a thread-safe FIFO queue, and the consumer takes care of the parsing. This pipeline is unbalanced, resulting in the consumer spending more time to elaborate a single line than the producer to read it from memory. This lead to the queue being full most of the time. This lead to computation times being up to 7 times longer than the sequential approach, being similar to its timings only with a long enough queue (around 10000 elements).

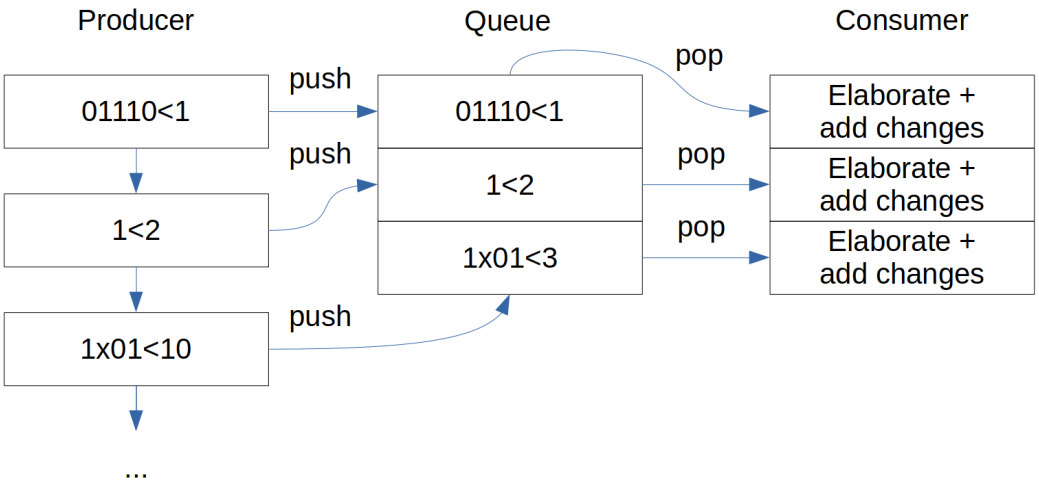

Figure 5.2: The principle of the producer-consumer approach

One producer, many consumers In order to leverage the work for the consumer, a possible improvement could be adding consumers. This ap-

proach proved to be unsuccessful, since it lead to longer parsing time; the culprit, in this case, is the waiting time in context switching: profiling tools show that the waiting time for the threads keeps increasing and the queue has the tendency to be filled before those waiting times expire.

Since the insertion of each change must be in order, the approach chosen is for each thread filling a temporary queue, which will be later elaborated by the main thread.

Avoiding system calls A possible optimization can be using busy waiting instead of fully relying on the scheduler expiration times for the waiting. This approach turned out to be more successful than the previous one, yet it was not enough to reach the performance of the sequential approach.

5.6.3 The file part splitting approach

Instead of reading each line and then leaving the heavy computation to a secondary thread, this approach is about splitting the loaded file part in almost equally-sized chunks and leaving the computation of each smaller chunk to the threads involved.

The file division problem By splitting the file equally among threads, we can obtain three different situations:

- the first line that a thread elaborates is a time information
- the time information is not in the first line
- the time information is not present in the section of file managed by the thread

While the first is the ideal situation, it is unlikely that it is always the case; when the second case or the third case happens, the time information for the signal changes is lost. In order to solve this issue, each thread fills a queue with partial information about the signal; when the working threads complete the elaboration on their divided file, the main thread starts reading the values in each queue, always memorizing the last valid time information and filtering changes that do not represent a toggle. By keeping the last valid

time information, it is possible to reconstruct the following ones; in order to achieve this, queues have to be read in order.

The main thread The logic for the main thread consists in launching the threads, starting them through a barrier mechanism, computing the last line (since each FileIterator holds a buffer unaware of lines, it might be necessary swapping the iterators to read the entire line). After the reading, the loading of the next part of the file is done completely in parallel with the remaining threads, and the main thread waits for their computation to be finished through an optimized barrier technique: for each thread, it waits for an ending in-order, so that the elaboration can possibly be parallel to the other threads work. For each ending thread, the main one empties the relative queue and fills the Signal structures with those temporary changes. In figure 5.3 a parallelism between the CPU stages of the pipeline and the file elaboration has been developed:

- FETCH: this stage is represented by the input file reading from disk;
- DECODE: this state is represented by the division of the work for the threads;
- EXECUTE: represented by the file parsing; this stage is executed in parallel by the worker threads;
- MEMORY: represented by the main thread filling the correct structures; this is the longest stage in the pipeline in terms of duration, since it is work that cannot be performed in parallel; while small optimizations have been developed to reduce its length, it still takes a significant time of the execution;
- WRITE-BACK: represented by the final result being written on disk.

The worker threads The other threads do the same parsing as before, but fill temporary queues in order for them to be processed by the main thread. An optimization has been done for the first worker thread: the order clause is already satisfied, since it works on the first chunk of the buffer, it

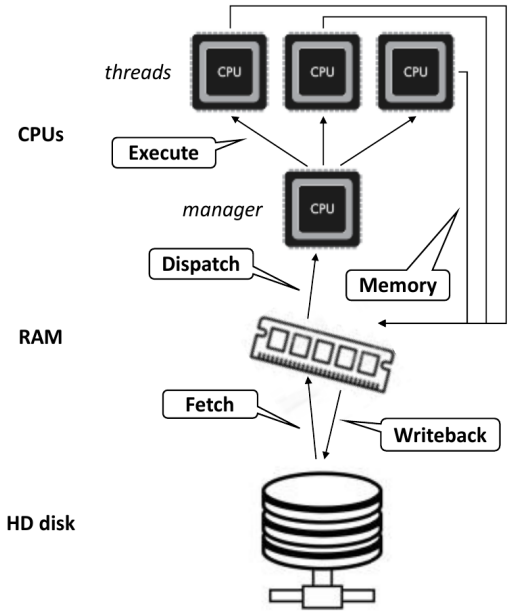

Figure 5.3: The pipeline stages

directly adds the changes to the Signal structures and updates the current time.

5.6.4 Experimental results

Experimental results are retrieved using the file part splitting approach, on a server with the following specificatons:

- CPU: AMD EPYC 7301 16-Core Processor (16 core, 64 threads)
- RAM: 128 GB
- Magnetic HDD

Files which are not of 600 MB, 6 GB or 66 GB are simulations with an increasing number of changes based on the 66 GB file; because of this, the multithreaded approach gives us worse results, since the number of signals is much larger than the number of changes, representing the worst case scenario; however, the most frequent case is the one where we have the number of

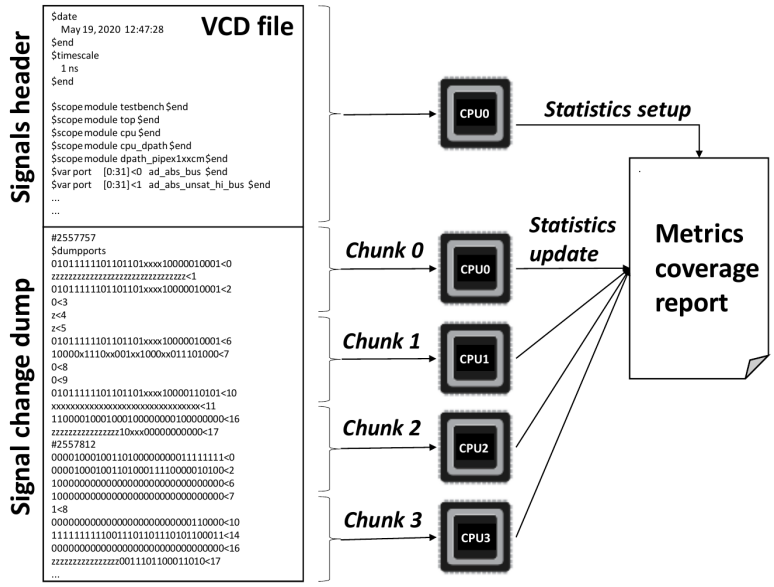

Figure 5.4: The main idea of how the threads elaborate the file

changes much larger than the number of signals. Table 5.1 shows the timings obtained using different files with different sizes, using the reconstructed time of the original program; speed-ups are measured comparing with respect to the original approach. It is possible for some cases that the program fails to execute: this happens because the program ran on a safe environment provided by RUNLIM, limiting the RAM usage to 120 GB.

Since files larger than 6 GB can not be analyzed using the original approach, speed-ups and comparisons in table 5.2 are measured with respect to the single threaded approach.

a [GB]	**b** [s]	**Single thread** [s]	*	**4 threads** [s]	**	**8 threads** [s]	***
0.6	67.01	13.55	4.95	7.99	10.00	6.68	10.03
1.2	122.52	48.84	2.51	43.54	2.81	45.02	2.72
1.5	170.69	56.66	3.01	51.42	3.32	49.54	3.45
1.8	229.57	65.71	3.49	58.15	3.95	56.96	4.03
2.1	325.87	83.63	3.90	66.92	4.87	58.97	5.53
2.5	346.87	78.59	4.41	73.35	4.73	65.02	5.33
2.8	332.41	86.07	3.86	74.31	4.47	66.77	4.98
3.0	354.93	94.52	3.76	79.70	4.45	77.03	4.61
3.4	416.58	104.11	4.00	88.80	4.69	85.88	4.85
3.7	490.74	114.11	4.30	108.15	4.54	86.59	5.67
4.0	585.11	125.66	4.66	115.24	5.08	89.84	6.51
4.3	610.04	144.59	4.22	94.12	6.48	114.72	5.32
4.6	722.65	160.18	5.51	129.10	5.60	118.32	6.11
4.9	716.39	153.64	4.66	139.58	5.13	90.07	7.95
5.2	753.24	160.41	4.70	115.02	6.55	105.01	7.17
5.5	792.94	163.15	4.86	128.99	6.15	99.53	7.97
5.8	N/A	171.05	N/A	135.46	N/A	132.95	N/A
6.0	715.68	214.74	3.33	150.28	4.76	119.27	6.00

[a] File size
[b] Original approach
[*] Speed-up single thread VS original approach [times]
[**] Speed-up 4 threads VS original approach [times]
[***] Speed-up 8 threads VS original approach [times]

Table 5.1: Empirical time measurements and comparisons between the original approach (reconstructed) and the optimized ones on small files

Figures 5.5 and 5.6 provide a graphical overview of the elaboration time trend with the increasing of the file size.

File size [GB]	Single thread [s]	4 threads [s]	Speed-up	8 threads [s]	Speed-up
10	309.00	199.23	1.55x	170.46	1.81x
20	639.81	340.49	1.88x	490.84	1.30x
29	1012.83	677.22	1.50x	526.95	1.92x
38	1482.35	885.97	1.67x	772.29	1.92x
47	1834.59	1148.19	1.60x	972.44	1.89x
57	2305.86	1359.37	1.70x	1421.55	1.62x
66	2840.83	1523.77	1.86x	1480.52	1.91x
67	2350.59	1802.15	1.30x	1294.69	1.82x

Table 5.2: Empirical time measurements and comparisons between the single threaded approach and the multithreaded ones on large files

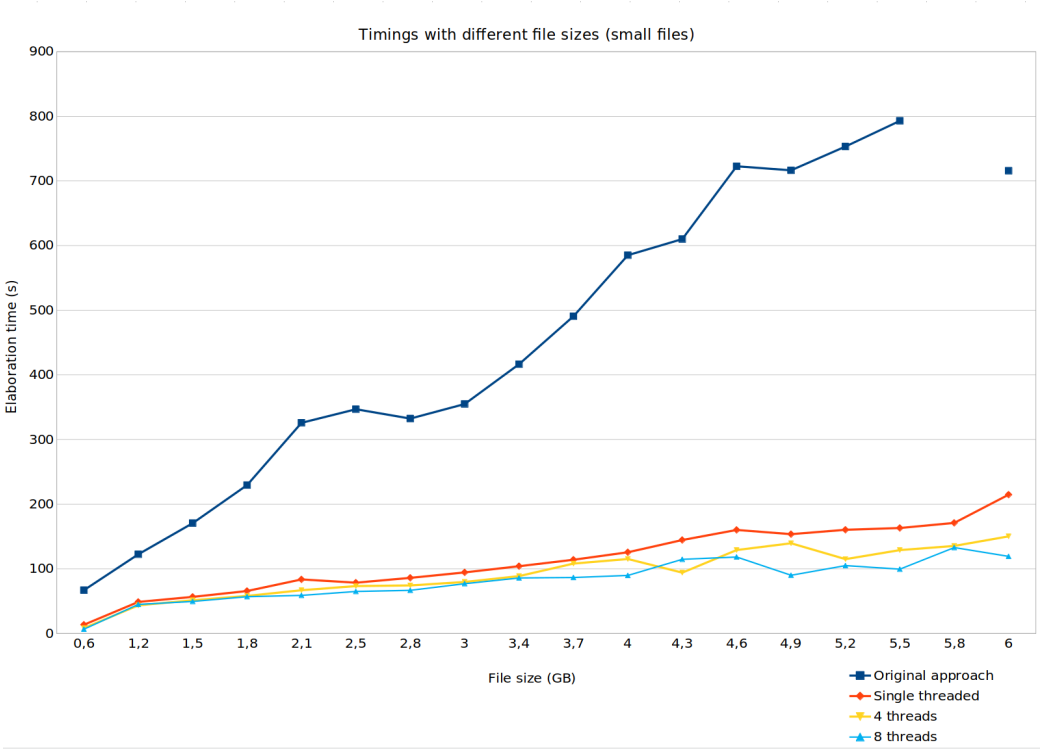

Figure 5.5: Timing trends with increasing file size (small files)

RAM usage Table 5.3 shows the RAM usage for each file size, comparing the original approach to the single threaded and the 4 threaded ones. some files, the RAM usage could be imprecise; this happens because the servers

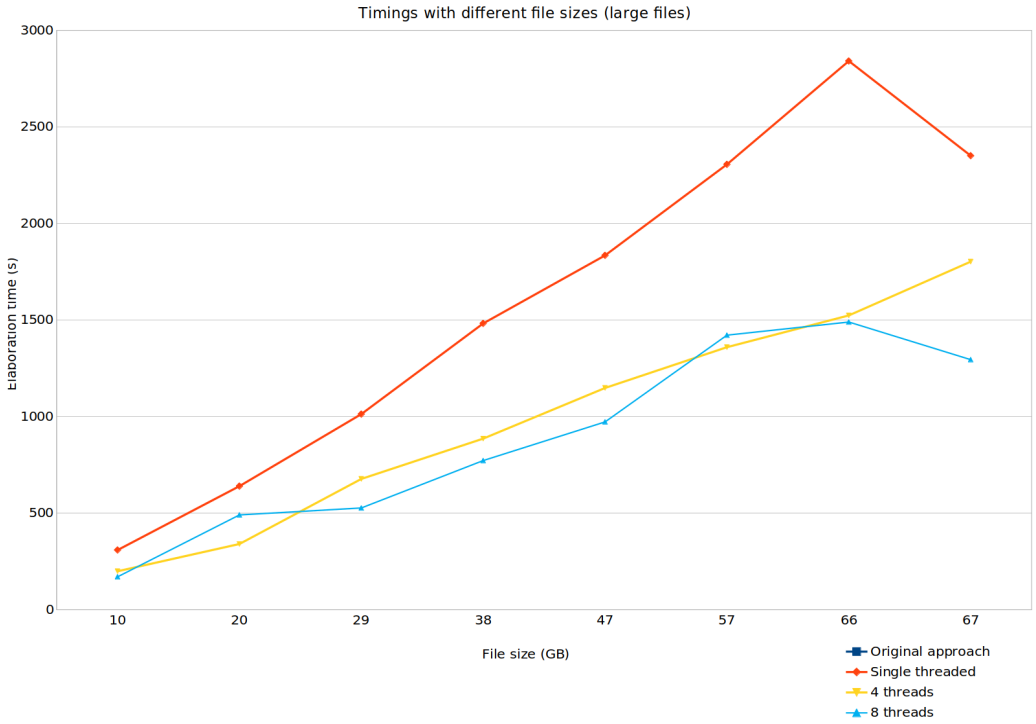

Figure 5.6: Timing trends with increasing file size (large files)

starts swapping in order to keep the memory usable. In some cases, the original approach is not suitable anymore for the full analisys, since the program ran with resources artificially limited to 120 GB by using RUNLIM.

When the number of changes increases, it is possible to notice that the amount of RAM required is approximately 100 MB per each thread running in the background; an exception is the 6 GB file, which uses much more RAM than forseen, while still remaining a valid approach for the file analysis. Tests using 8 threads confirm this trend.

File size [GB]	**Original approach [GB]**	**Single thread [GB]**	**Improvement [times]**	**4 threads [GB]**
0.6	6.5	0.7	138.27	1.0
1.2	103.6	6.8	15.25	6.9
1.5	103.6	6.8	15.15	6.9
1.8	103.6	6.8	15.15	6.8
2.1	103.6	6.9	14.94	7.2
2.5	103.6	7.3	14.14	7.6
2.8	103.6	7.8	13.29	8.1
3.0	103.6	8.1	12.82	8.4
3.4	103.8	8.5	12.25	8.8
3.7	105.9	8.9	11.84	9.2
4.0	108.4	9.5	11.36	9.9
4.3	108.4	9.8	11.01	10.2
4.6	112.9	10.1	11.20	10.4
4.9	115.2	10.3	11.20	10.6
5.2	117.5	10.5	11.15	10.9
5.5	119.8	10.9	10.95	11.3
5.8	N/A	11.5	N/A	11.8
6.0	70.3	6.0	11.70	7.6
66	N/A	75.2	N/A	75.6

Table 5.3: Empirical timing comparisons between the original approach (reconstructed) and the optimized ones

Figure 5.7 shows how much the RAM usage tends to increase between the various scenarios. While the RAM usage is remarkably high on worst-case files, it is possible to see a trend with the increase of the number of changes in signals.

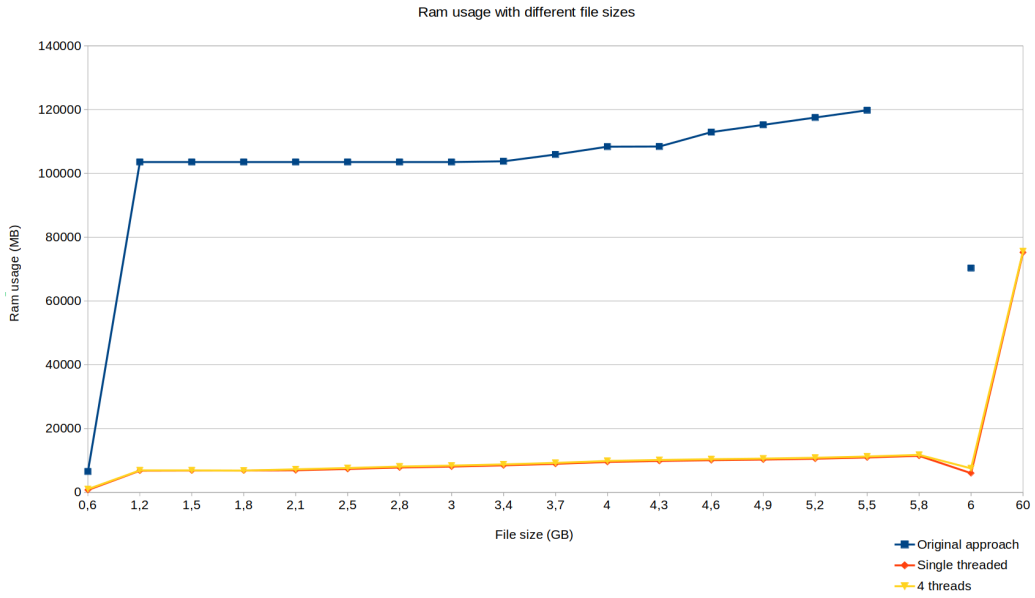

Figure 5.7: RAM trends with increasing file size

Time data considering only the simulation part is not shown, due to the fact that the signal declaration and initialization are much less significant in this case. They are, however, much more significant in the statistical analysis.

5.6.5 Considerations about full analysis

Advantages become less significant when threads are added above 8.

An approximate calculation of the Amdahl's law shows the theoretical limit to the speed-up; with the aid of a profiler, the complete analysis runs in parallel around the 70% of the time (with few oscillations).

In the current state, the theoretical speed-up limit is between 3 and 3.5 times using the formula 2.2; using the formula 2.1 mentioned in section 2.3 with 8 threads, the theoretical limit is 2.96. Amdahl's law does not take into account many parameters such as how synchronization is implemented; because of this, it is only a theoretical limit which is reachable in ideal cases.

Timings and speed-ups refer to the complete execution of the program. Files with smaller size (except the 0.6 GB file) spend most of the time loading signals instead of loading and elaborating changes. Parallelization advantages

tend to be limited considering those files, yet as files grow in size it is more and more noticeable the improvement of the parallel implementation.

The algorithm is mainly stuck on a serial operation; considering the span of the execution stage and the one of the memory stage, more in-depth experiments show that the slowest part is the memory stage, being almost balanced by using 4 working threads for the execution part, representing most of the span of the operations. The in-order insertion of of values from the queues is only doable in a single-threaded environment in order not to order the values later; With 8 threads, the memory stage is between 1.5 and 4 times slower than the execution stage; increasing the number of threads linearly decreases the execution time of the parallel part.

A possible approach that might be worth trying is one where every insertion is done out-of-order, unifying the memory stage and the execution stage, leaving a final *sorting stage* that can be processed in a parallel fashion. This, however, can cause a memory overhead in the case of repeated bus values; moreover, because of this case, it is necessary to implement a data filter when writing the output on file.

5.7 Statistical analysis

5.7.1 The general approach

The declaration and the initial assignment sections work the same way as the full analysis ones, while the Signal structure and the Change structure received drastic changes.

The CHANGE structure now contains two data: the time and a sub-time. Since the time given by the simulation result is not needed in this analysis, in this case the time and the sub-time are data related to threads; they are needed because the order in which we find 0 or 1 still matters, because of the existence of the buses.

The TIME represents the thread index; this allows the program to understand in which chunk of the partial file a change has been found.

The SUB-TIME is an internal counter and gives information about the order in which a change has been found in the same thread; this is mandatory if we do not assume that a change appears only once per thread.

The SIGNAL structure for the statistical analysis contains, instead of a dynamic vector of changes, a static one: we only need two toggle events to cover all the cases needed for this analysis. Value changes are added using the following rules:

- CHANGES[1] contains the change with the same final value as the initial value;
- CHANGES[0] contains the change with the opposite final value compared to the initial one;
- if the starting value is not 0 or 1, the first change replaces it with its value;
- only the first time CHANGES[0] has been found is relevant;
- CHANGES[1] is overwritten until a value is found having the time reference > CHANGES[0]

5.7.2 The file part splitting approach

The approach is similar to the full analysis, with small changes.

The Signal logic for adding changes allows each thread to work completely concurrently, being synchronized only when the new file chunk has been read and split. This way, we avoid the MEMORY stage on the main thread, dividing the work among all the threads.

5.7.3 Experimental results on statistical analysis

Full time analysis

Table 5.4 shows improvements on the whole program; figure 5.8 shows the results in a graphical way, focusing on how multithreading effectively improves the analysis time. It is noticeable how timings are similar with files of smaller size, while the difference increases with larger files. In particular, the 16 threads approach is the one that gives more advantages overall.

File size [GB]	**Single thread** [s]	**16 threads** [s]	speed-up [times]	**32 threads** [s]	speed-up [times]
0.6	3.58	1.60	2.24	1.13	3.17
1.2	44.39	46.22	0.96	42.28	1.05
1.5	49.48	43.53	1.14	44.07	1.12
1.8	49.38	44.65	1.11	47.75	1.03
2.1	54.30	45.47	1.19	44.91	1.21
2.5	56.46	50.02	1.13	48.08	1.17
2.8	62.43	46.63	1.34	49.50	1.26
3.0	62.18	48.40	1.28	47.99	1.30
3.4	69.09	49.55	1.39	48.29	1.43
3.7	68.53	48.18	1.42	47.94	1.43
4.0	68.45	46.25	1.48	47.75	1.43
4.3	87.64	52.87	1.66	49.37	1.78
4.6	87.37	49.91	1.75	50.64	1.73
4.9	82.81	47.66	1.73	55.30	1.50
5.2	88.05	52.20	1.69	51.34	1.72
5.5	86.66	52.61	1.65	49.40	1.75
5.8	83.11	57.48	1.46	50.43	1.67
6.0	58.04	13.26	4.38	11.50	5.05
10	115.97	58.43	1.98	57.49	2.02
20	118.96	72.89	2.56	72.06	2.62
29	269.37	88.53	3.04	89.01	3.03
38	320.51	103.35	3.10	101.98	3.14
47	366.78	116.88	3.14	140.64	2.61
57	465.25	131.41	3.54	166.79	2.79
66	498.79	148.89	3.35	264.89	1.88
67	487.05	142.11	3.43	219.64	2.22

Table 5.4: Empirical timings and improvements of the whole statistical analysis

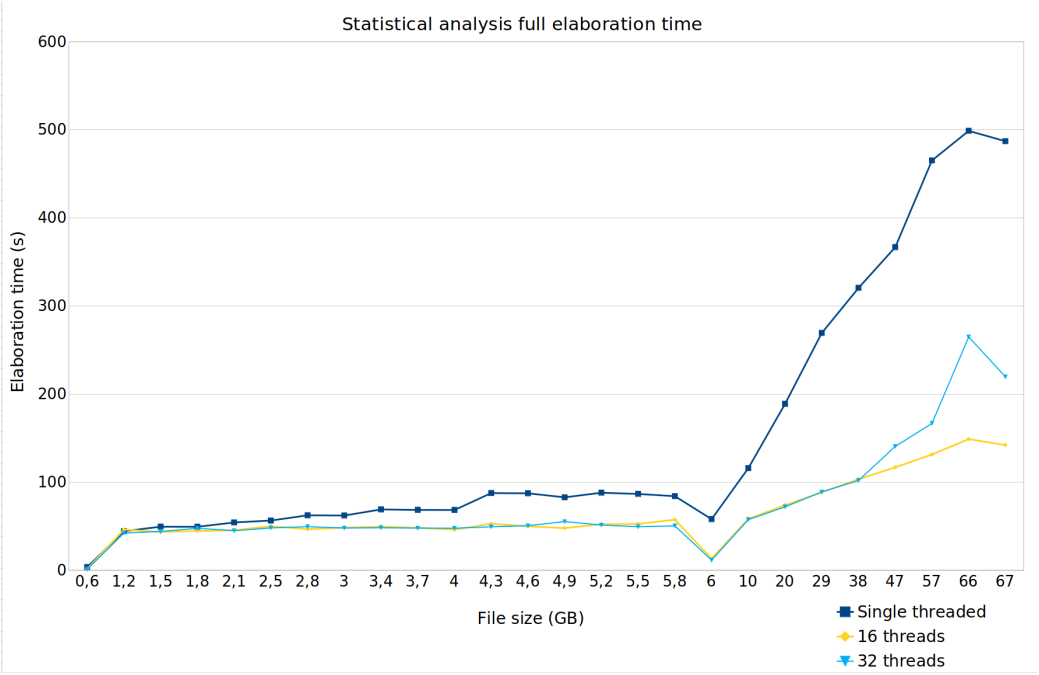

Figure 5.8: Graph statistical analysis improvements on whole analysis

Parallel part analysis

Advantages of the parallel approach are also confirmed by table 5.5 and figure 5.9, showing only the improvements of the parallel part of the statistical analysis.

The parallel part consists in the elaboration of the change list, having the only synchronization when the next part of the file has been read and is ready to be elaborated.

File size [GB]	**Single thread** [s]	**16 threads** [s]	speed-up [times]	**32 threads** [s]	speed-up [times]
0.6	3.06	1.05	2.91	0.61	5.02
1.2	0.0007	0.002	0.35	0.003	0.23
1.5	4.43	0.46	9.63	0.58	7.64
1.8	5.74	0.95	6.04	0.91	6.31
2.1	10.01	1.70	5.94	1.61	6.27
2.5	12.26	1.95	6.29	1.87	6.56
2.8	19.09	2.61	7.31	2.41	7.92
3.0	16.50	3.03	5.45	2.84	5.81
3.4	23.61	3.45	6.84	2.09	11.30
3.7	23.96	3.94	6.08	3.84	6.24
4.0	24.65	4.40	5.60	4.23	5.83
4.3	36.39	5.23	6.96	4.75	7.66
4.6	41.54	5.48	7.58	5.28	7.87
4.9	36.93	5.89	6.27	6.06	6.09
5.2	41.66	6.27	6.64	6.20	6.72
5.5	44.15	6.81	6.48	6.58	6.71
5.8	39.78	7.46	5.33	7.01	5.67
6.0	52.31	7.93	6.60	4.88	10.72
10	69.99	14.19	4.93	14.07	4.97
20	140.28	28.38	4.94	28.08	5.00
29	224.42	48.66	4.61	42.98	5.22
38	275.07	57.64	4.77	56.96	4.83
47	324.28	69.92	4.64	86.51	3.75
57	419.33	88.36	4.75	113.08	3.71
66	455.48	104.58	4.36	213.35	2.13
67	443.39	99.46	4.46	166.43	2.66

Table 5.5: Empirical timings and improvements of the simulation section statistical analysis

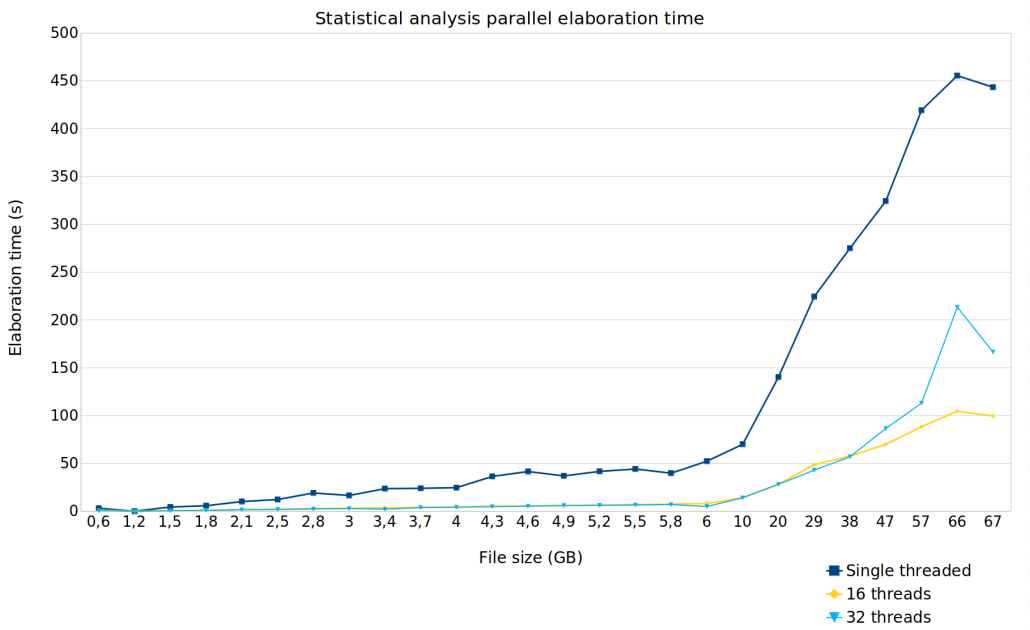

Figure 5.9: Graph statistical analysis improvements parallel version only

5.7.4 Considerations about the results

While this analysis does not provide the complete view about the simulation, its duration makes it more frequently usable. While the speed-up of the whole program can never reach the number of threads involved, it receives significant advantages by using the multithreaded approach and the RAM cost is much cheaper than the full analysis.

Memory requirements are also decreased using the statistical analysis, depending only on the number of signals loaded, which is supposed to be much lower in number than the number of changes to all the signals. On average, the parallel paradigm improves by 5.6 times the duration of the whole computation if we only consider the changes analysis using 16 threads; a little less using 32 threads.

In this case, the whole span of the algorithm is used by the thread parallel sequence and a small synchronization part, which can lead the CPU to idle for more time than necessary. It could be possible to implement a busy waiting system after this part in order to reduce system calls.

5.8 Conclusions

While it is possible to gain significant advantages by using the multithreaded approach, it is difficult to get an advantage proportional only to the number of threads. While semaphore system calls are inherently slow, context switching, due to the discovery of CPU security issues, became even slower, making it more difficult to rely on cache. A more comprehensive study for the limited performance gain is show in the article [4], showing through the speed-up stack model how non-idealities affect performance. In particular, the main issues with this implementation are the spinning on the synchronization barrier ensuring that the new file part is loaded and possibly some hardware costs, since the CPU is dated for today standards.

Future improvements may come by using the latest features of C++20, coroutines, that are much lighter than threads. During the analysis, since synchronization primitives are used only to form a barrier, it could be more useful using communication through coroutines in order to achieve parallelism with less effort from the point of view of the CPU.

Another future work might be thinking about a parallel-only approach, using the GPU in order to do the parsing, using frequent synchronization between CPU and GPU and frequent memory copy on a server where there is at least one available. This would not be advisable in the current state, where results need to be put in order in the final structure.

It is in development a version on a wait-free data structure, having as many file iterators as twice the number of threads (due to the double buffering technique). In conjunction to this, the Signal class has been modified in order to be a wait-free structure. Wait-free data structures have the property of allowing all of the threads to run concurrently [15]. The first results are promising, while the current issue with this approach is the memory overhead produced. The current work and possibly the future work will be reported in [2].

Bibliography

[1] Luciano Bonaria et al. *Test-Plan Optimization for Flying-Probes In-Circuit-Testers*. 2019 ITC-Asia. 2019.

[2] Andrea Calabrese et al. *Accelerated Analysis of Simulation Dumps through Parallelization on Multicore Architectures*. ITC 2020. 2020.

[3] Clyde Coombs and Happy Holden. *Coombs' Printed Circuits Handbook*. Ed. by McGraw-Hill Education. 2016.

[4] S. Eyerman, K. Du Bois, and L. Eeckhout. "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications". In: *2012 IEEE International Symposium on Performance Analysis of Systems Software*. 2012, pp. 145–155.

[5] Y. Hiratsuka et al. "A design method for minimum cost path of flying probe in-circuit testers". In: *Proceedings of SICE Annual Conference 2010*. 2010, pp. 2933–2936.

[6] K. Hird, K. P. Parker, and B. Follis. "Test coverage: what does it mean when a board test passes?" In: *Proceedings. International Test Conference*. 2002, pp. 1066–1074.

[7] "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590.

[8] K. Marcinek and W. A. Pleskacz. "AGATE - towards designing a low-power chip multithreading processor for mobile software defined radio systems". In: *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2012, pp. 26–29.

[9] Michael McCool, Arch D. Robison, and James Reinders. *Structured parallel programming: Patterns for Efficient Computation*. Ed. by Morgan Kaufmann. 2012.

[10] Microsoft. *Parallel Patterns Library*. 2016. URL: https://docs.microsoft.com/it-it/cpp/parallel/concrt/parallel-patterns-library-ppl?view=vs-2019.

[11] NVIDIA. *CUDA toolkit documentation v11.1.0*. URL: https://docs.nvidia.com/cuda/.

[12] NVIDIA. *CUDA zone*. URL: https://developer.nvidia.com/cuda-zone.

[13] K. Tsai, R. Guo, and W. Cheng. "A Robust Automated Scan Pattern Mismatch Debugger". In: *2008 17th Asian Test Symposium*. 2008, pp. 309–314.

[14] H. van Schaaijk, M. Spierings, and E. J. Marinissen. "Automatic generation of in-circuit tests for board assembly defects". In: *2018 IEEE 23rd European Test Symposium (ETS)*. 2018, pp. 1–2.

[15] Anthony Williams. *C++ concurrency in action - Practical Multithreading*. Ed. by Manning. 2012.