Universidad Zaragoza

Master's Final Project

Characterization and Performance Enhancement for ORB-SLAM: A Study of Multi-Processing Parallelization Techniques

Author

Jorge Ferrer Beired

Advisors

Rubén Gran Tejero

Alejandro Valero Bresó

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2023

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Jorge Ferrer Beired,

con nº de DNI 73213391K en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) en Robótica, Gráficos y Visión por Computador, (Título del Trabajo) Characterization and perfromance enhancement for ORBSLAM: A study of multi-processing parallelization techniques.

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 5 de diciembre de 2023

Fdo: Jorge Ferrer Beired

Agradecimientos

A mi familia por su amor incondicional y su apoyo constante en cada paso de mi vida.

A María por estar siempre ahí para mí, y apoyarme en todo lo que hago.

A mis tutores Rubén y Alejandro y mi tutor extra-oficial Darío por acompañarme en este viaje en el que me han prestado sus grandes conocimientos y su casi infinita disponibilidad en los momentos necesarios. Este trabajo y todos los conocimientos que he conseguido haciéndolo son gracias a su entrega y dedicación.

Contents

Chapter 1

Introduction

This chapter introduces the reader to the context of this work, highlighting the importance of parallel computing and computing vision applications, the scope and main goals of this work, and a general structure of the remainder of the document.

1.1 Motivation

Parallel computing has become a critical component of the computing technology of this century. Multiprocessors represent the high-performance end of almost every segment of the computing market, from supercomputer or datacenters to domestic-oriented products like personal laptops or smartphones. Even lately, Arduino-like microcontrollers are joining this trend like the ESP-32 SOC [1, 2]. This transformation has propelled the adoption of parallel computing techniques across a myriad of domains, including computing vision.

Simultaneous Localization And Mapping (SLAM) is a family of algorithms that consists in the concurrent construction of a model of the environment (a map), and the estimation of the state of a robot moving within it. In particular, Visual-SLAM cameras capture images of the environment, providing a rich source of data that can be used to identify and track features and understand the environment's layout for a fairly low-cost sensor. SLAM is a much harder problem than mapping or localizing on isolation, but from an academic point of view, Visual-SLAM has mostly been technically solved in the last decade [3]. Current efforts in the field move toward improving the long-term robustness on real-world application or high-level understanding like the use of semantic segmentation [4, 5].

One of the technical breakthroughs of the last decade came with ORB-SLAM, first released in 2010 [6] and being iterated upon until its most recent official version ORB-SLAM3 in 2020 [7]. It manages to operate robustly on real-time running on conventional hardware, and thanks to being open source, made SLAM more accessible

to a wider range of developers that have produced their own versions and modifications of the algorithm [8, 9, 10, 11, 12].

Parallelization stands as a compelling solution to address the computational bottlenecks encountered in SLAM applications. By distributing the computational load across multiple processing cores, parallel SLAM algorithms can significantly reduce processing time and extract the maximum performance possible for a given hardware. This enhanced performance paves the way for more sophisticated and autonomous robotic systems capable of operating in challenging environments.

Some attemps of performance enhancement on ORB-SLAM3. On the software side, some examples include defining pragma directives [13] or by changing filtering strategies to try to reduce the workload [14]. Some exploration has already been done by designing a hardware accelerator specially designed to compute ORB features[15]. But to the best of our knowledge, there are no references on scientific literature of an exploration that manages the parallel scalability levels that we reach in this theses.

1.2 Goals

In the present thesis, we evaluate the potential for parallelization of ORB-SLAM3 [7]. We thoughtfully study the application for the identification of bottlenecks and implement and evaluate the appropriate parallelization techniques, and measuring their impact on performance.

As such, the main objective is to provide an example of parallel algorithms applied on already existing programs and a working workflow of how such evaluations (from fitness identification to implementation and measurement) should be. Of course, we expect as a secondary objective that our work improves the performance of the study application in a significant way.

The tasks proposed to achieve such goals are:

- Develop tools and methods to characterize ORB-SLAM3 both on its original version and in our modifications.
- Identify and prioritize performance bottlenecks on the application.
- Implement working proposals based on exploiting parallelism when possible to fix those bottlenecks.
- Quantitatively evaluate the effectiveness of the proposed changes by using the tools developed on the first task.

1.3 Structure of the Document

The rest of this work is organized as follows. Chapter 2 describes the background of this work. On it, we provide an introduction to parallelism from a historical point of view, as well as a review of one of its most important laws: Amdahl's law. We, then proceed to describe the main tools that were used in the development of this work and the metrics used. Chapter 3 presents The ORB-SLAM3 system. Also introduced by a historical approach to its development, it is followed by a minimal description of its systems. Then we particularize the metrics to the algorithm, describing in detail its meanings and implications as well as providing a modified base version of the algorithm for testing. Finally, we describe our proposed improvement of the application based on exploiting the multi-core capabilities of a CPU. Finally, Chapter 4 provides the results both of the reference version and the parallelized proposal while discussing the outputs and justifying the decisions taken based on the data obtained. Finally, Chapter 5 closes up this thesis by providing the final conclusions.

In practice, the parallelization of a program is achieved by implementing threads. Threads enable the simultaneous execution of multiple tasks. These threads can be executed concurrently (during the same time span, not necessarily in parallel), allowing the processor to switch context between different threads rapidly, giving the illusion of simultaneous execution. For example, one thread could be executing an e-mail program, another a web browser, and another playing music in the background. We take this for granted, but not that long ago, this was not the case.

Chapter 2

Background

This chapter presents a brief introduction to CPU architecture and its limitations, including Amdahl's Law as a view of the limitations on performance improvements, followed by a description of the tools that enable to identify performance bottlenecks in ORB-SLAM and parallelize the application. Finally, different metrics of interest and a description of the timers used in this work are introduced.

2.1 An Historical Introduction to CPU Architecture

The following historical approach is based on the great introduction provided by Culler and Singh[16].

In the past, computer vendors improved their CPUs fueled by advances in material science, manufacturing processes, the understanding of semiconductor physics, and processor architecture. This was the time when Moore's Law stood as the driving force of microprocessor development in the mid-20th century. Moore's Law predicts that the number of transistors on a microchip will double approximately every two years, leading to a subsequent increase in computational power while also reducing the cost per transistor. For decades, this law held as a guiding principle and became a self-fulfilling prophecy as the industry rallied around it, driving research and development to meet the projected goals.

However, as we moved into the 21st century, the challenges of sustaining this exponential growth in transistor count became more pronounced. The traditional Dennard Scaling, achieved through shrinking transistor dimensions and reducing feature sizes, encountered physical and economic limitations. It became evident that the "easy" path of continually shrinking transistors was reaching its limits. The so-called "Dennard Scaling", where smaller transistors would inherently consume less power and perform better, started to break down. As transistors became smaller, they began to exhibit leakage currents and power dissipation challenges that could not be easily mitigated.

This marked a pivotal moment in the evolution of CPU architecture. Engineers and chip designers could no longer solely rely on the transistor count doubling every two years to provide performance improvements. Instead, they needed to explore alternative strategies to continue enhancing system performance while managing power consumption and heat dissipation. Not only it became technically harder to provide more powerful processors, but from an energetic point of view, two parallel processors at a given frequency consume less energy than a single processor at double the frequency. Hence, the industry shifted its focus towards parallelism and specialization. CPUs began to integrate multiple cores, allowing for parallel execution of tasks. This shift represented a move from relying solely on single-threaded performance gains to leveraging the power of multiple cores working in tandem.

This approach, however, came with its own set of challenges, including the need for efficient thread scheduling, synchronization, and the development of software that could effectively exploit parallel architectures. This last claim is very important since a sequential algorithm might not be optimal for a parallel architecture and vice versa.

We briefly describe several paradigms of parallelism such as Data/Task/Pipeline parallelism in Annex A. In this work, we will explore how to apply a parallel pipeline, as well as data parallelism in the form of `parallel fors`.

2.2 Amdahl's Law

Proposed by Gene Amdahl in 1967, Amdahl's Law offers a quantitative insight into the limits of performance improvement when enhancing only a portion of a computation with parallel processing, while keeping the remainder sequential [17]. Although it was initially targeted to parallel architectures, it is applicable to any enhancement that improves the performance of a computer system. It is also applicable to other metrics such as power or reliability, and even many human activities.

As can be seen in Figure 2.1, if only a region of the code F is improved, the maximum possible speedup is limited by the fraction of the code that cannot be speeded up. In the limit where the execution time of the enhanced region is reduced down to 0, the entire application will still take $(1 - F)$ time.

We can see the same concept expressed in mathematical form in Eqs. 2.1 and 2.2 where T_o and T_i are the execution times respectively of the original and the improved program.

$$T_i = T_o \cdot [(1 - F) + F/S] \tag{2.1}$$

Hence, the speedup gained by the enhancement is given by Equation 2.2.

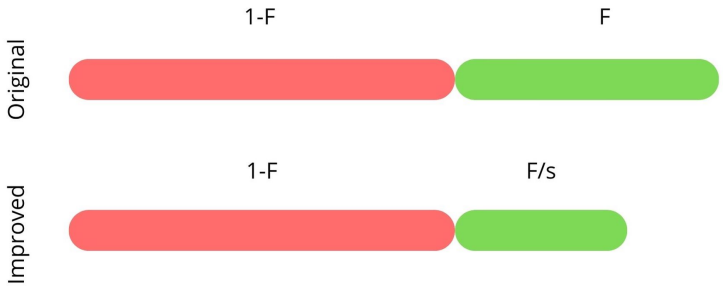


Figure 2.1: A program originally takes some time to execute. After improving a region F by a speedup factor of S the execution time is improved, but the $1 - F$ region latency remains unchanged.

$$\text{Speedup} = \frac{T_o}{T_i} = \frac{1}{(1 - F) + F/S} \tag{2.2}$$

Amdahl's Law highlights the importance of identifying and optimizing the most time-consuming and largest fraction of the execution time. It is not useful to try to accelerate functions that take a small fraction of time. Fortunately, it usually turns out that the most time-consuming functions are the simple ones.

Finally, it does not pay off to throw unlimited resources to improve the speedup for that function. The Law of Diminishing Returns comes into play here, offering a perspective on the constraints imposed by Amdahl's Law. According to the Law of Diminishing Returns, as additional resources are allocated to improve a specific aspect of a system while keeping other factors constant, there comes a point beyond which the incremental gains diminish and may even turn negative.

In the context of Amdahl's Law, the idea is that dedicating excessive resources to enhancing a specific portion of a computation might not yield the desired improvements in overall performance. While initially, a small enhancement can lead to significant speedup gains, there reaches a threshold where the diminishing returns become pronounced. Beyond this threshold, devoting more resources could result in higher costs, increased power consumption, and even potential system complexities that outweigh the benefits gained from the enhanced parallelism.

Of course, the time it takes the programmer to implement the improvement is an important factor. In layman's terms, the low-hanging fruits that give small improvements will improve significantly the overall execution. However, the more convoluted and time-consuming to implement enhancements will generally give smaller improvements.

For example, consider a scenario for $F = 0.5$. The maximum theoretical speedup is 2. For an improvement factor of $P = 2$, the speedup is $x1.33$ which shows that there is

still a potential increment of 0.66 to the speedup. But after that, the marginal speedup gain drops rapidly. We must increase S to 5 to gain another 33% for a speedup of $x1.66$. As P increases, it becomes less and less rewarding to add resources to improve the speedup further, and the potential gains left to reap decrease as well.

Now consider the case where these improvements or enhancements come from parallel designs. Equation 2.2 takes a new meaning. The improvement factor P is the number of processors in a multiprocessor or the number of cores in a CPU. F, then, becomes the fraction of the code that can be parallelized by these cores.

Again, the ideal speedup is given by the number of cores assigned to the work. This happens when the full code can be parallelized, but even if as much as 99% of it can be fully parallelized, that sequential 1% causes that the maximum speedup cannot increase over 100 even if infinite cores are deployed and the execution time of the parallelized section tends to zero.

2.3 Tools

The analysis and implementation carried out in this work were aided by the usage of third-party tools that help us to focus on the problem itself rather than developing everything from scratch. TBB helps us to implement the parallel algorithms while PERF is used to obtain performance metrics of the programs.

2.3.1 Threading Building Blocks (TBB)

The advent of modern programming languages has largely absolved programmers from concerns like memory alignment and register assignments. These aspects are typically efficiently abstracted by compilers, allowing programmers to focus on the higher-level logic of their applications. In a similar vein, Intel Threading Building Blocks (TBB) aims to abstract away thread management complexities, enabling programmers to concentrate on identifying and exploiting parallelism at a more conceptual level rather than delving into the intricacies of low-level thread management[18].

TBB introduces the concept of tasks as units of work that can be executed concurrently. Programmers can create task-based parallelism by defining tasks and their dependencies, while TBB handles the underlying threading details to optimize execution on multi-core processors.

A key strength of TBB lies in its task scheduler, which dynamically assigns tasks to available threads based on system resources and workload characteristics. This dynamic load balancing ensures that tasks are efficiently distributed across processing cores, maximizing parallelism and overall system performance.

TBB provides a comprehensive collection of parallel algorithms and data structures, facilitating the parallelization of common programming patterns such as parallel loops, reductions, and pipelines. These components are designed to reduce the complexity of writing parallel code while maintaining scalability and performance.

Under this paradigm, the programmer's role is to identify opportunities for parallel execution within his/her code. This involves analysing the algorithm and determining which portions can be executed concurrently without introducing conflicts or dependencies or modifying the algorithm to avoid those conflicts entirely.

Once the programmer has exposed parallel opportunities, TBB takes over the responsibility of managing and executing these tasks efficiently at run time but does not guarantee that the tasks will be executed concurrently. TBB's task scheduler dynamically assigns tasks to available threads, ensuring optimal utilization of processing cores. It employs various scheduling techniques, such as work stealing and task priorities, to adapt to changing workload conditions.

To effectively execute tasks, TBB utilizes a thread pool, that is, a collection of pre-created threads ready to execute tasks. When a task becomes available, the task scheduler selects an idle thread from the pool, assigns the task to it, and wakes the thread up. The thread then executes the assigned task until completion, before returning to the pool for further assignments.

2.3.2 PERF

PERF, an abbreviation for "Performance Events", stands as a crucial subsystem within the Linux kernel, dedicated to monitoring and analysing performance-related events on a system. It equips users with a robust set of tools designed for profiling and optimizing both application and system performance.

PERF leverages hardware performance monitoring units (PMUs) present in modern processors. PMUs are capable of tracking diverse hardware events, spanning CPU cycles, cache misses, branch instructions, and so on. By tapping into these hardware counters, PERF enables users to extract detailed performance data, offering insights into the intricacies of their applications and the overall system.

PERF encompasses a range of functionalities and commands that provide detailed insights into system and application performance. There are plenty of commands in the perf's toolset for very diverse applications and measurements. The two main commands that we used through the project are `perf stat` and `perf record/report` to obtain CPU statistics and timed profiling, respectively.

CPU Statistics

Perf-stats provides a summary of various performance-related statistics for a specified command or process. The basic usage of `perf stat` is as follows:

perf stats [options] **command** [arguments]

Perf-stats runs a specified command (or the whole system if no command is provided) and collects performance statistics during its execution. It provides a summary of interesting insights about CPU microarchitectural metrics, including not only hardware events but also software events like system calls or tracepoint events. It is highly customizable, but a simple usage example looks like this:

```
#perf stats du -sh /
 Performance counter stats for 'du -sh /':

          8.628,09 msec task-clock                #    0,596 CPUs utilized
            32.250      context-switches          #    0,004 M/sec
                19      cpu-migrations            #    0,002 K/sec
             5.151      page-faults               #    0,597 K/sec
    28.230.092.072      cycles                    #    3,272 GHz                      (83,20%)
     1.252.822.287      stalled-cycles-frontend   #    4,44% frontend cycles idle     (83,48%)
     9.341.617.792      stalled-cycles-backend    #   33,09% backend cycles idle      (83,49%)
    29.560.352.162      instructions              #    1,05  insn per cycle
                                                  #    0,32  stalled cycles per insn  (83,42%)
     5.466.402.042      branches                  #  633,559 M/sec                    (82,96%)
       141.436.920      branch-misses             #    2,59% of all branches          (83,45%)

      14,472571331 seconds time elapsed

       0,899093000 seconds user
       7,752093000 seconds sys
```

Timed Profiling

One of the main uses of PERF applied on this work is the sampling mode for CPU-cycle events. When the counter of the PMU cycles reaches a certain value, PERF registers an event. This event contains information about the application that triggered the event. The application can be instructed to sample at a given frequency, and PERF will dynamically adjust the said value to maintain a constant sampling frequency. While this approach incurs some information loss due to the sampling nature of the method, it usually provides a sufficiently clear performance snapshot. There is an overhead that increases with a higher number of registered events, but keeping reasonably bounded parameters, it does not significantly affect the application load balance.

A key technique employed by PERF is profiling applications by sampling the frame pointers. This method facilitates the capture of stack traces at regular intervals, revealing the most frequently executed functions and their call stacks. This insight unveils performance bottlenecks and hotspots within the application, aiding in targeted optimization efforts. Notably, this low-intrusive approach eliminates the need for instrumenting the code, making it a relatively low-effort method to gain valuable performance metrics.

`Perf-record` can profile CPU usage based on sampling the instruction pointer or stack trace at a fixed interval (timed profiling). It can be configured to sample any

event (like cache misses or any other low-level processor events), but the default refers to CPU cycles. It generates a binary file with sampled events information. Basically, every event contains its type, its instantaneous period and the stack trace of where the instruction pointer was when PERF was awaken. The complete lists of events can be obtained by using `perf-script`, which converts the output file to a plain text file. But more commonly it is analysed by using `perf report`. This command generates an informative report of which symbols had more events associated with them. It is shown in a tree-like structure, where indented symbols are children of the upper layers.

For example, to sample CPU stacks at 99 Hertz (-F 99), for the entire system (-a, for all CPUs) in idle state, with stack traces (-g, for call graphs), for 10 seconds. We need to call `perf-record`, which creates the perf binary file.

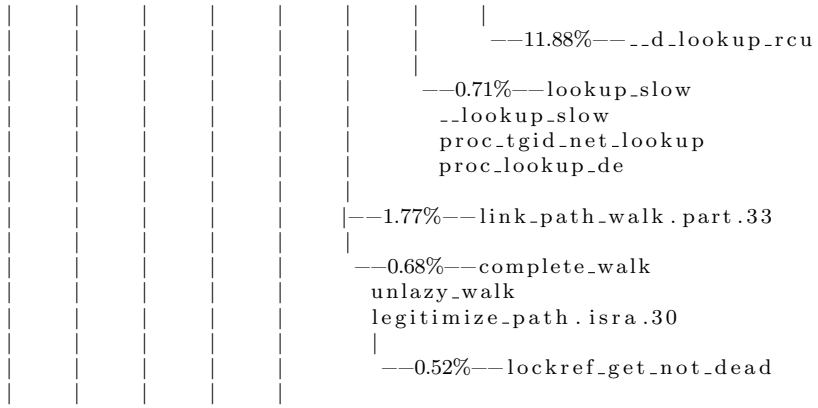
```
# perf record -F 99 -g -- du -sh /
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0,108 MB perf.data (645 samples) ]
```

By using `perf script` we can analyze individual events. For example, here can be seen one of such events:

```
du 102485 10065713.250878:   37899658 cycles:
    ffffffffbb57f0e2 proc_fill_cache+0x72 ([kernel.kallsyms])
    ffffffffbb57f4ef proc_map_files_readdir+0x2ff ([kernel.kallsyms])
    ffffffffbb4fb43a iterate_dir+0x9a ([kernel.kallsyms])
    ffffffffbb4fc3db __x64_sys_getdents+0xab ([kernel.kallsyms])
    ffffffffbb205207 do_syscall_64+0x57 ([kernel.kallsyms])
    ffffffffbbe000a4 entry_SYSCALL_64_after_hwframe+0x5c ([kernel.kallsyms])
        7f23ae744d58 __getdents64+0x18 (/lib/x86_64-linux-gnu/libc-2.27.so)
```

The output of calling `perf report` shows an interactive tree-like structure of the call stack and the frequency of each function. This default representation is such that the upper-level function is the parent of the nested functions. The percentages represent the time spent on that function or any of its children.

```
# Total Lost Samples: 0
# Samples: 645  of event 'cycles'
# Event count (approx.): 23296534249
# Children   Self  Command  Shared Object      Symbol
# ........  ....  .......  .............  ...................................
#
    76.74%  0.32%  du [kernel.kallsyms]  [k] entry_SYSCALL_64_after_hwframe
   |
    --76.42%--entry_SYSCALL_64_after_hwframe
        |
        --76.10%--do_syscall_64
            |
            |--39.40%--__x64_sys_newfstatat
            |     __do_sys_newfstatat
            |     |
            |     |--36.07%--vfs_statx
            |     |     |
            |     |     |--24.98%--user_path_at_empty
            |     |     |     |
            |     |     |     |--17.17%--filename_lookup
            |     |     |     |     |
            |     |     |     |     |--16.36%--path_lookupat
            |     |     |     |     |     |
            |     |     |     |     |     |--13.09%--walk_component
            |     |     |     |     |     |     |
            |     |     |     |     |     |     |--12.38%--lookup_fast
```



It is important to have a lookup for the symbols' names. Otherwise, instead of named functions, all that we get is gibberish hexadecimal numbers representing the memory addresses of such functions. For system calls or other kernel-related symbols, the kernel must be compiled with the appropriate flag activated or with a debug-info package. Similarly, user code lines should be compiled with frame pointers. This is done by passing the appropriate flag to the compiler, as described in Section 4.1. By default, most compilers include omitting frame pointers as a small optimization. There are other stack walking techniques (like dwarf or Last Branch Record), but returning the frame pointers (i.e., the translation from memory address to function name symbol) is the more convenient because the other methods are very slow to process or have limited stack depth and should only be used when using frame pointers is a no-go.

The ORB-SLAM application under study has very fast and unpredictable "program phases" (i.e., frames), meaning that a very fine-grained approach is sometimes needed. For some analysis is useful to think about the performance on a per-frame basis. The time window of a frame is about 20-50 ms and the number of perf-events in that time for a reasonably small sampling frequency is in the order of only a few dozens. This small number is not sufficient to study the application with such a low granularity. Consequently, the analysis is restricted to a broader view of the application as a whole. For this reason, most of the metrics are given with statically instrumented timers set at a surgical location. PERF is more useful when used as a surveying tool and once relevant functions are identified, instrument the code with more detailed timers as described in the next section.

2.4 Metrics and Timers

Performance is usually the most important criterion when comparing systems or different versions of an application. Let t_0 and t_1 the execution time of two versions of an application, where t_1 corresponds to an optimized version ($t_1 < t_0$) and the speedup is

given by $S = \frac{t_0}{t_1}$.

To measure time, several timers have been used to instrument the code statically. Time metrics are measured by using the monotonic clock `std::steady_clock`. This clock is not related to wall clock or CPU ticks. Hence is the most suitable for measuring intervals because the time between ticks is constant. Several timestamps are taken at different points of interest throughout the program.

As we will see in Chapter 3, ORB-SLAM3 has several operating modes. For the purpose of this work, we will only examine the non-inertial stereo sensor because it offers a clear program path that is easier to follow in the source code while still having very robust trajectory predictions.

Chapter 3

ORB-SLAM3: Description and proposed modifications

This chapter provides an introduction to ORB-SLAM3 from two angles. First, a brief historical introduction of the three official versions published is described. Then, a brief overview of the ORB-SLAM3 systems is presented. Finally, several parallel optimizations, whose implementation will be evaluated in Chapter 4, are proposed.

3.1 Development of ORB-SLAM

ORB-SLAM is a state-of-the-art visual simultaneous localization and mapping (SLAM) algorithm that combines feature-based tracking, mapping, and loop closure detection to achieve robust and accurate real-time camera pose estimation and 3D reconstruction.

It has undergone an evolutionary development process, resulting in several distinct versions: ORB-SLAM [6], ORB-SLAM2 [19], and finally ORB-SLAM3 [7]. Each version represents a significant advancement over its predecessor, expanding its capabilities and addressing limitations.

The original ORB-SLAM version was designed for monocular cameras, utilizing the ORB (Oriented FAST and rotated BRIEF) feature detector and descriptor. It provided real-time camera tracking and mapping by leveraging both local and global features. Building upon this foundation, ORB-SLAM2 introduced support for stereo and RGB-D cameras, along with loop closing and relocalization capabilities. Loop closing aimed to detect and close loops in the trajectory, enhancing map consistency, while relocalization enabled recovery from tracking failures. ORB-SLAM2 incorporated a bag-of-words [20] approach for place recognition and a co-visibility graph for map optimization.

The most recent iteration, ORB-SLAM3, further improved upon its predecessors. Among other inclusions, it added intermediate advances that integrated IMU (Inertial Motion Unit) [21, 22] support for monocular images and extended it to all sensor modes.

It was also the first complete multi-map SLAM [23] system able to handle visual and visual-inertial systems, in monocular and stereo configurations.

ORB-SLAM3 can also represent a set of disconnected maps in a structure called `Atlas`. When the system got lost in previous versions, it remained in that state until a loop closing was detected with the map or it had to restart the execution entirely losing the previous map information. With multi-mapping, this situation is handled simply by starting a new map and storing the old one in the `Atlas`. If a loop close situation is identified with an archived map of the `Atlas`, the active map can be seamlessly merged with it.

3.2 Brief Review of ORB-SLAM3 System

ORB-SLAM is a complex program. The reader can refer to the original papers [6, 19, 7] for a detailed description. However, since the information is dispersed among the papers, we have condensed our own version of the explanation in Appendix B. Next, we present what we consider the minimal knowledge in order to understand the concepts presented in this work. We focus only on the non-inertial stereo version of the algorithm since that is how we run all the experiments.

3.2.1 What Does Matching Mean?

While a full image contains a vast amount of data, not all of it is necessary for certain tasks. Usually, focusing on a few hundred significant points is enough to identify and describe the most important features in the image, reducing the computational complexity and improving the overall efficiency of the task. This is what is known as sparse methods, in contrast of dense methods, where all (or most) of the pixels of the images are used.

ORB (Oriented FAST and Rotated BRIEF)[24] is a feature detector and a 256-bit binary descriptor used in computer vision to identify and describe `KeyPoints` in images. It is based on two existing techniques: FAST and BRIEF (Binary Robust Independent Elementary Features).

FAST[25] (Features from Accelerated Segment Test) detector is a `KeyPoint` detector that efficiently identifies KeyPoints in images. It works by identifying pixels that are significantly brighter or darker than their surrounding. These pixels are likely to be corners or edges, which are often good locations for `KeyPoints`.

ORB features are descriptors that describe the local appearance of `KeyPoints`. They are based on BRIEF, which is a binary descriptor that is very efficient to compute. ORB extends BRIEF by adding orientation information to the descriptor. This makes

ORB features more robust to rotation than BRIEF descriptors.

In summary, FAST interest points provide the locations for ORB descriptors to be extracted from. The ORB descriptor is then evaluated to describe the local appearance of the `KeyPoint`.

Having two images with their ORB features identified, we can compare them by taking a distance metric (usually Hamming distance is used) and, if they are below a threshold and satisfy several geometric requirements, we can say that those points are the same. In practice, not all pairs of KeyPoints are brute-forcedly tested but sensible guesses are used based on computer vision techniques (such as epipolar or homography calculations).

3.2.2 ORB-SLAM3 Systems

Figure 3.1 shows a high-level abstraction of ORB-SLAM3 consisting of three main tasks that run concurrently on three threads:

- Tracking: this is the main thread of the execution. Two main sub-tasks can be differentiated:
  - 1) Process the images: it takes a pair of images and extracts its ORB features. Then performs stereo-matching between them and triangulates a depth prediction of the matched points.
  - 2) Track: it takes the current frame and performs matching with the previous frame's `MapPoints` (that is, points of the last frame that were identified as points of the map). With all of these matches, Bundle adjustment (a non-linear optimization) is performed on the camera pose to minimize the reprojection error. This means that camera pose is updated so the 3D MapPoints of the last frame are projected on the current frame the closest possible to its actual position. Then, the process is repeated with the LocalMap (points in the map that might have not been present on the last frame but are potentially visible from the current Frame). Matching of the current frame is performed with those LocalMapPoints and the pose is refined again with said matches. Note that Track does not create any MapPoint, it only matches KeyPoints of the current Frame to already existing MapPoints. Finally, a decision is taken on whether the current frame is upgraded into a KeyFrame and then included on the map.
- LocalMapping: whenever a KeyFrame is created by the Tracking operation, this thread activates. It maintains a co-visibility graph of all the KeyFrames to ease

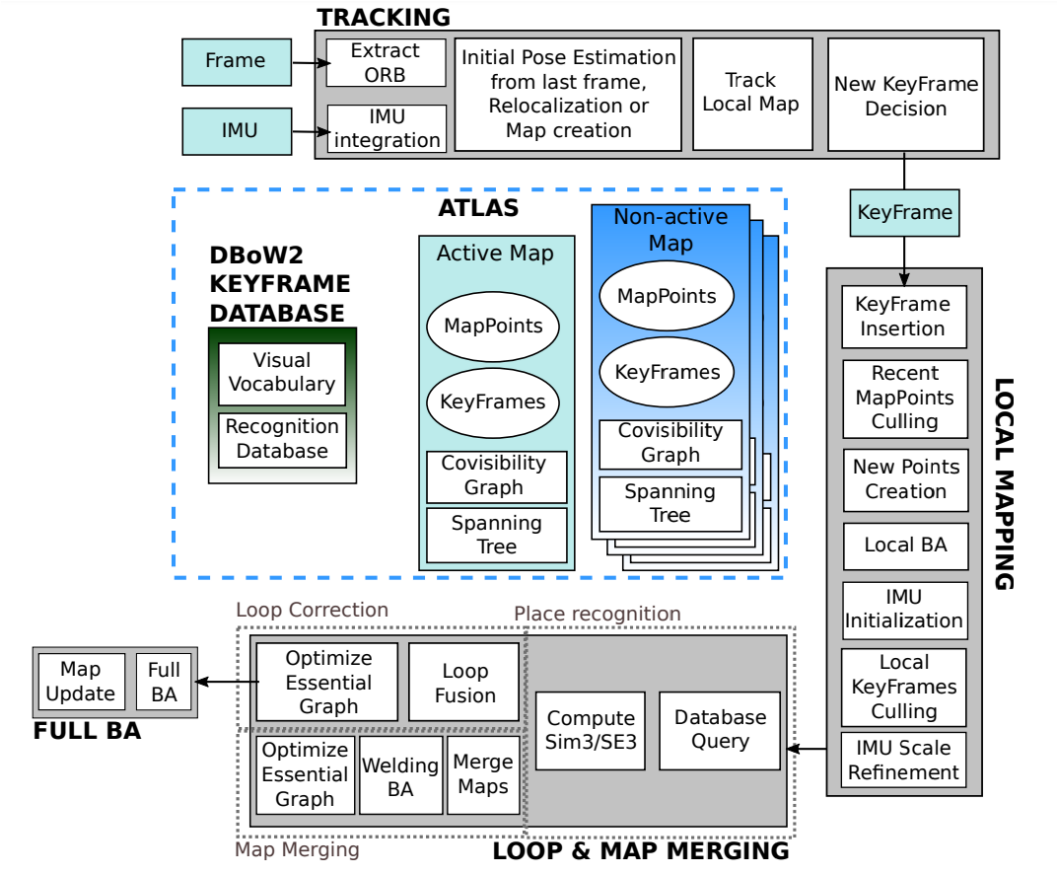


Figure 3.1: Main system components as plotted in the original paper [7]. The three main threads (`Tracking`, `LocalMapping`, and `LoopClosing`) are depicted, as well as the tasks described for the `Tracking` thread.

the identification of the LocalMap for Tracking. It performs MapPoint creation by performing matching among all co-visible KeyFrames. A similar process of pose optimization is performed, but in this case, the 3D position of the MapPoints is included in the optimization, and the target is again to reduce the combined reprojection error on all the KeyFrames in which a given MapPoint is observed. It also computes the culling of repeated or unnecessary MapPoints and KeyFrames. This keeps in check the memory usage by removing redundant information and improving performance.

– LoopClosing: this thread also starts when a KeyFrame is inserted into the map. It looks for similarities of that KeyFrame with other KeyFrames by looking at its BoW [20] similarities. If a match is positive, loop closing or map merging is performed depending on whether the matches KeyFrame is in the active map or not, respectively. The process is similar in both cases, as it involves performing a global bundle adjustment of all KeyFrames and MapPoints of the map(s).

3.3 Performance Metrics: Tasks and Timers

3.3.1 Timers

By using statically defined clocks as described in Section 2.4, timestamps are taken at the start and end of the program, whose difference gives the whole execution time. This includes tasks that are not directly related to the SLAM algorithm, such as loading configuration files and printing the results to the output files after the dataset has been fully processed.

To get a more representative timing metric, an alternative interval is defined as the algorithm execution time. The start point is when the ORB-SLAM system is initialized and the end point is when the processing of the provided dataset ends. When we refer to "execution time" going forward, we will be talking about this metric, because it is more straightforwardly tied to the ORB-SLAM tasks.

Then, more timers are set across the program that measures the time it takes to process different sections of the code. They comprise segments of code that are somewhat semantically related. Each of these intervals is called a task and together covers most of the computation of each frame. They are inspired by the tasks proposed by the original ORB-SLAM developers, but they are not perfectly equivalent, since some of them have been expanded to avoid getting sections of the code untracked. The difference with a timer that measures the latency of the full frame since the image is loaded until the tracking ends is labeled as "Other". These intervals are measured per-frame and its addition is equivalent to the algorithm execution timer described above.

3.3.2 Definition of Tasks

Each pair of timers defines an activity bounded in a tie interval, or"task". Most of the task names are self-explanatory, but others require a small explanation. We have divided the tasks into three groups. This division will become important in Section 3.4. Note that these tasks match with the work described in Section 3.1.

- **LOAD FILE**: this task plainly loads the image files (two per frame) from the filesystem and stores them in an OpenCV image format.
- **PROCESS IMAGE**:
  - Image Rect: this function rectifies the images if needed. For the dataset used in this work (see Section 4.1), they are already rectified, so this corresponds only to a memory copy operation.

- ORB Exc: the images are passed to the ORB extractor and it computes the descriptors for a maximum of 1200 `KeyPoints`. The two images are processed in parallel by the invocation of two `std::threads`.
- Stereo Match: this includes the rest of the processing operations. The most important one is to perform matching between the two stereo images in order to provide depth prediction for those points through triangulation. More work is done inside this task but does not take significant time, such as putting the `KeyPoints` of the images in a cell grid to be easily accessed later.

– **TRACKING**:

- Init Tracking: this operation comprises the locking of some `mutexes` shared with the parallel threads before performing actual computations. This value can be seen as a reference of the synchronization bottleneck, although there are also other synchronization points along the other tasks.
- Pose pred: this task is the first pose prediction refinement by computing matches with the previous frame (Visual Odometry) and then doing pose-only bundle adjustment.
- Track LM: this is the second pose refinement where matches are taken against the `LocalMap`. Although they are not broken down, it has three relevant sub-tasks: 1) building the local map checking all the `KeyFrames` for potential MapPoints, 2) compute the matching against those `LocalMapPoints`, and 3) performing pose refinement by bundle adjustment.
- KF dec: evaluates whether a new KeyFrame is necessary or not. It also includes the cost of creating it if positive. The time spent here is mostly used to build the KeyFrame itself rather than making the decision.

3.3.3 Sampling Frequency and Communication Cost

Remarkably, we include the time of loading the images from the filesystem as a computable time of the Frame processing. We can think of the workflow as two steps: the image acquisition and the algorithm computation. In a real-world example, the images are captured by a camera, and this imposes a lower limit on how fast the application can run, which is given both by the sampling frequency and the communication cost of sending the image to the computing unit. The original ORB-SLAM simulated the sampling wait by forcing the program to wait and match the sampling rate of

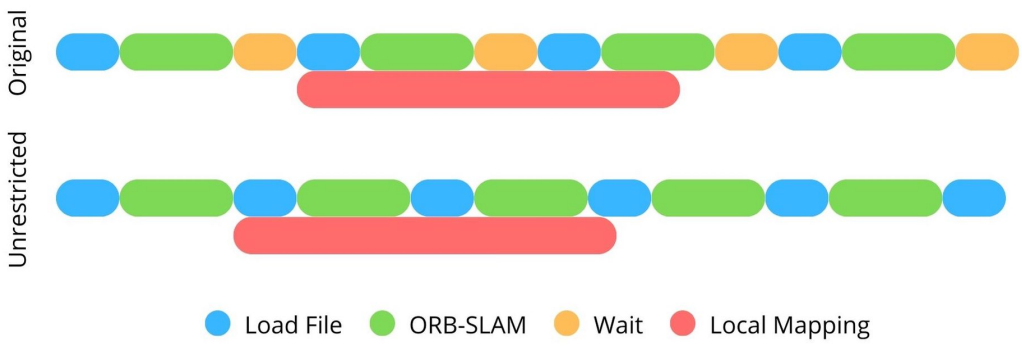


Figure 3.2: Scheme of the program flow on the original version (with waiting between frames to simulate camera sampling rate) and our baseline version where no waiting occurs at all. Moreover, in our versions, the cost of loading the file is included in frame latency computation.

the dataset's camera. On the contrary, in our experiments, we remove the sampling limitation (i.e., an image is read as soon as it is requested by the algorithm) to explore how the system would behave with a high-speed camera that provided images at very fast sampling rates.

In the original implementation, the delay produced by loading the image from the filesystem is not computed as a cost of the `Frame` processing. But we mimic the communication cost with the camera, by including this latency of loading the image as a cost of the processing. Hence, we expect to observe a bigger latency than the original version because of the inclusion of this delay. However, we will see how in the proposed parallelized version we will manage to completely hide this cost.

We can see the two different approaches in Figure 3.2. Our "unrestricted version" has a second-order interaction with the system that might not be obvious. Since the original system waits for the next frame to be available to match the camera sampling rate of the dataset, the parallel threads `LocalMapping` and `LoopClosing` can still perform work while the main Tracking thread is waiting for the camera to "take" the next image.

These threads present some level of synchronization with the Tracking thread since some data structures are shared and mutex-locking has to be used to avoid race condition situations. In our case, as soon as Tracking ends processing one image, a new one is requested. We should, then, expect increased concurrency among the threads, Hence, we should expect some extra amount of synchronization bottleneck because the tracking and the parallel threads are more concurrent, slowing the time of some tasks slightly down. Another effect is that in some cases since the frequency of the Tracking is higher, some frames will have a version of the Map that the `LocalMapping` thread has still not refined but that in the original version, the waiting gave enough leeway to end the

optimization before that frame starts being tracked.

3.4 Proposed Parallelism Enhancement

This section presents the contributions of this work, that is, two TBB-based parallel techniques consisting of pipelining and parallelizing loops. Prior to introducing these techniques, some initial considerations are mentioned.

3.4.1 Initial Considerations

We started developing our work from commit 4452a3c, which was uploaded in December 2021 and is the most recent one as of December 2023. We define a baseline model that we will compare our modifications to. The differences against the official version are a different distribution of the timers, some small bug fixes, the unrestricted tracking frequency as described in Section 3.3.3, and the deactivation of the viewer functionality. In the original version, an additional thread is used to handle a GUI that shows the camera images in real-time as well as the tracking information of the algorithm. We cannot use this thread because the executions are sent from a ssh environment and hence, we remove that work from the code. The performance difference is about 0.5 ms per frame attributed to sending the information to the viewer thread.

ORB-SLAM3 managed real-time execution with very small pose prediction error thanks to implementing novel concepts at the higher level of the algorithm. However, some of the data structures used are not optimal for performance. For example, the way most of the arrays are traversed does not allow for good cache-coherence behaviour. There are also other situations where the system could benefit from an algorithmic restructuration. However, our philosophy when approaching the improvement proposal is to respect the original algorithm as much as possible. We are going to retain everything on the algorithm as-is, and try to extract the maximum parallelism possible with minimal intervention.

As described in Section 3.2, the original version separated the program into three threads for the three independent semantic tasks: `Tracking`, `LocalMapping`, and `LoopClosing`. We are going to respect that separation, but we will divide the Tracking thread into logical segments that we match to three separate high-level tasks. These correspond to the grouping described in Section 3.3.2.

3.4.2 Pipeline

Figure 3.3 summarizes the aforementioned tasks. The three tasks correspond to: loading the file from disk (blue), processing the image (i.e., generating the `Frame` by

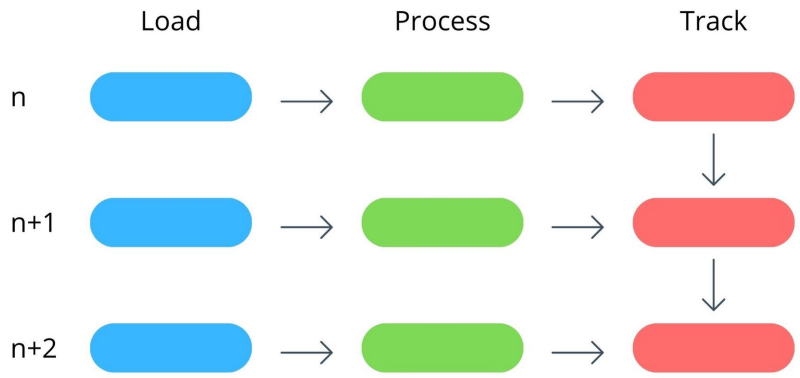


Figure 3.3: Dependencies of the three high-level tasks identified in ORB-SLAM3. The direction of the arrow indicates that the element of the end requires the element of the origin to be completed.

extracting the ORB-features, green), and finally `Tracking` the `Frame` (yellow). The arrows indicating the dependencies enforce the task ordering; i.e., that we need to have loaded and processed an image before tracking it. Moreover, the tracking phase also depends on itself. That is, we can not start to track image $n+1$ until the n-th image has been properly tracked. This is because the process of tracking, iteratively refines the Pose and we can not start the next `Track` until we have a definitive answer from the last one. However, extracting the ORB-features of an image (and most definitely neither loading it from disk) does not depend on the last image ending the tracking phase. We can start loading/extracting an image even though the last one has not yet ended processing. More importantly, the independence between different images allows us to do this in parallel.

The process we are describing fits perfectly on a pipeline model as the one described in Appendix A, where each of the three tasks (load, process, and track) are stages of a parallel pipeline. The only limitations are: 1) the tracking stage must be sequential, and 2) the images must be tracked in order. In other words, `Frame` $n+1$ must wait for `Frame` n to end the tracking, although the first two stages can be parallel. And after processing `Frame` n, we need to track `Frame` $n+1$.

The order requisite might seem obvious, but it is not the default of `tbb::parallel_pipeline`, since the last sequential stage processes the element in the order in which the second stage ended, so if the second stage of Frame $n+2$ was particularly fast to process and ended earlier than $n+1$, it would be taken by the last stage (`Tracking`) before $n+1$. To fix this, we define an additional dummy sequential stage at the beginning that does no work but tells TBB the order of the items.

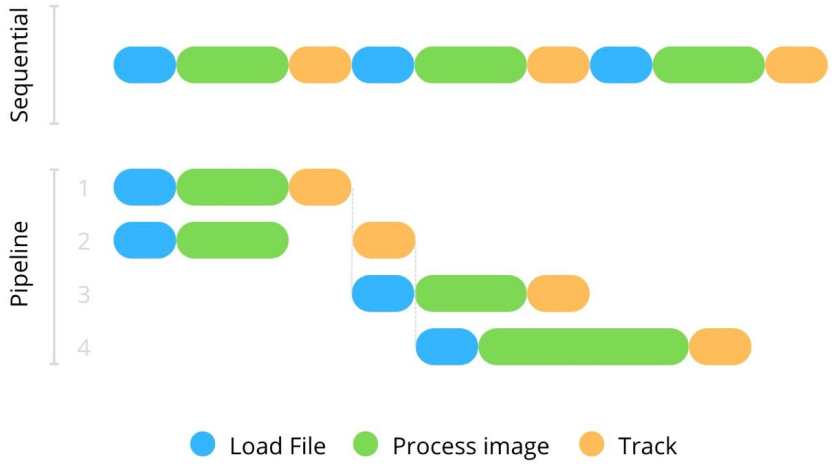


Figure 3.4: Execution example of a sequential program and a pipelined version of two tokens with the three high-level tasks (load file, process image, and track frame) as staged. Numbers in the pipeline version represent items.

Figure 3.4 plots an execution example of a sequential version, in which all stages are performed sequentially, and a pipeline version. In the latter, loading and processing the image is performed in parallel and independently of the `Tracking` stage. If an image has already been processed by the time the n item ends, its `Tracking` stage can be launched immediately, as it happens for the second item of the example. If this is not the case, it still has to wait for the `Frame` to be ready as in items 3 and 4. On the sequential version, it always must wait for the full execution of the ORB extraction.

In the ideal case, the `Tracking` stage always has a `Frame` ready to be processed without waiting, and then, the execution time of the program will be the execution time of the tracking phases.

A key parameter is the number of tokens of the pipeline. That is the maximum number of items that are processed concurrently on it, no matter at which stage they are. In the scheme of Figure 3.4, the number of tokens is 2. Notice that there are never more than two items being processed, even if no stage is being executed for a particular time. Once one item is processed (that is, it ends the last stage), the first stage of a new item is launched.

This means that, in practice, we can define a circular buffer whose size is the number of tokens. The outputs of the intermediate stages are stored there and wait in the queue until the last `Tracking` stage asks for them. The circular buffer allows us to reserve the minimal memory necessary to handle the intermediate products of the pipeline, without wasting resources.

We also introduced a small change in the ORB Extraction. As described in

Section 3.3.2, the two stereo images of a frame are processed in parallel. The baseline version uses two instances of `std::thread`. On the parallel versions of our implementation, to avoid interactions with the TBB routine, we modify that section and use `std::parallel_invoke` instead. Functionally, they should work the same, but by using the thread pool of TBB. However, TBB does not guarantee that the two tasks will be executed in parallel immediately since it might decide to wait until some threads become free. However, by using `std::thread`, the execution always starts, even though the system is oversubscribed, possibly damaging other sections of the code. This type of micromanaging is what we are hiding behind a framework such as TBB, where scheduling patterns are enforced to avoid regrettable situations like the mentioned oversubscription.

3.4.3 Parallel-For

Another way to improve scalability is the use of `tbb::parallel_fors`. This is a TBB structure that allows for the parallelization of loops. The work of the loop is divided into chunks, and each chunk is potentially sent to a worker of the thread pool of TBB. As always, TBB does not guarantee that the execution of all the chunks will, in fact, be concurrent. It might assign all of them to the same thread if there are none free.

In general, the default version of `tbb::parallel_fors`, uses load-balancing algorithms to determine the adequate number of chunks (or reciprocally, the size of the chunks) depending on the dynamic evaluation of the performance. However, the loops that we identify as parallelizable are not very long, so the overhead of this work might be too prohibitive. For this reason, we decided to use a static partitioner, in which the size of chunks is fixed beforehand. This parameter must be chosen carefully; if the chunks are too big, there is less room for parallelization because there are fewer of them. In contrast, if there are too many, it could also damage performance because of the increased overhead of managing them. We will study this parameter to ensure maximum performance.

Several regions could benefit from the parallelization of their loops. We will perform an analysis of the performance and only apply them where they could be more useful. In fact, we will see how the application of more parallelism in certain regions would provide no performance gain at all.

Chapter 4

Experimental Evaluation

This chapter provides a quantitative analysis of the tasks performed by the baseline and the proposed optimizations of ORB-SLAM in terms of execution time. Based on this data, we verify the choices of the proposed parallel version in Section 3.4..

4.1 Machine Environment

Experiments were launched on a machine with Ubuntu 18.04.6 LTS OS built with the kernel Linux 5.4.0-139-generic. The system includes a CPU consisting of an AMD Ryzen Threadripper 1920X 12-core processor with up to 24 threads, hyperthreading capability, 3.31 GHz of operating frequency, and 96 GB of RAM.

The build files were generated with cmake 3.10.2 and compiled using gcc 7.5.0 with compilation flags `-O3 -g -fno-omit-frame-pointer`.

The `-g` flag does not affect performance, but only increases the binary size. The `-fno-omit-frame-pointer` flag could potentially harm the performance, but we have not observed any significant effect in our implementation and helps with debugging.

4.2 Tested Datasets

Datasets are essential for developing and evaluating algorithms. They provide a standardized source of data that can be used as standarized validation. This is important for benchmarking the performance of different algorithms and for ensuring they are reproducible.

The EuRoC MAV Dataset [26] is a publicly available dataset of stereo images and synchronized IMU measurements collected onboard a micro aerial vehicle (MAV). It is widely used for research in visual-inertial odometry (VIO) and simultaneous localization and mapping (SLAM). In fact, it consists of two types of datasets, each with several sequences for a total of 11. An example of the two types of datasets is shown in Fig. 4.1.

(a) Machine Hall (b) Vicon Room

Figure 4.1: Snapshots of the two types of environment present on the EuRoC dataset.

The dataset on the left side of the figure (Machine Hall or MH) is recorded in a realistic industrial scenario. The environment was unstructured and cluttered, which renders the dataset challenging to process. It is composed of five sequences of increasing difficulty in terms of flight dynamics and lighting conditions.

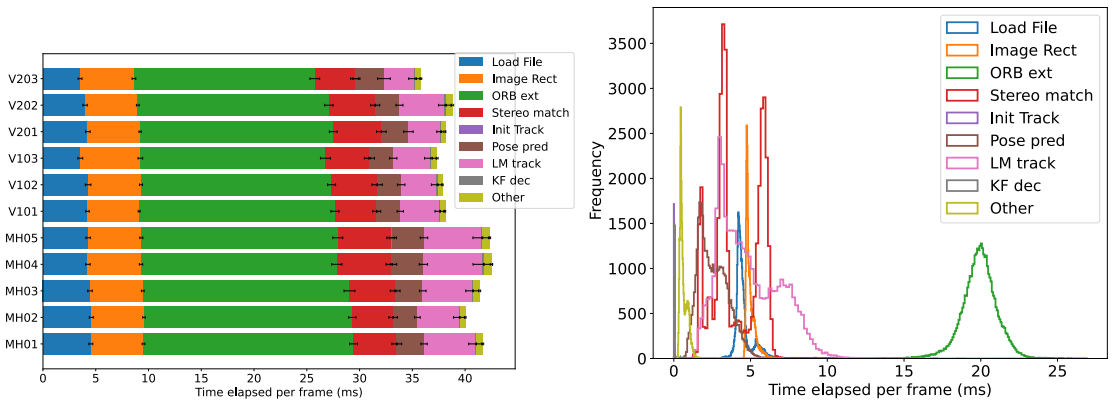
The dataset on the right side (Vicon Room or V) is recorded on a small room (8 m × 8.48 m × 4 m) cluttered with objects of different nature both to render the reconstruction more challenging and to provide more visual texture. As an additional challenge, there are some moving curtains visible in some parts of the dataset. Two different scenarios of the room were used prepared ViconRoom1 and ViconRoom2 datasets, each with 3 sequences.

For ground truth, two different systems were used. On the Machine Hall datasets, laser tracker measured the position of a prism on top of the MAV. For the ViconRoom datasets, a Vicon motion capture system provided 6D pose measurements of the drone. The pose measurements were recorded, respectively, at a rate of 20 and 100 Hz.

4.3 Baseline Model Characterization

This section characterizes the behavior of the baseline version. The key idea of this study is to determine which of the tasks defined in Section 3.3.2 contribute more time to the overall execution, to assest the maximum potential parallelism, and whether the datasets significantly affect the performance.

Figure 4.2a plots the average task breakdown for each dataset. Each colour represents a task, and they are stacked such that we can easily interpret the total latency of processing a frame. Results show that there are no significant differences among datasets. On the image loading and processing tasks (those are Load File, Image Rect, ORB ext, and Stereo match) there are only subtle differences among them attributable to stochastic reasons or small complexity diversity in image processing.



(a) Task distribution per dataset.

(b) Histogram of task latencies for MH01.

Figure 4.2: Task latency metrics for the baseline version (n=10) for both Vicon and Machine Hall datasets

The remaining tasks (mainly Pose pred and LM track) are responsible for tracking the image. In this case, the differences are still not remarkable, but more significant. We see that Machine Hall datasets (Figure 4.1a) take longer to track than ViconRoom (Figure 4.1b). In those datasets, the cluttered environments increase the feature density. Therefore, any given `Frame` has the `KeyPoints` more densely packed, so for matching a MapPoint, it has to be checked against more candidate `KeyPoints`. Also, the resultant maps include a higher number of `MapPoints`, so, again, more checks have to be performed when tracking the `LocalMap`.

Another interesting observation is the case of V203, where ORB-SLAM3 cannot successfully track the camera (as confirmed by the high error in Figure 4.4). It systematically gets lost, so it does not manage to generate a proper map, but a lot of smaller maps. This produces a reduced LocalMap that, analogously to the previous discussion, reduces the execution time. Notice that this is not a desirable outcome: fast executions are desirable but never at the expense of prediction precision. Therefore, it is mandatory to take into account the error of the trajectory to assure that such situations do not occur.

Figure 4.2b plots an histogram of task latencies for MH01. Most tasks present Gaussian shapes, which is the expected behaviour for a repetitive execution. As an exception, Stereo match presents three peaks. As described in Section 3.3.2, this task computes matches between the two stereo images. There will be images where it is impossible to get the target of 1200 `KeyPoints` because the image does not have enough texture or because of a quick motion that produces a blurry image. These images will be quicker to track. Conversely, scenes with lots or KeyPoints (more precisely, densely packed KeyPoints) present longer latencies because of the increased number of

match-checking iterations needed. This last situation usually happens at the beginning of the dataset because, in all sequences, the drone stays quiet for several seconds producing very clear images without motion blur.

For the LM track task, the situation is different since there are two peaks, but its distribution over time is particularly interesting. As can be seen in Figure 4.3, the latency tends to increase over time. This is because at the beginning the map is still small. As it grows by the insertion of new `KeyFrames` and `MapPoints`, the vector of `LocalMapPoints` that are potentially seen for a given Frame also increases, producing the observed higher latency.

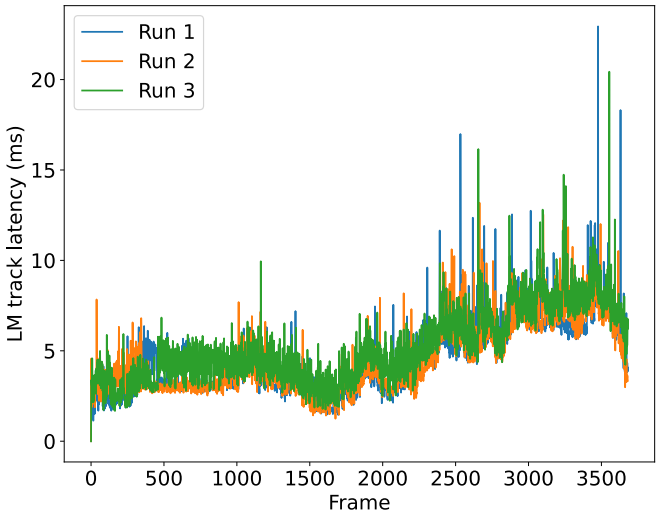


Figure 4.3: Evolution of the latency for LM track task on the MH01 dataset for three different execution runs.

According to the previous analysis, and specially Fig. 4.2a, on average, all the tracking tasks together only take 8 ms, which only represents around 20% of the total frame latency. Most of the time is spent processing the image (extracting features and performing stereo matching between the two images) which takes about 70% of the frame's execution time, which offers potential for a parallel pipeline implementation.

4.4 Pipeline Implementation

First, this section checks the correctness of the pipeline version by validating the Absolute Trajectory Error. Then, the section explores the fine-tuning of the pipeline to optimize performance by selecting the optimal number of tokens.

4.4.1 Pipeline version Correctness Validation

After implementing the pipeline as described in Section 3.4, we validate its behavior by using it with many number of tokens to guarantee that the dependencies between tasks never break.

Since the pipeline required small changes in the algorithm, we must ensure that the output trajectories remains correct. For this, we check the Absolute Trajectory Error (ATE), which is the conventional metric for checking the error of a trajectory. What we are interested in is that this value remains the same to the baseline version no matter the number of pipeline tokens, as shown in Figure 4.4, where there is no statistically significant difference between the errors. Then, we conclude that our modifications did not transform into a less precise trajectory estimation.

4.4.2 Latency Analysis

Figure 4.5 shows the latency, or elapsed time, of the different tasks for several numbers of pipeline tokens. In general, increasing the number of tokens produces an increased latency of the tasks. This is because a larger number of tokens; i.e., number of in-fligh frames, imposes a larger overhead at the scheduling runtime. However, overall throughput improves because more work is advancing in parallel, as we will explore in section 4.4.3.

As described in Section 3.4, the ORB extraction task (ORB ext in Figure 4.5) which processed the two stereo images are processed in parallel. In our version that is done by calling `tbb::parallel_invoke`. For this execution to run concurrently, the TBB scheduler must get two free worker threads. As more token pipelines are used, fewer free workers are available at a given moment, because they are busy computing other stages. Then, we observe more latency because the execution is delayed until TBB decides that can safely start. There will be instances where some worker is available and the computation can start immediately. However, situations will arise where no workers are available and the latency will be particularly big. These instances are very bad, because there is a risk that they will delay the full execution if the image is not ready for the Track stage in time.

In the scenario where the execution starts immediately (which always happens for 1 token since all the threads are free), the latency is less than the baseline version (please see Fig. 4.5). Ideally, it should be equal to the sequential version or slightly worse because of the overhead of managing the pipeline. However, TBB algorithms (such as the parallel pipeline used) put the focus on preserving the locality of the data so the cache is hotter and fewer misses happen. In the baseline version, by using two

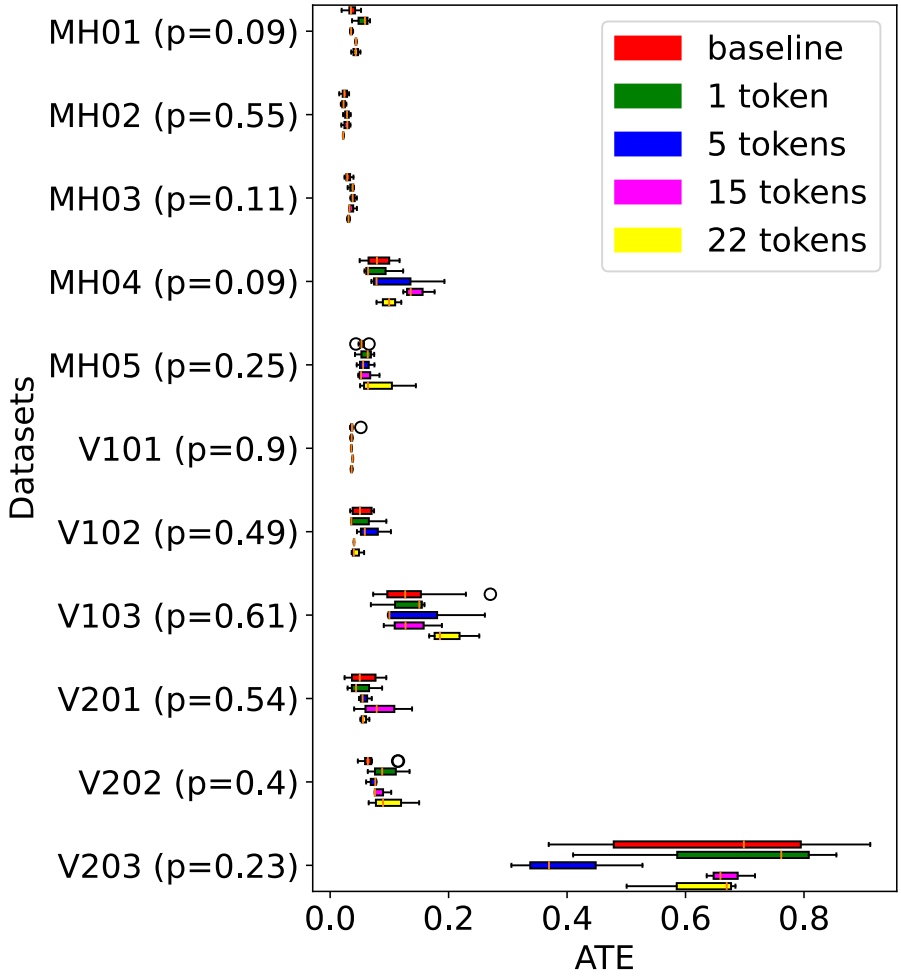


Figure 4.4: Absolute Trajectory Error of the baseline version and different number of tokens for the pipeline: 1 and 22 correspond to the sequential and best value and 5 and 15 to have two extra points in between. Each dataset is accompanied by the p-value of performing a one-way analysis variance test (ANOVA) with all of the results. Since neither of them is under the typical significance threshold $p = 0.05$, we cannot rule out that the measures are the same.

instances of `std::thread`, the processed images are not always on the relevant core's cache. Moreover, the instantiation of two `std::thread` for each frame also comes with an overhead Vs TBB that re-uses an already running working thread.

For the Tracking tasks, we see in Figure 4.5 that Pose Prediction (current Frame to last Frame matching), the latency remains the same as in the baseline version. However, for the LocalMap Tracking (LM track) it is increased. This happens because for those number of stages, as we will explore in Section 4.4.3 the Tracking happens more often, and hence we have more synchronization with the LocalMapping thread, blocking the execution sometimes.

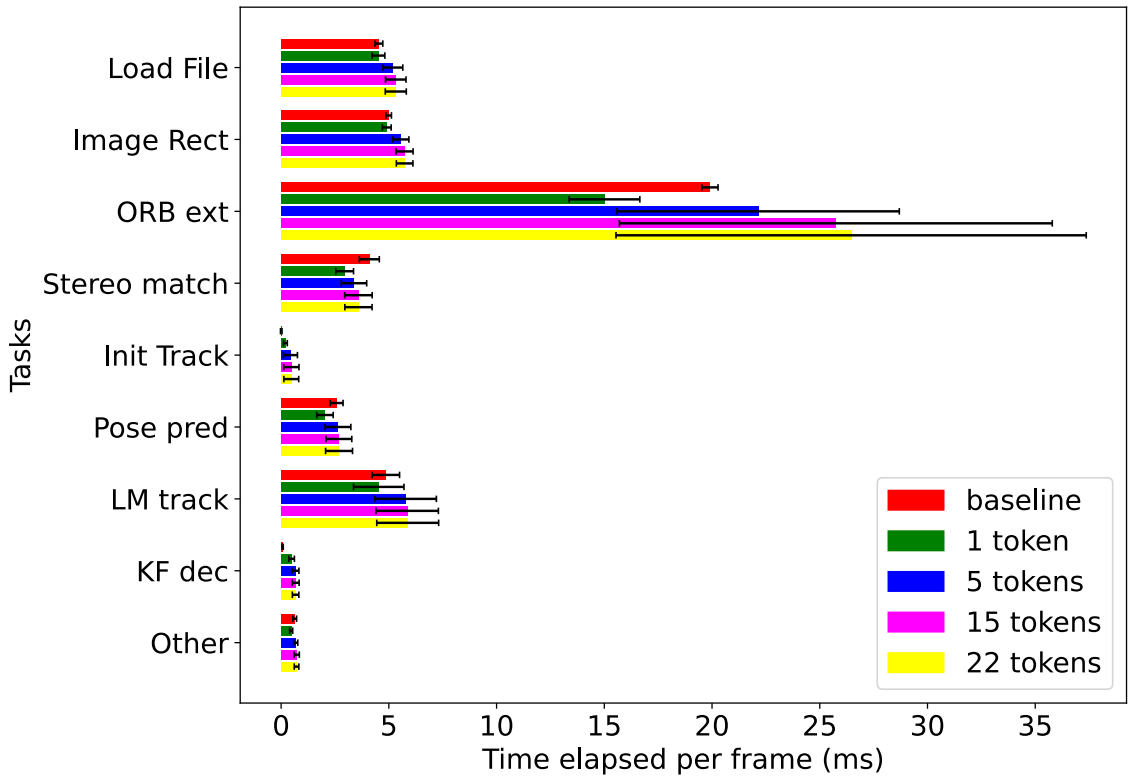
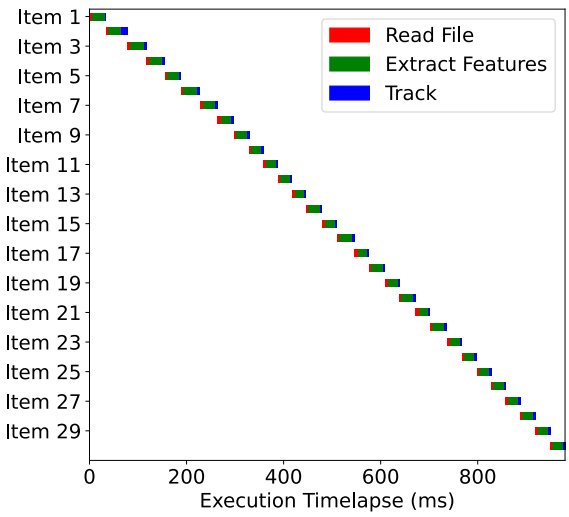


Figure 4.5: Latency of tasks for different versions on the MH01 dataset.

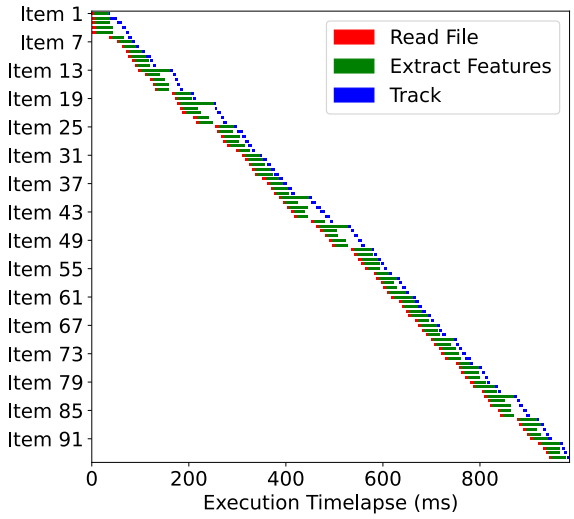
4.4.3 Choice of the Best Pipeline

By only analysing the latency metrics of the previous section, one could get the wrong impression that the highly tokenized pipeline decreases performance. But the key observation is that in those cases, the loading and processing of the images are performed in parallel thanks to the pipeline. So even more time is spent to perform each individual `Frame`, the throughput is higher. We can clearly see this in the schemes of Figure 4.6.

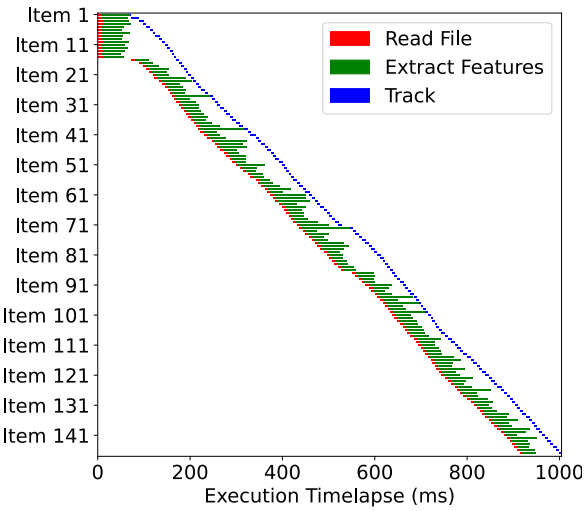
For 1 token in Figure 4.6a, the program works sequentially. Only one item (image) is processed at a time. Until the last stage (Tracking) of one image ends, the first stage of the next item does not start. As the number of tokens increase, more images can be processed in parallel, but the tracking remains sequential because of the dependency with itself. In the pipelines with more than one token, the tracking does wait often for the processing of an image, since that computation has already been performed and stored in a buffer, so the tracking can start right away. On the contrary, if the number of tokens is not large enough; e.g, when a particularly high latency task is found on the first two parallel stage, then the tracking must wait for it. As we progressively increase the number of tokens to 5 and 15, these scenarios are less common (in the execution portrayed on Figure 4.6c) we see one instance around Item 71, where the second stage



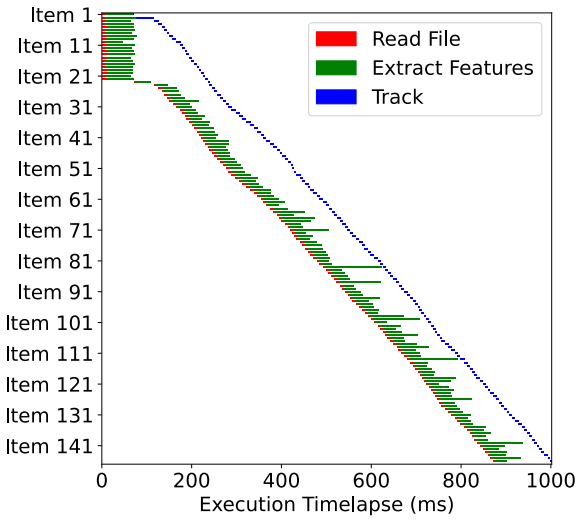
(a) Pipeline with 1 token.



(b) Pipeline with 5 tokens.



(c) Pipeline with 15 tokens.



(d) Pipeline with 22 tokens.

Figure 4.6: Real pipeline measures for the first second of execution for different numbers of tokens. The number of tokens is represented by the number of items processed simultaneously. For a high number of tokens, the `Track` stage can advance seemingly without having to wait for a `Frame` to be processed. Please note that each item corresponds to a pair of images that will form a frame

(green bar) takes too long, and the last stage (blue bar) cannot start. For the fastest version of 22 tokens (Figure 4.6d) this is not observed in the diagram. In this scenario, even though some images take very long to process (as can be seen by very long green bars representing the latency of that stage), the accumulated images in the buffer are enough to dampen this effect and still feed Frames to the tracking stage and allow it to advance seemliness.

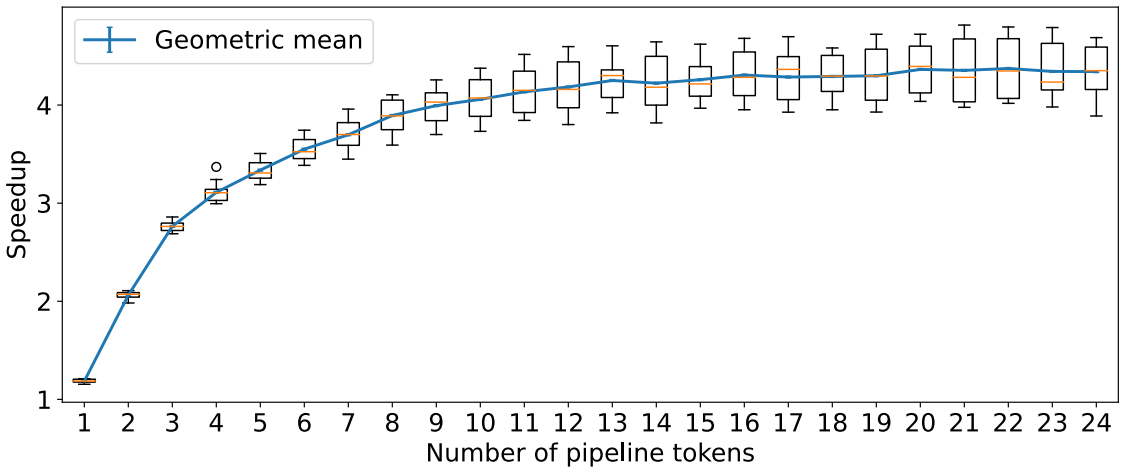


Figure 4.7: Speedup obtained with a varying number of pipeline tokens. The boxplots represent the distribution of the 11 datasets and the blue curve is the geometric mean of its measurements.

We can sense that the execution is faster the higher the number of tokens since there are more items on the Y axis of Figure 4.6 for the same timelapse of 1 second. But the definite answer comes when looking at Figure 4.7, where the speedup, geometric mean, against the baseline version is shown.

Some datasets have more speedup than others, so as a combined metric, the geometric mean of all of them is used as a reference. In the beginning, a small increase in the number of tokens correlates with a high increase in speedup. But as more tokens are used, diminishing returns are observed and less speedup is gained until it saturates and the increase becomes negligible. This is no other than Amdahl's law as described in Section 2.2. With the analysis we have done by looking at Figure 4.6, we perfectly understand it now. Adding more parallelization (pipeline tokens) to the already parallel section of the program does not improve the overall execution, which becomes limited by the sequential segment (Tracking).

The parallelization of the first two stages is obtained for a fairly low number of tokens. Increasing them over this point improves the "cushion effect" that provides a longer buffer to store already available Frames to dampen the effect of high latency of the first two stages. The great news is that we have managed the maximum ideal performance of this pipeline design, where the execution time of the program is just all the tracking phases, where the rest of the execution (loading and processing the images) is handled concurrently on the background.

To choose the best number of tokens of the pipeline, we take the maximum of Figure 4.7 that happens at 22. The differences in its vicinity are very small and not

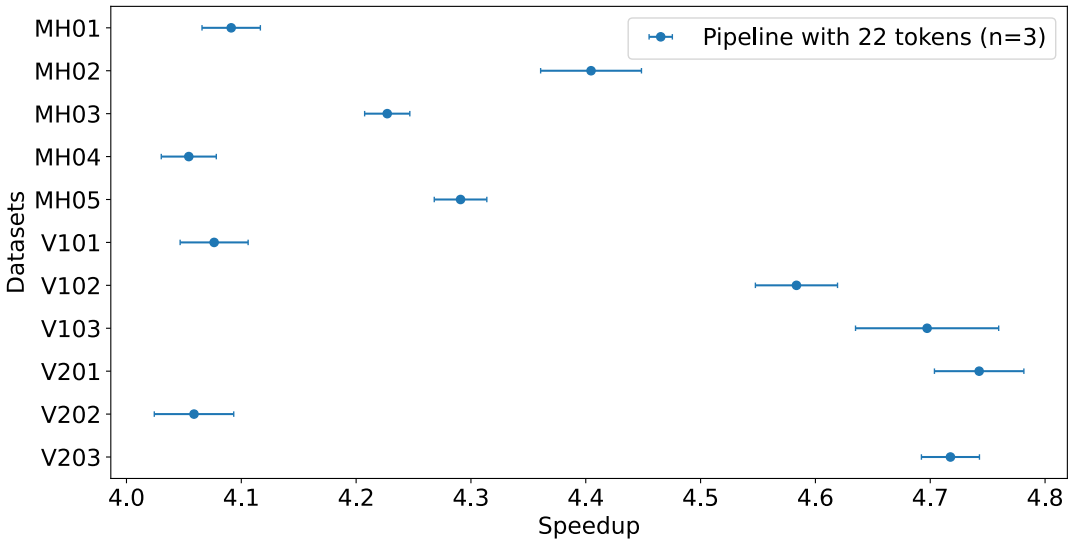


Figure 4.8: Speedup obtained for the optimal pipeline (22 tokens) for the 11 EuRoC datasets.

statistically significant, but 22 is not only the nominal maximum (albeit probably for stochastic reasons) but also checks out with the fact that the effective number of available threads is also 22. From the 24 that the machine has, two threads are used by ORB-SLAM on `LocalMapping` and `LoopClosing` respectively. Then, 22 makes sense both statistically and semantically. The performance gain shown in speedup for 22 tokens can be seen in Figure 4.8 where the speedup for the different datasets is in the range between x4 and x4.8 for a geometric mean of all of them of x4.38.

4.5 Single Tasks Optimization: Parallel-For Implementations

Having reached the ideal limit of parallelism for the pipeline, it is time to turn our attention to the next step. It is important to not lose perspective and only optimize the sections that really matter. At this stage of development, a negligible improvement will be achieved for optimizing the first two stages. As can be seen in Figure 4.6, those stages are already fully parallelized in the sense that all of them happen concurrently and the Tracking stage always has an available Frame to process. Parallelizing them even more could reduce its latency (which could be marginally positive since that reduces the risk of emptying the buffer of Frames), but not significantly improve the performance. As already discussed in the previous Section 4.4 we have reduced the execution to sequentially perform the tracking stage. Therefore, This tracking stage is where the efforts should be focused.

We saw that the Track stage has a strong dependency on itself, so there is not a lot of space to operate, but some extra parallelism can be squeezed by performing some long loops in parallel. The tasks defined for characterization in Section 4.3 are too coarse-grained to perform this evaluation. So we can use perf to enlighten our search. By using `perf record/report` as seen in Section 2.3.2, we get the following report for the child functions of `Track`:

```
- 10,02% ORB_SLAM3::Tracking::Track
        - 6,68% ORB_SLAM3::Tracking::TrackLocalMap
            + 2,74% ORB_SLAM3::Optimizer::PoseOptimization
            - 2,22% ORB_SLAM3::Tracking::SearchLocalPoints
                 1,01% ORB_SLAM3::ORBmatcher::SearchByProjection
                 0,75% ORB_SLAM3::Frame::isInFrustum
              1,28% ORB_SLAM3::Tracking::UpdateLocalPoints
        - 2,85% ORB_SLAM3::Tracking::TrackWithMotionModel
            + 2,14% ORB_SLAM3::Optimizer::PoseOptimization
              0,64% ORB_SLAM3::ORBmatcher::SearchByProjection
```

This report and the subsequent analysis are performed only on the dataset MH01 for simplicity reasons. We have already validated the idea that there are no great differences among the EuRoC datasets, so we will analyze only one of them and validate our conclusions on all of them again at the end. Now we can look closer into the culprits and identify three potential loops that are strong candidates for parallelization:

- Parallel-For1 (PF1): It is inside the task Pose Pred and corresponds to the `SearchByProjection` function inside `TrackWithMotionMode`. It iterates all the `MapPoints` of the last frame and looks for matches in the current frame for each one. Is in the order of 1000 iterations and takes $0.63395 \pm 0.24557ms$ per frame.

- Parallel-ForF2 (PF2): Corresponds to the task Track LM and is associated with creating the `LocalMap`. Once the `LocalMapPoints` are identified, all of them are iterated and a computation is evaluated to check if they are in frustum with the frame (`Frame::isInFrustum` on the perf report). This is a complex operation since it involves several matrix multiplications and trigonometric computations. Its loop size typically ranges from 1000 to 5000 iterations and takes $1.23917 \pm 0.77683ms$ per frame.

- Parallel-ForF3 (PF3): It is the analogous of PF1 but for the `LocalMap`. On the perf report, it is the other `SearchByProjection` child of `SearchLocalPoints` It iterates the `LocalMapPoints` that survived the filter of PF2 and looks for matches in the current `Frame`. For this loop, the number of iterations is around 1000 and takes $0.92918 \pm 0.53962ms$ per frame.

The added contribution of the parallel for selected accounts optimistically for around 20% of the Tracking latency, that is approximately $8 - 9ms$ per frame. Hence, the

expected improvement is much less than in the pipeline implementation. Even if we could potentially reduce the target sections to essentially zero, we would still be left with the remaining 80% of the sequential segment. Unfortunately, as per the perf report we see that more than 50% of the time is used on `Pose Optimization`, but the parallelization of this work is out of the scope of this work since the optimizer belongs to a third-party library.

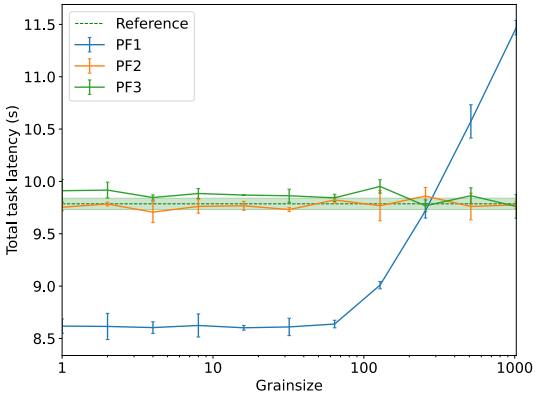
We mentioned in Section 3.4 that an essential parameter of the `tbb::parallel_for` is the number of chunks the work is subdivided on. However, TBB does not expose this parameter directly but does so in the form of the `grainsize`. This is an indication of how big (how many loop elements) every chunk should be. In practice, this is a more natural way of defining chunking, since it allows to have chunks of constant work. If for a given execution of the loop, more iterations are needed, what will happen is that more chunks will be spawned of roughly the same size.

In the process of implementation, we noticed that PF2 had a piece of work that was related to the viewer thread. This task is never performed in our version, so we remove it and add it as an optimization. For reference, we show in the following results the base time that this loop takes with and without this task on the sequential version.

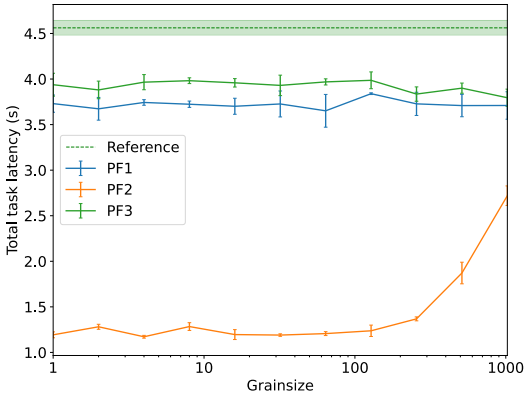
We put new fine-grained timers to measure specifically the regions of interest and launch three experiments. In a given execution, only one of the three parallel fors is active and the other two are left sequential. This way we can evaluate the performance of each loop in isolation. The results can be seen in Figure 4.9. We see that, in fact, there are interactions between the regions. For example, by analyzing the PF3 region, the PF2 curve is over the reference by about 0.5 s. This means that the inclusion of the parallel for on the PF2 loop, slightly worsened the PF3 region. No other significant interactions are observed.

As expected a bigger `grainsize` produces fewer chunks, hence less room for parallelization an potentially unbalanced workload. The expected reciprocal was that smaller `grainsize` produces a lot of chunks and this extreme parallelization does not overcome the overhead penalties of its management. However, this is not observed. In fact, the best points of minimal latency are found for very small `grainsizes`. The explanation is that most of the TBB threads are busy on the pipeline, and the task scheduler can only allocate a handful of them to the execution of parallel fors. Then, even though a lot of chunks are created, they are not frequently executed concurrently because there are not so many free worker threads. The optimal `grainsizes` are collected in Table 4.1.

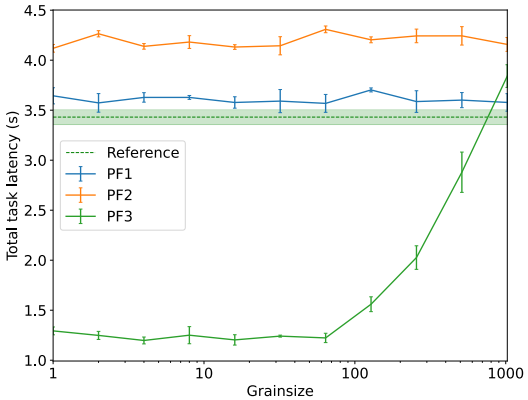
Finally, we also check the latency of the full algorithm on Figure 4.9d. This validation is important because we could have been worsening the execution in other places. We see



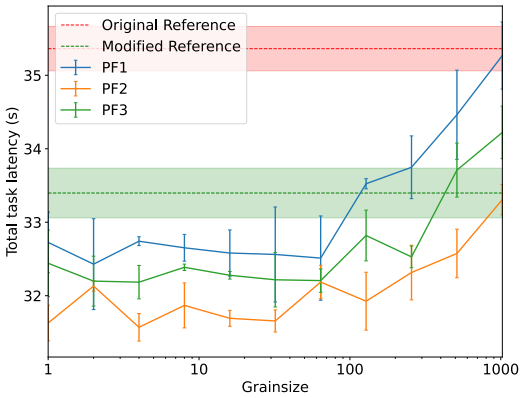
(a) Total latency of the PF1 region.



(b) Total latency of the PF2 region.



(c) Total latency of the PF3 region.



(d) Total latency of the full algorithm.

Figure 4.9: Optimization obtained for different *grainsize* (notice the logarithmic scale) on dataset MH01 for the three parallel fors compared with sequential references (n=10). The shaded region of the references indicate standard deviation. Each curve represents a program (n=3) with only that parallel for activated and the other two are sequential. For PF2 and t_algo(rithm), the modified reference corresponds to the version with the unnecessary work being removed.

that this is not the case, since all of the parallel fors improve the references. Remarkably, the improvement obtained from removing the viewer work (modified reference) is larger than any of the individual PF implementations. But we are most definitely not going to reject free performance.

Region	`grainsize`
PF1	16
PF2	4
PF3	4

Table 4.1: Optimal *grainsizes* for each region.

4.6 Final Evaluation

Having chosen one optimal value for each parallel for, we perform one last experiment where all of the three parallel fors are active simultaneously with its optimal `grainsize` in isolation. It is plausible that this value is non-optimal when combining the parallel fors because there is some degree of interaction among them as we have already shown. However, the selection of the best chunking is quite ambiguous because we show in Figure 4.9 that there is a big flat region where the differences are hardly statistically significant, so even if the final value is not the optimal but close enough to it, we expect similar results nevertheless.

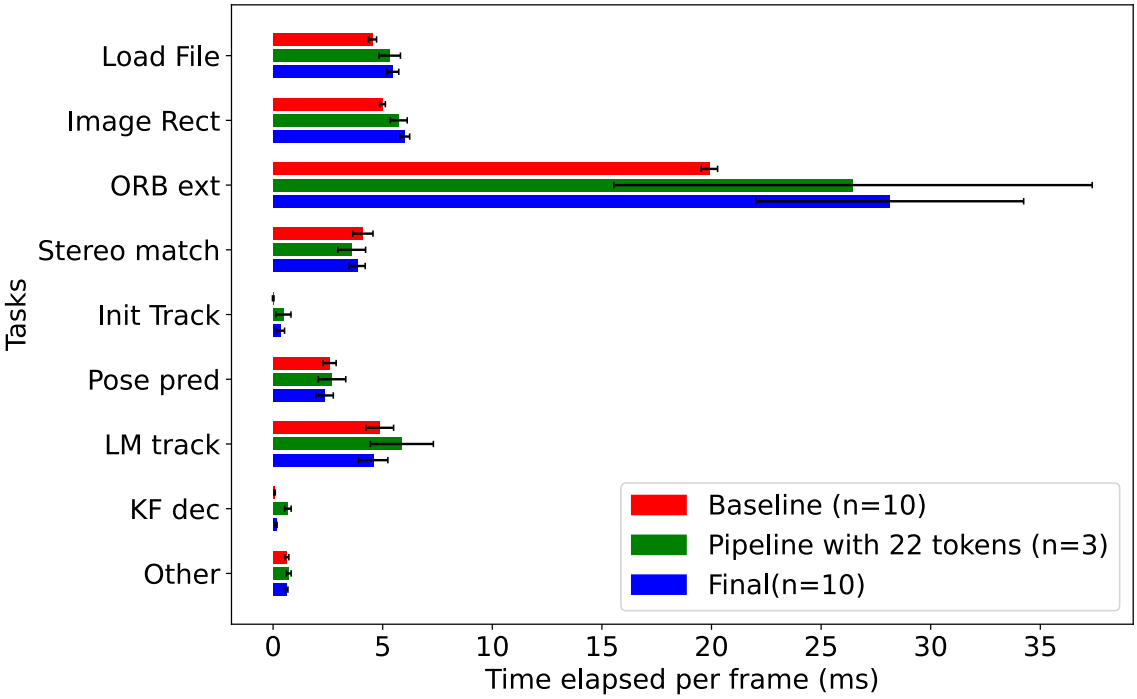


Figure 4.10: Latencies of the tasks on three versions tested.Final corresponds to the pipeline plus the parallel for version. n represents the number of trials.

We first show the improvement managed with all the implementations put together. The first thing to observe is how latency is affected on Figure 4.10. We focus our attention on the tasks `Pose pred` and `LM track` that were the targets of the *parallel for* optimization. We discussed in Section 4.4.2 how the improved performance caused `Tracking` to execute more often and then have more synchronization with the `LocalMapping`, which translated on higher latency for the `LM track` task, where the shared data structures are accessed by both threads simultaneously. For the final version, we should expect even more synchronization because we are going even more

faster. However, the optimizations overcome this issue and we get even less latency than in the original baseline version. However, the latency of the `ORB ext` task is even higher because now more threads are occupied doing parallel fors inside `Tracking` and, then, it is harder to get two workers for this task. However, since this stage is completely overlapped under the tracking, it does not affect the final performance.

Next, we validate our execution by checking the ATE on each dataset in Figure 4.13.

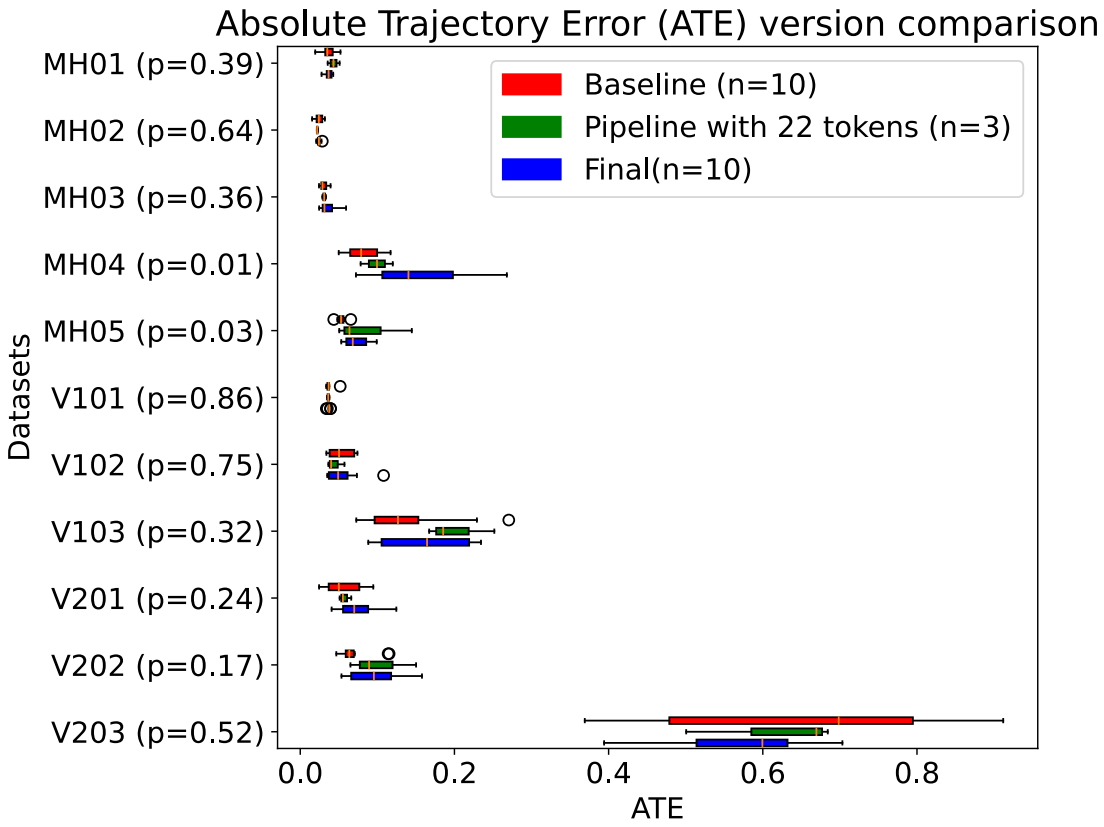


Figure 4.11: Absolute Trajectory Error of the baseline version, optimal pipeline with 22 tokens and final version. Each dataset is accompanied by the p-value of performing a one-way analysis variance test (ANOVA) with all of the results. MH04 and MH05 datasets are under the typical significance threshold $p = 0.05$, so we accept the null hypothesis for those datasets and confirm statistical signnificant differences.

We can observe that for the two hardest Machine Hall (MH04 and MH05) datasets there are some differences and the trajectory predicted has more error than the baseline and pipelined versions. The accuracy reduction is not very big, but it is statistically significant. We attribute this change to the fact that we are now tracking much faster than on the sequential (and pipelined) versions. But the `LocalMapping` or `LoopClosing` threads have not been changed, so it takes the same amount of time to process. This means that there will be some `Frames` use a LocalMap that is still unoptimized because

the other threads are still working on it. While on the slower versions, that same `Frame` had available an already refined version of the `Map`. Thankfully, ORB-SLAM is very robust and handles non-optimal reference of Poses and Positions thanks to its Bundle Adjustment approach, but we are reaching a point where the precision of the algorithm is getting affected. If additional performance was to be obtained, it would probably require to be paired with similar improvements on `LocalMapping` or `LoopClosing` so that the map is updated quicker and better versions are available sooner for the Tracking.

Finally, let's answer the most important question: How much does the final version improve the performance in terms of throughput, or frames per second?

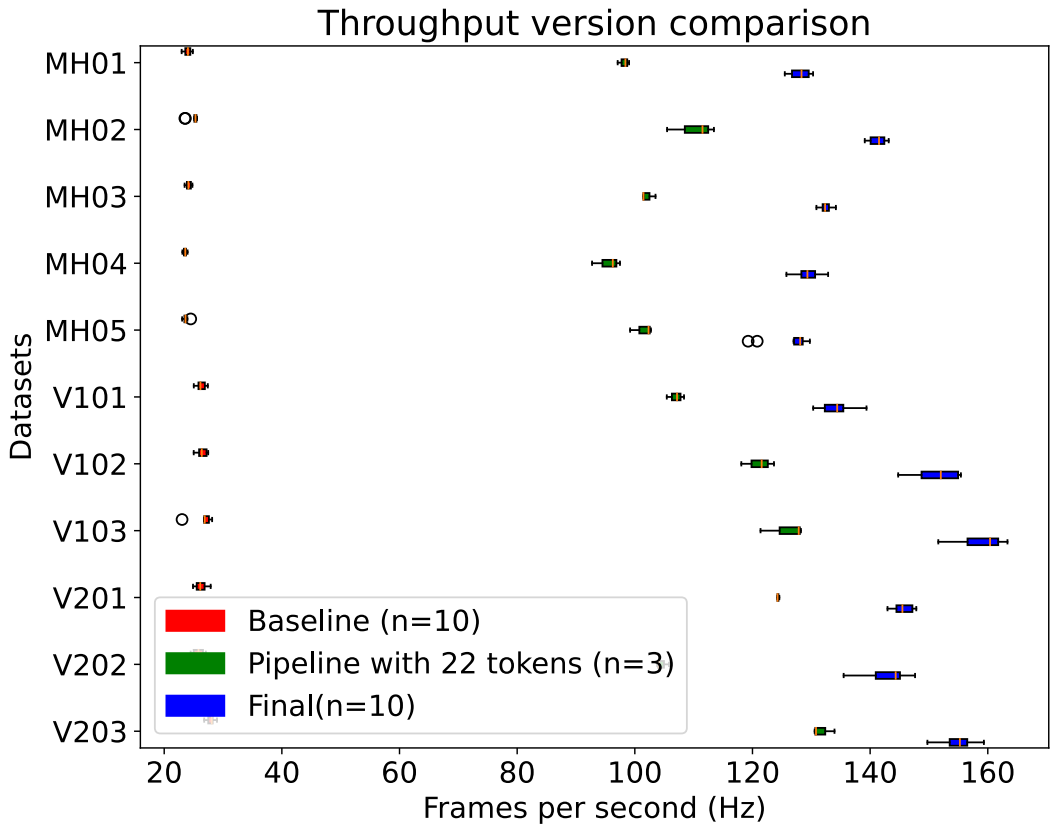


Figure 4.12: Throughput comparison of the three tested versions

We can see in Figure 4.12 that the pipelined version achieved a throughput of 100-130 FPS depending on the dataset and we managed to increase that to 130-160 FPS on the final version. Notice that by only analyzing the latency improvement as per Figure 4.10, the potential improvement might seem small. If we were to apply the parallel fors improvement to the baseline version, the improvement would be, in fact, small since in that version what takes more time is definitely the image processing.

But since this is applied to an already parallelized version where the cost of image processing has been hided and the bottleneck is only the `Tracking` stage, it has a very significant effect.

By analyzing the speedup in Figure 4.13 we finally get the average improvement on all datasets by taking the geometric mean of datasets as we alredy did in Section 4.4.3 and see that we managed 4.4× for the pipelined version and improved that value to 5.6× on the final version.

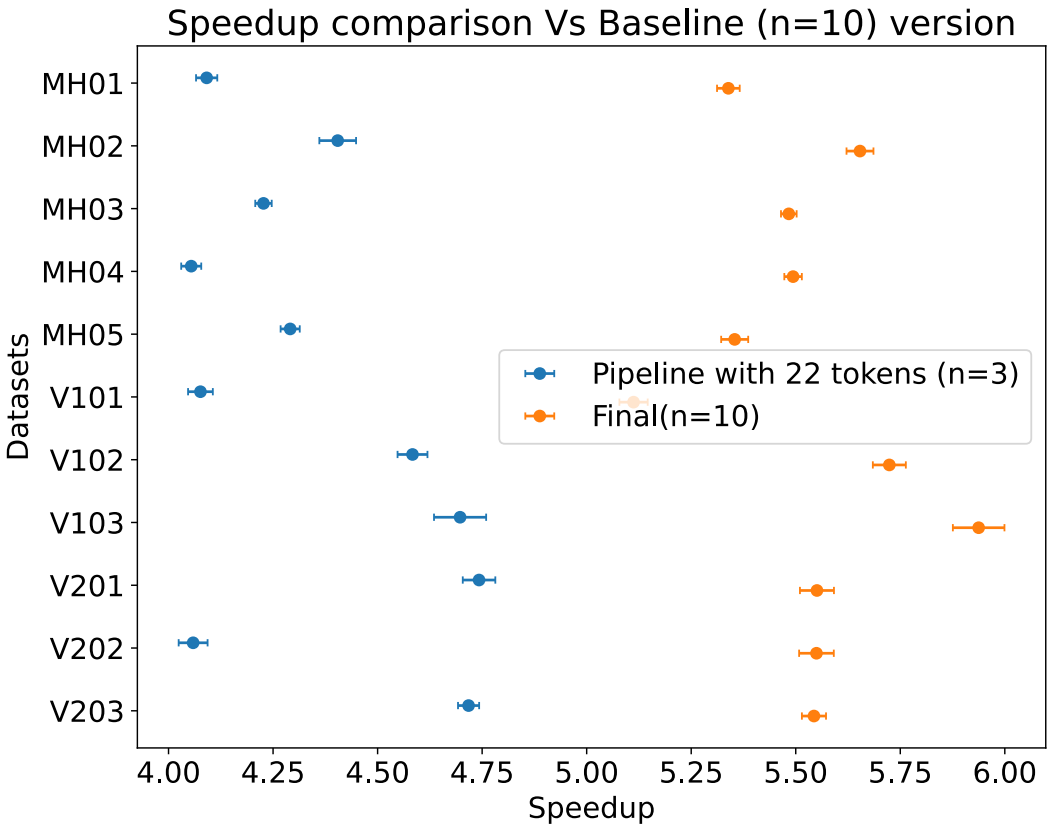


Figure 4.13: Speedup obtained with the final version.

Chapter 5

Conclusions

5.1 Contibution of the thesis

Our work has evaluated a production-level application such as ORB-SLAM3 to determine several potential parallelization strategies and shown the performance improvements derived from them.

The separation that ORB-SLAM3 does of the `LocalMapping` and `LoopClosing` provides an advantage when compared to the hypothetical scenario where everything was done on one thread and the Tracking would need to stop every time that a new `KeyFrame` is inserted. However, this approach does not show great scalability. It benefits from having 3 cores in the system where the three threads can run parallelly, but the addition of more cores does not translate in performance because the application cannot take advantage of it.

On the contrary, The proposed parallel strategies do scale and can take advantage of having more cores. The final and from this fact derive the $\tilde{x}5.6$ speedup obtained versus the sequential version, managing throughputs of up to 160FPS for some datasets.

5.2 Future work

Several topics remain up for exploration. More parallel algorithms could be used after more exploration of the code based on the method of this work. In fact, we identified other opportunities, but they were not possible to parallelize without altering significantly the structure of the algorithm, which was against our foundational principle of not interfering with the original ORB-SLAM3 algorithm.

More changes that could be used are data structures specially designed to be accessed concurrently, instead of STL objects. This would allow to remove the locking-based synchronization among threads and reduce this cost. On the same line, more relaxed memory access could be studied, since some of the locking mechanisms might be

redundant in some of the cases studied.

More validation work is needed to asses out performance improvement as a general case. We have finetuned our parallel algorithms hyper-parameters for the EuRoC dataset and the particular machine on which the experiments are run. A different dataset family, could have different latency distribution (for example, if the images are of lower resolution it is expected that the image processing would have less latency). Another environment with less cores available could also present other parameters for the algorithms to be optimal.

Then, the validation of optimality in other environments is left as an important task for the future, as well as a dynamic mechanism of adaptation of the hyper-parameters so they don't need to be fine-tuned for each tuple of dataset and system.

Chapter 6

Bibliography

[1] ESP32 Series Datasheet v4.3. Technical report, Espressif Systems, 07 2023.

[2] P. Gepner and M.F. Kowalik. Multi-core processors: New way to achieve high system performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13, 2006.

[3] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016.

[4] Linyan Cui and Chaowei Ma. Sof-slam: A semantic visual slam for dynamic environments. *IEEE Access*, 7:166528–166539, 2019.

[5] Phuc H. Truong, Sujeong You, and Sanghoon Ji. Object detection-based semantic map building for a semantic visual slam system. In *2020 20th International Conference on Control, Automation and Systems (ICCAS)*, pages 1198–1201, 2020.

[6] Raúl Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. Orb-slam: A versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[7] Carlos Campos, Richard Elvira, Juan J. Gómez Rodríguez, José M. M. Montiel, and Juan D. Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.

[8] Qiuyu Zang, Kehua Zhang, Ling Wang, and Lintong Wu. An adaptive orb-slam3 system for outdoor dynamic environments. *Sensors*, 23(3), 2023.

[9] Arash Azimi, Ali Hosseininaveh Ahmadabadian, and Fabio Remondino. Pks: A photogrammetric key-frame selection method for visual-inertial systems built on orb-slam3. *ISPRS Journal of Photogrammetry and Remote Sensing*, 191:18–32, September 2022.

[10] Laura Oliva Maza, Florian Steidle, Julian Klodmann, Klaus Strobl, and Rudolph Triebel. An orb-slam3-based approach for surgical navigation in ureteroscopy. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization*, 11(4):1005–1011, December 2022.

[11] Meng Cao, Jia Zhang, and Wenjie Chen. Visual-inertial-laser slam based on orb-slam3. *Unmanned Systems*, page 1–10, April 2023.

[12] Xinge Zhao, Qingwu Hu, Xujie Zhang, and Haiyin Wang. An orb-slam3 autonomous positioning and orientation approach using 360-degree panoramic video. In *2022 29th International Conference on Geoinformatics*, pages 1–7, 2022.

[13] C. Pereira, G. Falcao, and L. A. Alexandre. Pragma-oriented parallelization of the direct sparse odometry slam algorithm. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 252–259, 2019.

[14] Liuxin Sun, Junyu Wei, Shaojing Su, and Peng Wu. Solo-slam: A parallel semantic slam algorithm for dynamic scenes. *Sensors*, 22(18):6977, September 2022.

[15] Raúl Taranco, José-Maria Arnau, and Antonio González. Locator: Low-power orb accelerator for autonomous cars. *Journal of Parallel and Distributed Computing*, 174:32–45, April 2023.

[16] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[17] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[18] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress, 2019.

[19] Raúl Mur-Artal and Juan D. Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[20] Dorian Galvez-López and Juan D. Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012.

[21] Raúl Mur-Artal and Juan D. Tardós. Visual-inertial monocular slam with map reuse. *IEEE Robotics and Automation Letters*, 2(2):796–803, 2017.

[22] Carlos Campos, José M.M. Montiel, and Juan D. Tardós. Inertial-only optimization for visual-inertial initialization. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 51–57, 2020.

[23] Richard Elvira, Juan D. Tardós, and J.M.M. Montiel. Orbslam-atlas: a robust and accurate multi-map system. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6253–6259, 2019.

[24] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011.

[25] Piotr Dollár, Ron Appel, Serge Belongie, and Pietro Perona. Fast feature pyramids for object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(8):1532–1545, 2014.

[26] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 2016.

[27] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016.

[28] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 225–234, 2007.

[29] Mark Buckler, Suren Jayasuriya, and Adrian Sampson. Reconfiguring the imaging pipeline for computer vision. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 975–984, 2017.

[30] Raúl Taranco, José-Maria Arnau, and Antonio González. A low-power hardware accelerator for orb feature extraction in self-driving cars. In *2021 IEEE*

33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pages 11–21, 2021.

[31] Raúl Taranco, José-Maria Arnau, and Antonio González. Locator: Low-power orb accelerator for autonomous cars. *Journal of Parallel and Distributed Computing*, 174:32–45, 2023.

List of Figures

List of Tables

Appendices

Appendix A

Terminology of Parallelism

Specific vocabulary and some definitions will be used throughout this text, so in order to set a common ground it is important to properly define them. Also, this section can help to internalize some basic concepts of parallel programming.

At the highest level, parallelism is found in the possibility of data to operate in parallel, or in the form of parallel-task execution. These two options are not exclusive and a mixed parallelism is not only possible but desirable.

Task parallelism Task parallelism refers to different, independent tasks that are applied to the same data. For example, one can imagine a dataset of numbers, to which different mathematical operations are applied. We could parallelly compute the arithmetic mean, the standard deviation, the maximum, and the minimum. Finding opportunities for task parallelism becomes limited by the number of independent operations that can be envisioned.

Data parallelism For a given data, data parallelism applies the same operation to all of its elements. For example, starting with a piece of text, we could apply a transformation that turns all the letters uppercase. In this simple example, since the operation can be applied element-wise, we can perform the same task in parallel to each element.

Approaches based only on task parallelism are limited by the different task types that an application has. But data parallelism approaches are limited by the amount of data that can be processed in parallel on a particular machine. It is very important to find opportunities for data parallelism in order to have a scalable parallel program as we discussed in Section 2.2.

Pipelining A specific type of task parallelism Is worth highlighting pipelining. In fact, it will be a core part of this work. In this parallel pattern, many tasks need to be

applied to a stream of data. But instead of performing all of them concurrently for the same piece of data element, it is applied one after another. The neat thing is that as soon as the first task is applied to the first element, it is applied again to the second element at the same time that the second task is applied to the first element, and so on.

This is very reminiscent of Henry Fords' vision of production chain in a factory, where the data flows through the pipeline or, in other words, the pipeline moves advancing through the data.

The data can be processed faster than sequentially because different items can pass through different stages at the same time. Also, notice how the time it takes for a single element to be computed is the same (referred to as latency). In practice, it can even be slightly worse because of overhead effects on the management of the pipeline. However, the big advantage is that the throughput is greater. If we imagine that all stages take the same time and all the items move uniformly (which is definitely not the case in a real scenario), one item is produced by the pipeline per stage length once the pipeline is full. Whereas in a sequential scenario, the new data item must perform all the stages before ending the process.

Appendix B

Not so brief review of the ORBSLAM3 system

ORBSAM is a complex program, and its description can be approached from various angles. In this chapter, we aim to provide an illustrative explanation to those readers that are not familiar with ORBSLAM (or SLAM in general). This means that some operations are explained disordered in relation to how they occur in the real system because narratively seems more appropriate to be addressed sooner, but not algorithmically. Also, some complex details (such as scale pyramids or rotation invariance validation of the ORB features) are omitted. In fact, only the main path is described, since the situations where this path is not followed are uncommon and their contribution to the execution time is negligible according to our measurements. Especially in the tracking, there are special situations (such as map initialization) or diverging branches (when matching fails or the system gets lost, more generous thresholds for matching are accepted) that are ignored for the sake of simplicity. For a detailed and complete description of the algorithm, please refer to the original papers[6][19][7] or the source code.[1]

B.1 Tracking: The Image's Journey

The starting point is always an image. In real-life situations, it will come from a camera, but for most research works it is taken from pre-recorded image stream datasets. The use of these datasets is crucial to compare different algorithms in a standardized way and identify which situations are more difficult to solve.

The image is first processed via feature extraction. This process identifies several points of interest or KeyPoints on the image and each of them is assigned a descriptor that identifies it. ORB-SLAM uses ORB for this matter, which are oriented multi-scale FAST corners with a 256-bit descriptor associated. Think of the descriptor as a vector,

[1] `https://github.com/UZ-SLAMLab/ORB_SLAM3`

and the difference with any other such vector should be ideally zero if the other feature is visually identical. This allows for comparing points on different images: if their descriptors are very similar, it is a promising indication that they refer to the same physical point.

At this point, the system "forgets" about the image and only those KeyPoints will be analysed. In the case of a stereo image, for every left KeyPoint, a corresponding KeyPoint is search on the right image. If the result is positive, the point depth can be triangulated according to the camera specifications. All the information of the KeyPoints and their descriptors is enclosed into a `Frame`.

After feature extraction, the Frame enters the Tracking phase. The objective of this phase is to determine the position of the camera in the world space by taking matches of the KeyPoints of the Frame both with the previous frame (visual odometry) and with the local map.

First, an initial guess of the new camera pose is done by computing a constant velocity model. [ELABORATE??]. Then, a search by projection is performed with the last Frame. This consists of taking the last Frame's 3D MapPoints and projecting them into the current Frame 2D space. In the ideal case in which both the initial current Frame's pose estimation and the last frame's MapPoints were perfect, we would find an ORB KeyPoint in the same projected position. That is, the projection of the 3D points of the last frame would lie perfectly on the 2D projection on the current frame.

However, since we do not live in a perfect world, this reprojection error will not be zero. This causes two problems: 1) to find the matching KeyPoint we must look in a 2D window of the projected point and keep the one with the minimal descriptor distance (the actual criterion is slightly more complicated), and 2) the reprojection error of the matched points is not going to be zero. However, we can minimize it by performing a non-linear optimization of the camera pose. This process of "light" bundle adjustment keeps the MapPoint's fixed and only changes the camera Pose to find the one that minimizes the reprojection error of the matched points. This gives an improved camera pose estimation over the initial one performed only with the velocity model. After an outlier filter, the matches found are "upgraded" to MapPoint also in the current frame since now they have a confident match with an observed 3D MapPoint.

The next step is to compute matches with the LocalMap. For this step, all the matches found in the comparison with the last frame (now MapPoints) are used to build the LocalMap. All the KeyFrames that have seen some of those MapPoints are added. Some extra KeyFrames are added even if they do not contain any MapPoint of the Frame as long as they are co-visible with KeyFrames that share MapPoints with the Frame. Then, the LocalMapPoints is the list of all the MapPoints that belong

to each of the LocalKeyFrames. The procedure is then quite similar to the last one: each of these LocalMapPoints is projected onto the current Frame and a search is performed in the proximity for similar KeyPoints. The matches found are upgraded too to MapPoints. After this, another pose refinement is performed that attempts to minimize the reprojection error of all the new matches and finally, the outliers of the optimizer are discarded.

We have described two matching mechanisms. The first one is Frame-to-Frame and the second one is Frame-to-LocalMap. In both cases, a list of candidate MapPoints are projected onto the current Frame and a local search is done in the vicinity of that projected position for the KeyPoints of the current Frame that are similar to the MapPoint being analysed. After each matching process, a Pose Optimization is performed to minimize the reprojection error, and the outlier MapPoints are removed. The number of MapPoints to check in the LocalMap case is going to be much larger than in the Frame-Frame situation. That is because the LocalMap is built with all the MapPoints of the KeyFrames that share some MapPoint with the current frame (that is inliers matches with the last frame of the first matching phase). Also, note that this process does not create any MapPoint, it just assigns them to KeyPoints of the current Frame, but they already exist. The MapPoints are created by the LocalMapping described in the next section B.2.

The image's journey usually ends here, since most of the Frames are only used as a disposable tool for keeping the Camera Pose up to date. But sometimes, the Frame is upgraded to KeyFrame if certain conditions are met. The actual heuristic for this decision is more complicated and can be consulted on the original ORBSLAM paper[7].

Once a new KeyFrame is queued for addition, the LocalMapping thread is activated.

B.2 LocalMapping: The Map's Refinement

When the LocalMapping is awaken, it works concurrently with the Tracking. New Frames will continue to be processed while the map gets updated by the LocalMapping thread. This process is quite resource intensive, and if it were to block the tracking, it would severely harm the real time capabilities of ORBSLAM3.

The first step is to add the new KeyFrame to the active map. As we have seen, KeyFrames serve as anchor points for the map, providing reference frames for tracking. The insertion is not as naive as it might sound, because the LocalMapping maintains an essential graph that links any two KeyFrames observing common points and a minimum spanning tree connecting all KeyFrames. These graph structures allow to retrieve local windows of KeyFrames, so that tracking and local mapping operate locally, allowing

to work on large environments. Each node of the graph is a KeyFrame and an edge between two KeyFrames exists if they share observations of the same map points, being the weight of the edge the number of common map points. Then, the bags of words representation of the KeyFrame are computed. That will help in the data association for triangulating new points.

New map points are created by triangulating ORB from covisible KeyFrames. For each unmatched ORB KeyFrame, a match with other unmatched point in other keyFrame is searched. This matching is done guided by DBoW2 similarity [LOOK FOR NEW METHOD IN OS3]. ORB pairs are triangulated, and some geometric checks are performed, such as epipolar constraint reprojection error and scale consistency. Initially, a map point is observed from two KeyFrames, but it could be matched in others, so it is projected in the rest of connected KeyFrames, and correspondences are searched in the vicinities. This creates a new MapPoint that will be identified in several of the KeyFrames that were already present in the Map.

Then, a local Bundle Adjustment optimizes the currently processed KeyFrame, and all the KeyFrames connected to it in the covisibility graph. This Bundle Adjustment is more complex than the one performed in the Tracking, where the only target variable was the current Frame pose. In this case, the Poses of all the covisible KeyFrames are updated simultaneously but, more importantly, also all the MapPoint's 3D positions seen by those KeyFrames. All other KeyFrames that see those points but are not connected to the currently processed KeyFrame are included in the optimization but remain fixed. As usual, observations that are marked as outliers are discarded at the middle and at the end of the optimization.

In general, MapPoints and KeyFrames are created with a generous policy, while a later very exigent culling mechanism oversees detecting redundant KeyFrames and wrongly matched or not trackable map points. This permits a flexible map expansion during exploration, which boost tracking robustness under hard conditions (i.e., rotations and fast movements), while its size is bounded in continual revisits to the same environment. The LocalMapping is also handled the responsibility of this task.

In the case of MapPoints, they must pass a restrictive test during the first three KeyFrames after creation in order to be retained in the map. This ensures that they are trackable and not wrongly triangulated, i.e., due to spurious data association. Once a map point has passed this test, it can only be removed if at any time it is observed from less than three KeyFrames. This can happen when KeyFrames are culled and when local bundle adjustment discards outlier observations. This policy makes the map contain very few outliers, but also very robust thanks to the low number of false negatives left unidentified thanks to the relaxed acceptance policy.

For a given KeyFrame, if most of its MapPoints have also been seen in other KeyFrames, it is deleted, since it contains redundant information. This helps prevent excessive memory usage and maintain the efficiency of the map, as bundle adjustment complexity grows with the number of KeyFrames. But at the same time, it enables lifelong operation in the same environment as the number of KeyFrames will not grow unbounded, unless the visual content in the scene changes.

B.3 LoopClosing: Going Back to the Roots

Whenever the mapping thread creates a new KeyFrame, place recognition is launched trying to detect matches with any of the KeyFrames already in the Atlas and trying to detect and close loops. In older versions of ORB-SLAM, it was also tasked with relocalization in case that the system got lost. However, with the multi-map implementation in the ORB-SLAM 3 version, a new map is started in this case.

ORB-SLAM uses the DBoW2 bag-of-words place recognition system. Given a query image in the form of its bag-of-words vectors, the system can efficiently provide the most similar KeyFrames according to their bag-of-words that were already computed when inserted by the LocalMapping. Some temporal and geometric constraints are also applied to avoid detecting false positives. If the matching KeyFrame found belongs to the active map, a loop closure is performed. However, if a loop is detected between the active map and an archived older map on the Atlas it is a multi-map data association, and the active and the matching maps are merged.

For the map merging case, a welding window is assembled by taking all the co-visible KeyFrames in both maps and transforming to a common reference. Then, the maps are merged by taking matches with the MapPoints, where the duplicated points are fused, and the essential graph is updated with the KeyFrames of both maps. Then, a local Bundle Adjustment is performed on this welding window and finally, the correction is propagated to the rest of the Map by essential-graph optimization.

The concept for loop closing is quite similar, but all the co-visible Frames belong to the same map. But an additional step is performed in the form of a global Bundle Adjustment. This is the most expensive bundle adjustment of all and involves all the KeyFrames poses of the active map and all the 3D positions of the observed MapPoints. The high cost of this refinement can only be assumed if the number of KeyFrames is below a certain threshold.