Parallel Programming on Embedded Multicore System ESP32

Universitatea Politehnica Timișoara

Marian Belean
Franz Joseph Pal

WS2019/2020
December 17, 2019

Abstract

The following documentation will focus on the principles of parallel programming in general and the mathematical background. In addition to the different parallel programming architectures, the various models for their implementation are also discussed. Moreover, in this thesis the prerequisites for mathematical calculation models, which are suitable for Parallel Programming, are elaborated.

For a practical example, the ESP32 microcontroller was chosen, an embedded multicore system. After a brief introduction to the hardware itself, further details of the project structure and the development of the application will be presented. Therefore, a short example will be explained to focus on the basics of parallel programming.

Finally, the aim of the project and the documentation is an automatic benchmark setup and a webfrontend result overview for visualization purposes, which will be discussed in more detail in the conclusion.

Contents

Chapter 1

Introduction

Multicore systems are becoming increasingly popular as part of digitalization and Industry 4.0[1] (German/EU) [2] [or 1] - also known as smart manufacturing in the USA [see 30, p1] - and are playing an important role in data processing and process automation [see 34, p294] [or 29, p1]. On the other hand, in addition to efficiency in energy consumption, performance in terms of computation time [see 34, p294] is required in almost every application field of multi-core systems.

In fact, multi-core hardware is not only especially for smart manufactoring. Nowerdays in almost every smart application like smart phones[2], wearables[3] or home automation we can find multi-core embedded hardware platforms, which guaranteed high performance [see 4, p7], network connectivity, security and reliability [see 36, p5]. This field of application is also known as Internet of Things (IoT)[4].

Especially for embedded systems mathematical models as well as numerical solutions, which can be executed both simply and parallel, are suitable. The question arises to what extent parallel execution of different sub-tasks to calculate a problem [see 33, p4] increases the desired cost factor in terms of energy consumption [see 14, chapter 3] and computational efficiency [see 33, p4 Figure 3].

[1]add.: https://www.epicor.com/en-ae/resource-center/articles/what-is-industry-4-0/

[2]e.g. ARM based processors for mobile phones like https://www.arm.com/solutions/mobile-computing/smartphones

[3]add. information on ARM based solutions and the current trend in wearables: https://www.arm.com/solutions/wearables

[4]for additional information about Internet of Things, please see [21]

Chapter 2

Overview

2.1 Problem definition

Compared to single-core execution of tasks, multi-core embedded hardware platforms like the ESP32[1] provide the ability to develop advanced parallel computing software applications to reduce execution time and power consumption.

On the one hand, a major problem is choosing the right hardware platform to meet the cost and size factor, and moreover, whether a single-core or multi-core calculation is required. Therefore, context switching time, power consumption and total execution time must be included in the evaluation.

In order to develop an optimal solution, the hardware platform must be included in addition to the mathematical model of the problem itself. So in this case, suitable prerequisites and characteristics can be worked out in order to make an evaluation of "Parallel Computation Tasks on Embedded Multicore Systems" possible.

2.2 Objective of the documentation

The main goal of this documentation is to focus on the current parallel programming techniques, depending on the execution time in general and the required mathematical model. For this purpose, an application which can compute different sections of the Mandelbrot fractal [see 20, p11] will be developed to compare single core and multi-core calculations. Before the practical implementation, an investigation based on parallel architectures and programming models will be conducted.

The elaboration is diveded into three different chapters: In the first chapter, the results of the general research are presented [see Chapter 3]. After that, the second chapter is pointing out the practical implemenation of the developed application on the ESP32 [see Chapter 4]. In the end, the results including the webforntend and the automatic benchmark setup [see Chapter 5] will be discussed.

[1]add. information: https://www.espressif.com/en/products/hardware/socs

Chapter 3

Parallel Programming in General

Since the 1970s [33], the decade in which the microprocessor era started, the overall performance of a processor has increased [14]. This goal was achieved by several points, including "*sophisticated process technology, innovative architecture or micro-architecture*" [see 14, Chapter 1, p2]. In fact, increasing the clock speed of a single core processor, like Moore's Law predicted [33], was usually reached by increasing the number of transistors on the chip [33]. However, this go along side with the increase in complexity [see 33, Pollack's rule], which mean, that doubling the logic of a processor result in a performance boost of only 40% [see 33, Chapter 2].

Another huge problem chip manufacturers have to deal with is leakage power [see 14, Chapter 2, p3], because the "*transistor leakage current increases as the chip size shrinks*" [see 33, p2] [see Chart 3.1]. An increase of leakage current of the transistors also result in a increase of the die's temperature [14] along side the total power consumption as well.

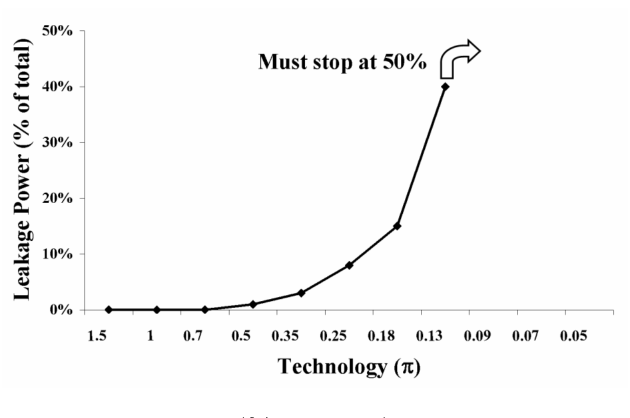


Figure 3.1: Leakage Power (% of total) vs. process technology [14]

Furthermore, a increase of the processor clock frequency to speed up the performance is only available to a suffisticating limit of 4GHz [33]. After this frequency threshold, also known as reaching the power wall, the "*power dissipation*" [see 33, p2] increases again.

Facing these types of problems such as "*chip fabrication costs, fault tolerance, power efficiency, heat dissipation*" [see 33, p3] along side with increasing processor performance, the only possible solution chip manufacturers and companies could offer was parallelism.

3.1 Basic Concept

Parallelism for processing is not something new. But due to the fact that real thread level parallelism [see Chapter 3.3.2] was only available after dual or multi-core processors were invented in 2005 [17], the topic itself and efficient software implementations are still treated in scientific work [3] [28].

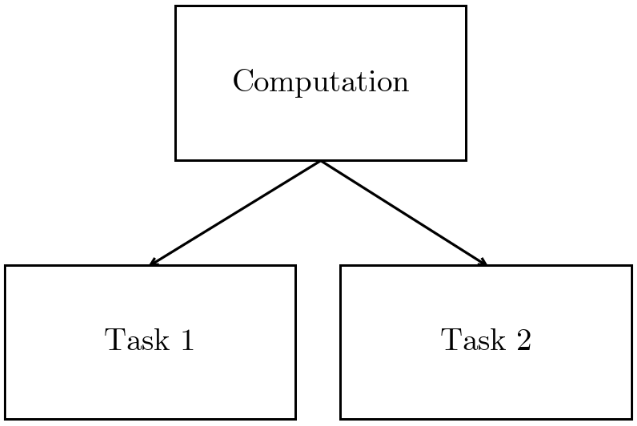


Figure 3.2: The basic concept of a simple concurrency computation.

In general, parallelism for programming means to split up a task or a computation into several sub tasks [e.g. Figure 3.2] or results, to decrease the execution time. Depending on the problem itself, these separated tasks can be independent or connected [see Chapter 3.1.3]. If we want to talk about the general concept of parallelism, we have to take a closer look to some mathematical laws, which try to describe the availability to parallel task execution and their limits.

3.1.1 Amdahl's Law

The first one, which quantifies parallelism, is called Amdahl's Law [8]. During the publication period of Amdahl's paper [6], critics claimed "*that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers*" [see 8, p80]. Of course this can be transferred on single and multi-core processors or even on multi threading [see 23, Chapter 1.3, p2], but in fact, like Amdahl claimed too, addressing hardware [8], and nowadays switching context time was not considered in this case.

Amdahl's Law wants "*to provide an upper limit on speedup*" [see 8, p81] in general to point out that there is a overhead [8], which can not pe implemented in parallel, but at the same time, "*apart from the sequential fraction, the remaining computations are perfectly parallelizable*" [see 8, p81].

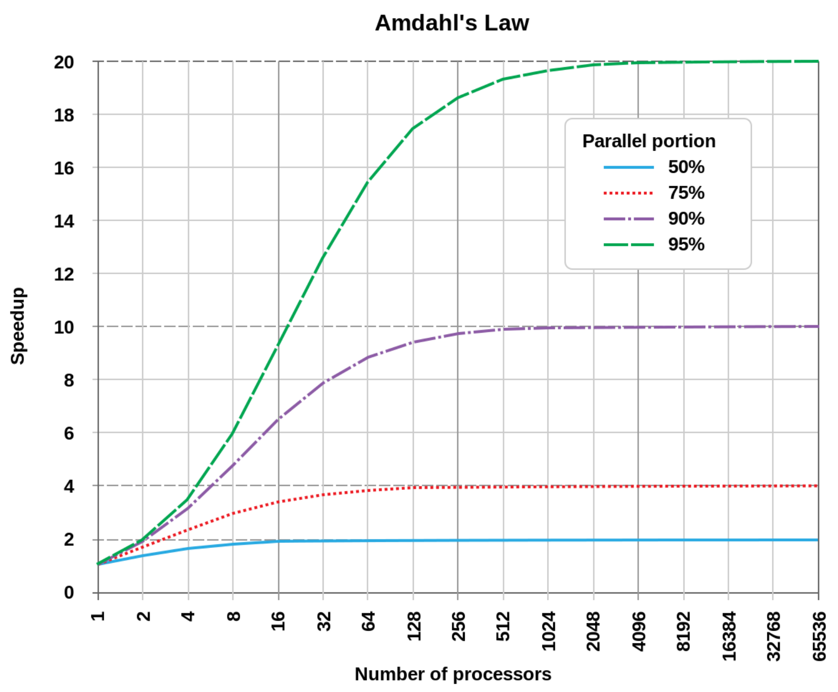


Figure 3.3: The limited speed-up of a program, which can be parallelized, depending on the number of parallel executions [13] [similar to 23, p4].

So regarding to Amdahl's Law, there has to be an upper limit of parallelism, due to the fact, that some sequential fractions still exists. In order to be able to describe this relationship, the time required to perform a calculation at once is related to the total time in parallel execution:

"*Let t_1 be the time taken by one processor solving a computational problem and t_p be the time taken by p processors solving the same problem. Finally let us denote the supposed inherently sequential fraction of instructions by f. Then, according to Amdahl, $t_p = t_1(f+(1-f)/p)$ and the speedup obtainable by p processors can be expressed as*" [see 8, p81]:

$$s = \frac{t_1}{t_p} = \frac{1}{f + (1-f)/p} \tag{3.1}$$

For example, a program which contains 90% of parallelizable code [see Figure 3.3] reaches his speed up limit at around 512 cores [formula 3.2]; in this case we substitute f= 0.2/2. After that number of processor cores, an significant speed up increase is not noticeable anymore .

$$\lim_{p\to\infty f\to 0.1}\left(\frac{t_1}{t_p}\right) = \lim_{p\to\infty}\left(\frac{1}{0.1 + (1-0.1)/p}\right) = 10 \tag{3.2}$$

Many other authors tried to claim that this upper limit of speed up, both in theory and practice, is not the final end. To proof that, only to mention a few , they took into account the "*energy per instruction (EPI)*" [Annavaram et al. in 8, Chapter 3, p81], a case study depending on "*asymmetric (or heterogeneous) chip multiprocessor architectures*" [Kumar et al. in 8, Chapter 3, p81] or even considering "*disk arrays to reduce input-output requirements*" [Patterson et al. in 8, Chapter 3, p81].

3.1.2 Gustafson's Law's

Due to the fact that Gustafson's Law's is based on "*the same concepts as the bases of Amdahl's law, it is more a variant, rather than a refutation*" [see 8, p81]. But in fact, it is another mathematical consideration, which offers, in comparison to Amdahl's Law, no upper speed up limit regarding parallelism. Related to Gustafson, the time a single core processor needs solving the same computational problem on the sequential would be *ft*$_p$, and on the parallelizable part *(1-f)pt*$_p$. Therefore, the total amount of achievable speed up by p processors can thus be calculated

$$s = \frac{t_1}{t_p} = \frac{ft_p + (1-f)pt_p}{t_p} = f + (1-f)p \tag{3.3}$$

using the formula 3.3 above. In this case, "*f is the same "inherently sequential" fraction of instructions as in the case of Amdahl's law*" [see 8, p81]. In addition to that, he doesn't take the "*sequential input-output requirements proportional to input and output sizes into account*" [see 8, p81].

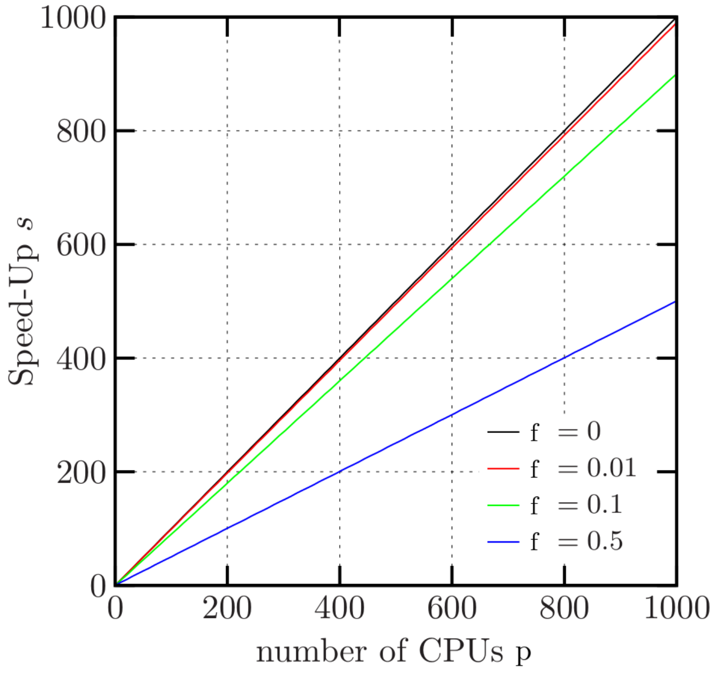


Figure 3.4: Gustafson's Law. In contrast to Amdahl we now have no upper limit to the speed up. f only determines the slope of the speed-up [18].

Regarding to [8], neither Amdahl's or Gustafson's Law are suitable to quantify parallelism in theory and practice, because both don't take into account, that "*sequential fractions of computations have negligible effect on speedup if the growth rate of the parallelizable fraction is higher than that of the sequential fraction*" [see 8, Chapter 7, p88].

Furthermore, [8] point out, that no simple formula governing parallelism exists. Both laws are more of an attempt to describe experimental results, and therefore understood rather as a draft rule of thumb as a law.

3.1.3 Principles of Parallel Computing

Despite trying to quantify the ability to parallelism task processing, it is also important to keep in mind the basic aim of parallelism to mention a "*design for concurrency*" [see 24, p4]. First of all, reducing the execution time is one of the most important objective in concurrency to make applications more **efficient**. Computations, and even tasks, are getting more and more complicated [9], not only in the application field of scientific researches. Today's software applications require sophisticated hardware like multi-core processors, to offer a suitable user experience, for example in gaming (e.g VR), augmented reality in automation (e.g. AR) or for IoT [see Chapter 1] devices.

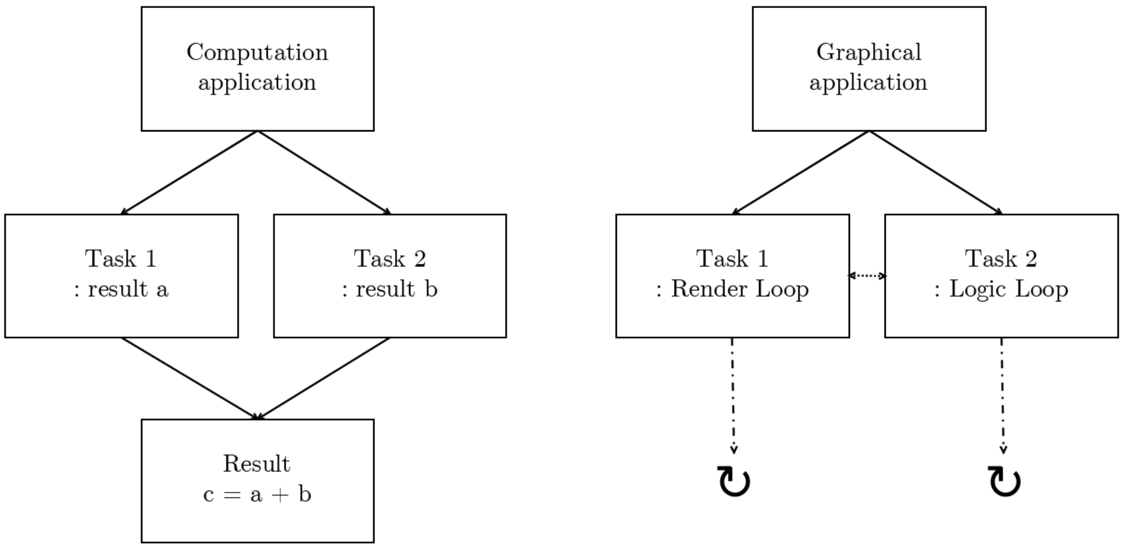


Figure 3.5: Comparison of independent and dependent tasks in a general concurrency application.

In case of software parallelism, which is the main objective of this chapter, we have to distinguish between tasks, which are independent and tasks, which are connected (called dependent). It depends largely on the application itself: e.g. for example, a numerical calculation [see Chapter 3.2] can be easily subdivided into sub tasks to speed up the computation, whereas a graphical application needs a logical loop and a render loop, both independently in different tasks [see Fig. 3.5]; a data exchange usually takes place via a shared memory [more on this in Chapter 3.3]. As already suggested in Chapter 3.2, our work will be limited to the division of numerical or generic mathematical calculations.

In addition to that, multi-core software applications offers also the opportunity, to provide low power systems, because **power consumption** results from performance and clock frequency [see 33, Fig. 3, p4], along side instructions per core (IPC) [14] [further inf. in Chapter 3].

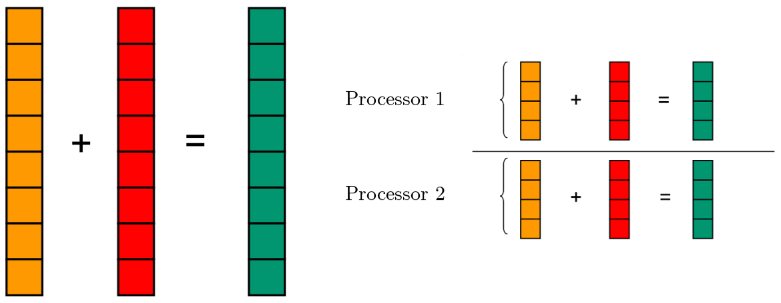


Figure 3.6: Adding to arrays of integers for speed up in concurrency [see 9, p3].

Furthermore, to guarantee **flexibility** for both, the software application and hardware platform, multi-core systems and concurrency is the way to go to fit customers and companies claims. This can easily be proofed due to the fact that today's hard- and software manufacturers always try to provide generalized solutions.

It can not be denied that as the degree of parallelism increases, the complexity of the application including the hardware realization also increases [33] [24]. A basic principle and design pattern of parallel computing is therefore to ensure maximum efficiency through parallelism while reducing complexity. A usual rule can be transferred on this topic: ensure **simplicity**.

Regarding to Figure 3.5 and Chapter 3.2, the best way to implement and computation concurrent is to guarantee that the sub tasks can work independent from each other. This has a major effect on the performance and complexity of the software implementation: "massively parallel vs. embarrassingly serial" [see 24, p15], alongside the ability to take less attention on control issues such as task orders, synchronization, (shared) memory access or even task communication [12]. In fact, complete concurrency is not possible, due to the fact that splitting a computation into sub tasks requires at least on remaining worker task to collect and combine the sub task results. Anyway, **Independence** of sub tasks enables almost the best efficiency.

Emphasises design [see 24, p5]: In summary, therefore, the following points should be seen as goals and thus as groundbreaking for the basic principles of parallelism:

1. Efficiency:

   Concurrency in hard- and software to solve large problems in less amount of time to reduce execution time and power consumption.

2. Flexibility:

   Environments will be more heterogeneous and the use in different application areas will be enabled.

3. Simplicity:

   Code for Parallel Computing will be more complicated, because synchronization or memory access have to be regulated. One reason more "*to strive for maintainable, understandable programs*" [see 24, p6].

4. Independence:

   To ensure maximum efficiency, parallel computations should be independent.

These principles result in four different **design patterns** [see 24, p11 ff.] for parallel software applications:

- Finding concurrency:

  This should be the main aim of all software applications today, set the case it makes sense.

- Algorithm structure:

  To ensure proper efficient, the implementation should be based on usual parallel programming models [see Chapter 3.4].

- Supporting structures:

  "*Useful idioms rather than unique implementations*" [see 24, p25] like Loop Parallelism, Fork/Join, Shared Data or Shared Queue [24].

- Implementation Mechanism:

  Using programming languages which offer the opportunity to real parallel computing (e.g. C++, OpenMP & Pthreads, MPI, OpenCL) [24].

3.2 Definition of parallel mathematical computations

We talked about how to quantifying parallelism, and above that, also about the principles of parallel computing and the resulting design patterns for parallel computing. Now its time to take into account the necessary mathematical computations. Not all calculations are suitable for parallelism, so we have to work out properties for the quantification of possible numerical or mathematical methods.

$$\vec{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_i \\ \vdots \\ a_{N-1} \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_i \\ \vdots \\ b_{N-1} \end{bmatrix} \tag{3.4}$$

The easiest way to do that is to take a look at two basic examples. The first one will be a simple scalar product of two N dimensional vectors $\vec{a}$ and $\vec{b}$ [see formula 3.4]. Both vectors have the same dimension N and are not orthogonal to each other; otherwise the scalar product would be very simple to calculate and the result null, regardless of the dimension of the vectors. The scalar product of two N dimensional vectors are defined as

$$s = \vec{a} \cdot \vec{b} = a_0 \cdot b_0 + a_1 \cdot b_1 + \quad \cdots \quad + a_{N-1} \cdot b_{N-1} \tag{3.5}$$

or more formally like

$$s = \sum_{i=0}^{N-1} a[i] \cdot b[i] \tag{3.6}$$

As mentioned in formula 3.5, theoretically any kind of scalar product can be easily calculated in parallel using N processor cores or tasks. We even don't have to take care about accessing resources, because all tasks will use their one values depending on the index elements of the vectors. Only in the end, the final result has to be calculated including the results, which are returned from the different tasks.

But to keep it simple, we will only separate the given scalar product [see formula 3.6] into to different tasks. A synonymous representation would be the following:

$$\begin{aligned} s &= \sum_{i=0}^{N/2-1} (a[i] \cdot b[i]) + \sum_{i=N/2}^{N-1} (a[i] \cdot b[i]) \\ &= \underbrace{\sum_{i=0}^{N/2-1} (a_{local}[i] \cdot b_{local}[i])}_{\mathrm{p_0}} \quad \underset{\substack{\uparrow \\ \text{process communication}}}{+} \quad \underbrace{\sum_{i=N/2}^{N-1} (a_{local}[i] \cdot b_{local}[i])}_{\mathrm{p_1}} \end{aligned} \tag{3.7}$$

The important sign in formula 3.7 is the "+". It seems inconspicuous, but in fact, this is the the one we have to take care about most. The "+" symbol indicates required "communication between the processes" [see 9, p] $\mathrm{p_0}$ and $\mathrm{p_1}$. So the essential part in every parallel computation will be collecting the sub results together. This could either be achieved by returning each result from the sub task to the main task or storing the sub result locally in memory, so it can be collected afterwards.

The complex theory would categorize this kind of computation as a decision problem in the *NC* class as a subset of *P* [26]. In general, all computations are stored in the *P* class of decisions. *NC* problems "*can be solved in parallel time (log n)$^{O(1)}$ using n$^{O(1)}$ processors*" [see 8, Chapter 10, p91]. In other words, *NC* problems can be solved in "*polylogarithmic time by using a polynomial number of processors*" [see 8, Chapter 10, p91]. Furthermore they are also known as fast parallel working algorithms [8] and in most of the cases as the most efficient ones [26].

In comparison to a almost perfectly independent parallel computations, the question arises, whether a "inherently sequential" problem exists. This type of computations, due to the complex theory, belong to the class of *P-complete* [8] decisions and are also known as difficult to parallelize effectively or being solved in limited space. Regarding to [8], no "real" *P-complete* problems exists, and considered to [26], sequential models are in fact parallelizable, but this comes "*at the cost of an enormous number of processors*" [see 26, Chapter 5.5, p69], resources or even execution time [see 26, p61].

So lets take a look at our second example, which might seems easily to compute, but in fact comes with a major bottleneck if we are trying to solve this kind of problem in parallel. The best known mathematical calculation to illustrate this is the prefix sum [25]. Let's say we have a vector with N elements like mentioned underneath [see formula 3.8]:

$$\vec{x} = \begin{pmatrix} x_0 & x_1 & \cdots & x_i & \cdots & x_{N-1} \end{pmatrix} \tag{3.8}$$

The prefix sum is defined as a partial sum of series of each the vector elements. A none formally expression is shown in the following formula [see 3.10].

$$\begin{aligned}
y_0 &= x_0 \\
y_1 &= x_0 + x_1 && = y_0 + x_1 \\
y_2 &= x_0 + x_1 + x_2 && = y_1 + x_2 \\
&\vdots \\
y_i &= y_{i-1} + x_i
\end{aligned} \tag{3.9}$$

It is obvious that the prefix sum can be easily calculated in sequential with the last equation in [3.10]. The resulting solution vector based on the input vector $\vec{x}$ [see formula 3.8] should be represented in the following notation:

$$\vec{y} = \begin{pmatrix} x_0 & x_0 + x_1 & \cdots & y_i & \cdots & y_{N-2} + x_{N-1} \end{pmatrix} \tag{3.10}$$

To major problem with this computation is the fact, that each part result is dependent on the result before. A parallel computation is possible, but not like the implementation mentioned in [3.7], because therefore no major decrease of execution time would be achieved. Indeed, there are some parallel solutions for the prefix sum, but with restrictions such as efficiency, span [16] or less parallelism [19].

If we want to quantify parallel mathematical computation models, based on the examples which we worked out, the following three points should be taken into account:

1. Divisibility of the calculation.
2. Independence of the sub task calculations.
3. Involvement of achievable work efficiency.

3.3 Parallel Computer Architecture

The terminology of computer architecture was invented in 1960s by the designers of IBM System to describe the structure of a computer. Computer architect's task is to write an suitable program code for the machine, keeping in mind every time this structure of computer, understanding all the factors like state-of-the-art technologies at each design level and changing those designs tradeoffs for their specific applications [35].

Parallel computing means the situation where tasks are separated into discrete parts that can be executed concurrently. Each part is diffused into a larger series of instructions which will be executed simultaneously on different CPU's or even in a pipeline [see Figure 3.7]. These kind of parallel systems have to deal with the simultaneous use of multiple computer resources that can include either a single computer with multiple CPU's, or a number of computers connected by a network creating a parallel processing cluster or combination of both.

The crux of parallel processing are CPU's. Based on the number of instruction and data streams that can be processed simultaneously, computing systems are classified into four major categories based on Flynn's Taxonomy [27].

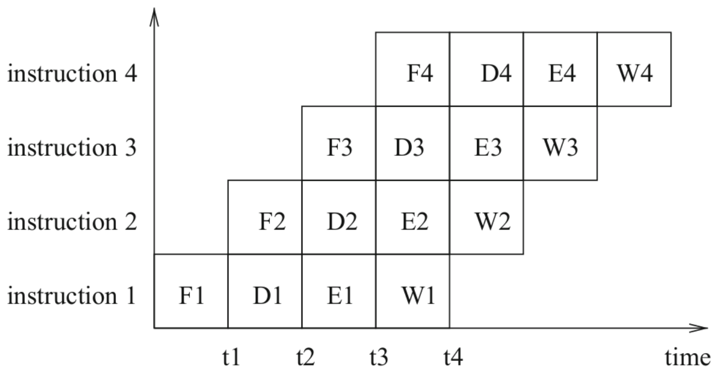


Figure 3.7: Overlapping execution of four independent instructions by pipelining. The execution of each instruction is split into four stages: *fetch (F)*, *decode (D)*, *execute (E)*, and *write back (W)* [see 31, Fig. 2.1, p11].

3.3.1 Flynn's Taxonomy of Parallel Architectures

Flynn's classification was first elaborated and proposed by Michael Flynn in 1966 and represents a scheme which is based on the notion of information stream. The term 'stream' defines a sequence or flow of either one of both existent types of information which flows and are operated into a processor: instructions or data.

Instruction stream defines the sequence of instructions performed by CPU, as in the same time the data stream defines the data traffic exchanged between the memory and CPU.His taxonomy left aside the machine's structure for classification of parallel computers and took over a whole new concept focusing on multiplicity of instructions and data streams observed by the CPU during execution.

The major four categories are the followings [comp. to Fig. 3.8] [31]:

1. **SISD** (single-instruction, single-data) systems:

   It designs an sequential computer which exploits no parallelism in either the instruction stream nor data stream. An SISD computing system is a uniprocessor machine capable of single stream executions.

2. **SIMD** (single-instruction, multiple-data) systems:

   It designs a multiprocessor machine capable of executing a single instruction stream on multiple different data streams. Instructions can be executed sequentially, such as by pipe-lining, or in parallel by multiple functional units.

3. **MISD** (multiple instruction streams, single data stream) systems:

   It designs a multiprocessor machine capable of executing different instructions streams on the same data stream.

4. **MIMD** (multiple instruction streams, multiple data streams) systems:

   It designs a multiprocessor machine capable of executing multiple instructions streams on multiple data streams. This architectures include multi-core superscalar processors and distributed systems.

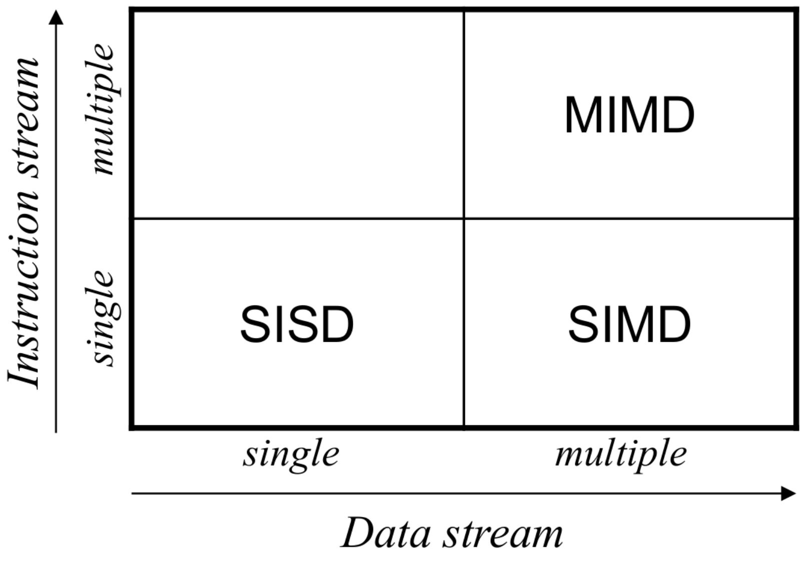


Figure 3.8: Visualization of Flynn's Taxonomy [see 12, p5].

3.3.2 Types of Parallelism

Parallel computing is used for multiple processing elements simultaneously to solve any type of problem, as it's already explained in [3.3].

The Parallel Computing is evolved from serial computing that attempts to emulate what has always been the state of affairs in natural World. Parallel Computing saves in comparison to Serial Computing time and money as many resources working together will reduce the time and cut potential costs. Furthermore it can be impractical to solve larger problems on Serial Computing.

Beside the advantages for parallel computing, the different types of Parallelism are listed as the followings:

1. **Bit-level parallelism**: This form of parallelism computing have it's roots based in the concept of increasing the processor's size. The amount of instructons it's reduced that the system must execute in order to perform a task on large-sized data.

2. **Instruction-level parallelism**: This form of parallelism computing is designed for a processor to only address less than one instruction for each clock cycle phase. The instructions can be re-ordered and grouped which are later on executed concurrently without affecting the result of the program.

3. **Task/Thread parallelism**: This form of parallelism computing is designed to employ the sub tasks fragments of a decompositioned task and then allocate the fragments sub tasks for execution concurrently [15].

Thread-level parallelism is the only way to execute independent programs or discrete parts of a single program, simultaneously, using different sources of execution, like multiple processors sharing code, data and most of their address space, which are called threads.

In these days the term thread is often used in a casual way to refer to multiple executions that may run on different processors, even if they don't share an address space. To take advantage of an MIMD multiprocessor with n processors, we must usually have at least n threads or processes to execute. The independent threads are typically identified by the programmer or created by the compiler. Since the parallelism in this situation is contained in the threads, it is called thread-level parallelism. This corresponding architectural organization is also called chip multiprocessing (CMP). An example for CMP is the placement of multiple independent execution cores with all execution resources onto a single processor chip. The resulting processors are called multi-core processors.

An alternative approach is the use of multi-threading to execute multiple threads simultaneously ona single processor by switching between the different threads when needed by the hardware [7].

3.4 Parallel Programming Models

A parallel programming model is defined as a set of program abstractions which helps application's parallel activities to fit to the underlying parallel hardware, spanning over layers like programming languages, compilers, libraries, network communications and input/output systems.

3.4.1 Classification of Parallel Programming Models

In general, models for parallel programming are distinguished according to their abstraction levels [31]. Therefore, the models are basically divided into four different classes:

1. Machine model:

   This model describes the lowest level of abstraction, in which the hardware description, the operating system, registers or input and output buffers are pointed out [see 31, p105].

2. Architectural models:

   The next level of abstraction are the architectural models. In this kind of model, the "*interconnection network of parallel platforms, the memory organization, the synchronous or asynchronous processing, or the execution mode of single instructions by SIMD or MIMD*" [see 31, p105-106] are described.

3. Computational model:

   The model of computation is based on cost functions and a more formal model of a "*corresponding architectural model*" [see 31, p106]. It also takes execution time along side with the providing of resources regarding to an architectural model into account [31]. Analytical methods for designing and evaluating algorithms are also part of this model to quantify proper computations as we already mentioned in Chapter 3.2.

4. Programming model:

   The highest level of abstraction is the programming model. A description here is based on "*the semantics of the programming language or programming environment*" [see 31, p106] to specify parallel programs. Therefore, a computation is mainly separated into "*(i) a sequence of instructions performing arithmetic or logical operations, (ii) a sequence of statements where each statement may capture several instructions, or (iii) a function or method invocation which typically consists of several statements*" [see 31, p106].

Based on the different types of models, two major group of model classification should also be discussed.

3.4.1.1 Process Interaction

Non independent parallel computations need synchronization and communication to exchange data or for notification purposes. The model of process interaction wants to provide possibilities to deal with that kind of problem [see 12, p4]. In general, we distinguish between three different process interaction models: the first one is called Shared address space programming (shared memory) model [12]. In the **shared memory** programming model, threads share a common address space, which they read and write in an asynchronous manner. The concept represents a semaphore or lock that is used for synchronization, when the situation that more than one thread accesses the same data variable, the semaphore keeps data local to the processor and makes private copies avoiding expensive memory accesses. In the case of multiple processors sharing the same data with writing possibility, it needs some mechanisms of coherence maintenance.

In the **message passing** programming model, threads used to communicate via message exchange having private memories. For message exchanging, each sends operation needs to have a corresponding receive operation. Threads are not constrained to exist on te same physical machine.

A suitable mixture of these two models is appropriate. Processors can't directly access memory from another processor. This is achieved via message passing, but what the programmer actually sees is shared-memory model [5].

Performing parallelism is also possible by compiler. In this case, not the programmer is responsible for the implementation itself. Regarding to the used programming language or the provided functions, parallelism will be achieved by an intelligent compiling process. This model is also known as Implicit Interaction [12] [11].

3.4.1.2 Problem decomposition

In comparison to process interaction, problem decomposition provides independent parallel execution. Generally, three different types of models have to be discussed. First of all, as we already mentioned in Chapter 3.3.2, task parallelism describes the basic method of threads, which contains assets of instructions or data. In addition to that, Flynn's Taxonomy usually classified this type of model as MIMD/MPMD or MISD [see Chapter 3.3.1] [11].

In contrast to that, data parallelism focus on processing different sections of data, e.g. an array of elements [see Chapter 3.2]. Processing the data will be independent, so therefore no race conditions appear. Flynn's Taxonomy classified this type of model as MIMD/SPMD or SIMD [see Chapter 3.3.1] [11].

Implicit parallelism is comparable to Implicit Interaction. The compiler and/or the hardware during runtime is responsible for parallelism. This can be achieved by translating serial into parallel code or by instruction-level parallelism [see Chapter 3.3.2] [11].

Chapter 4

Project documentation

4.1 Simple mathematical computation example for Parallel Programming

Before going into detail about the Mandelbrot computation, we will discuss as a first example a basic computation as already mentioned in Chapter 3.2. The reason we are doing that is to reduce complexity and to point out the main parts, we will need for the Mandelbrot computation as well. In formula 3.7, we have pointed out that the computation of a simple scalar product of two vectors with the same dimension N can be split into several sub task p. Our modified version is based on two sums. The outer one will iterate [index i, see formula 4.1] from zero to an upper limit N_1 and sum up the result from the inner sum, which will perform a sum from 0 to N_2 [index j, see formula 4.1] over a multiplication of i and j. The following formula represent the announced mathematical problem:

$$s = \sum_{i=0}^{N_1} \left(\sum_{j=0}^{N_2} (i \cdot j) \right) \tag{4.1}$$

To process the problem in parallel, it is only possible to seperate the outer sum into sub sums regarding to formula 4.2.

$$\begin{aligned} s &= \sum_{i=0}^{N_1} \left(\sum_{j=0}^{N_2} (i \cdot j) \right) \\ &= \underbrace{\sum_{i=0}^{\frac{N_1}{2}-1} \left(\sum_{j=0}^{N_2} (i \cdot j) \right)}_{\mathrm{p_0}} \quad \underset{\substack{\uparrow \\ \text{process communication}}}{+} \quad \underbrace{\sum_{i=\frac{N_1}{2}}^{N_1} \left(\sum_{j=0}^{N_2} (i \cdot j) \right)}_{\mathrm{p_1}} \end{aligned} \tag{4.2}$$

For collecting the part results from the parallel computed sums, some sort of process communication [see formula 4.2] between the threads is necessary, more then, its impossible without it.

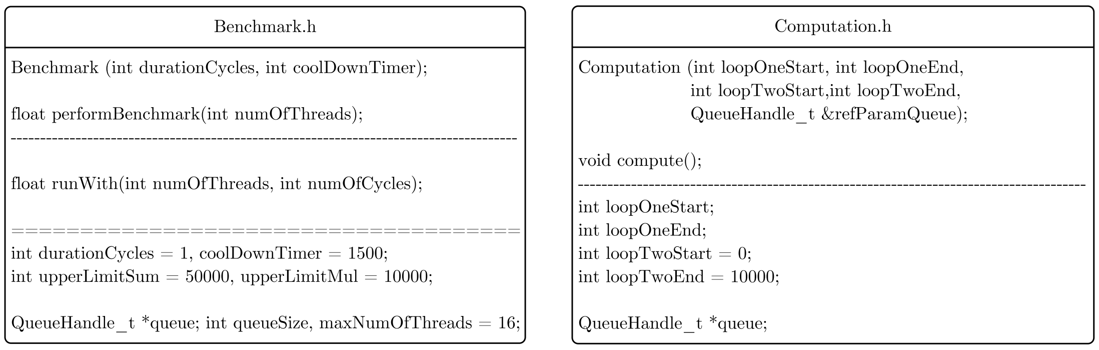
But actually, this is the task, we have to take care about most. Otherwise we will run into race conditions while addressing same resources or even blocking methods, which will result in wrong execution time measurements.

As mentioned in Chapter 3.4, a common solution to communicate between threads running on different cores is using the **Message Passing Model** [for more details, see Chapter 3.4.1.1]. Our aim is to pass the computed part results to a main "actor", which will collect the part sums to calculate the final result. Furthermore, for bench-marking this setup, we will implement a time measurement mechanism to evaluate the decrease of execution time running the computation on one or both cores.

Formula 4.2 is theoretically not limited to the dedicated number of cores, which the hardware platform offers. Each core can handle multiple threads, but in this case, we would leave real time parallelism, because each core now has to handle multiple threads by the scheduler. Every thread on each core will be given some execution time between switching the context to the next one. An overall speedup is not guaranteed with this method.

4.1.1 UML diagram

This little mathematical example can be found on our GitHub repository [22] and is basically split into three parts. The main **.ino* file, and two classes, the *Computation.h* and the *Benchmark.h*.



```
Benchmark.h
-------------------------------------------------------------------------------------
Benchmark (int durationCycles, int coolDownTimer);

float performBenchmark(int numOfThreads);
-------------------------------------------------------------------------------------

float runWith(int numOfThreads, int numOfCycles);

=====================================
int durationCycles = 1, coolDownTimer = 1500;
int upperLimitSum = 50000, upperLimitMul = 10000;

QueueHandle_t *queue; int queueSize, maxNumOfThreads = 16;
```

```
Computation.h
-------------------------------------------------------------------------------------
Computation (int loopOneStart, int loopOneEnd,
             int loopTwoStart,int loopTwoEnd,
             QueueHandle_t &refParamQueue);

void compute();
-------------------------------------------------------------------------------------
int loopOneStart;
int loopOneEnd;
int loopTwoStart = 0;
int loopTwoEnd = 10000;

QueueHandle_t *queue;
```

(a) UML diagram Benchmark.h class (b) UML diagram Computation.h class

Figure 4.1: Overview over both main implementations regarding Chapter 4.1

4.1.2 Implementation

The *Benchmark.h* class has several functions to set the necessary parameters to start a benchmark like mentioned at the end of Chapter 4.1. The computational problem discussed in formula 4.2 is implemented in *Computation.h* class regarding to the following function.

```
void Computation::compute() {
    long count = 0;

    for (int i = getStartOne(); i < getEndOne(); i++) {
        for (int j = getStartTwo(); j < getEndTwo(); j++) {
            count += i * j;
        }
    }

    /* process communication */

    xQueueSend(*queue, &count, portMAX_DELAY);
}
```

It is obvious, based on simple mathematical rules, that the outer sum, which is implemented through the outer for-loop, is divisible by limiting the thresholds, the inner sum unfortunately isn't. So for each part sum it's necessary to calculate the limits, on which each task has to compute their part sum results [see 22, Benchmark.h, line 94 ff.].

To start the benchmark, we have to include the *Benchmark.h* header file, and create an object of it. In the constructor, it is necessary to specify the amount of duration's per cycle (for averaging the result), the time to wait until the next benchmark execution will be started and the queue size.

```
Benchmark bench(1, 1500, 8);

for (int i = 0; i < numOfRunningThreads; ++i) {
    results[i] = new float(bench.performBenchmark(i + 1));
}
```

The queue size represents the maximum number of threads, which can be executed in parallel, because each thread has to return the result to the main thread; this is done by a ESP32 specific *QueueHandle_t* message passing model between tasks. Keep in mind, that this has an major effect of the available stack or heap size, and of course on the maximum amount of created threads [for more information about the *QueueHandle_t* thread communication, see *Benchmark.h* line 99 ff.].

So let's become more concretely: We start for our outer for-loop [see formula 4.2] with i from 0 to N_1 equals 50000. Our inner for-loop has it's limit at $N_2 = 10000$. If we now want to split this computation into four separated tasks, two on core 0 (t_1, t_2) and two on core 1 (t_3, t_4). The new limits for the part sums can be calculated using the formula below:

```
float Benchmark::runWith(int numOfThreads) {
    ...

    for (int k = 0; k < numOfThreads; k++) {

        /* outer sum i = 0 ... N1 */
        int lowLimSum = (50000 / numOfThreads) * k;
        int uppLimSum = (50000 / numOfThreads) * (k + 1);

        /* inner sum j = 0 ... N2 */
        int uppLimMul = 10000;

        if(50000 % numOfThreads != 0 && k == numOfThreads - 1) {
            uppLimSum += (50000 % numOfThreads);
        }

        c[k] = new Computation( lowLimSum, uppLimSum,
                                0, uppLimMul, *queue );

        ...
    }
}
```

Our new limits for the part sums are now the ones mentioned in formula 4.3 and are summed up in table 4.1.

$$
\begin{aligned}
s_a &= \sum_{i=0}^{12499}\left(\sum_{j=0}^{10000}(i\cdot j)\right) \quad , s_b = \sum_{i=12500}^{24999}\left(\sum_{j=0}^{10000}(i\cdot j)\right)\\
s_c &= \sum_{i=25000}^{37499}\left(\sum_{j=0}^{10000}(i\cdot j)\right) \quad , s_d = \sum_{i=37500}^{50000}\left(\sum_{j=0}^{10000}(i\cdot j)\right)
\end{aligned}
\tag{4.3}
$$

The implementation of the part sum limits is a little bit different to the mathematical description regarding formula 4.3, because we are using for-loops and to determine the end of the sum, we are using a "*less than*" operator. So therefore the threshold limits can be found in the table below.

Part sum limits			
Thread t_1 on core p_0	Thread t_2 on core p_1	Thread t_3 on core p_0	Thread t_4 on core p_1
i = 0 ⋮ i <12500	i = 12500 ⋮ i <25000	i = 25000 ⋮ i <37500	i = 37500 ⋮ i <50000

Table 4.1: Overview over the separated for-loop sum limits for four threads

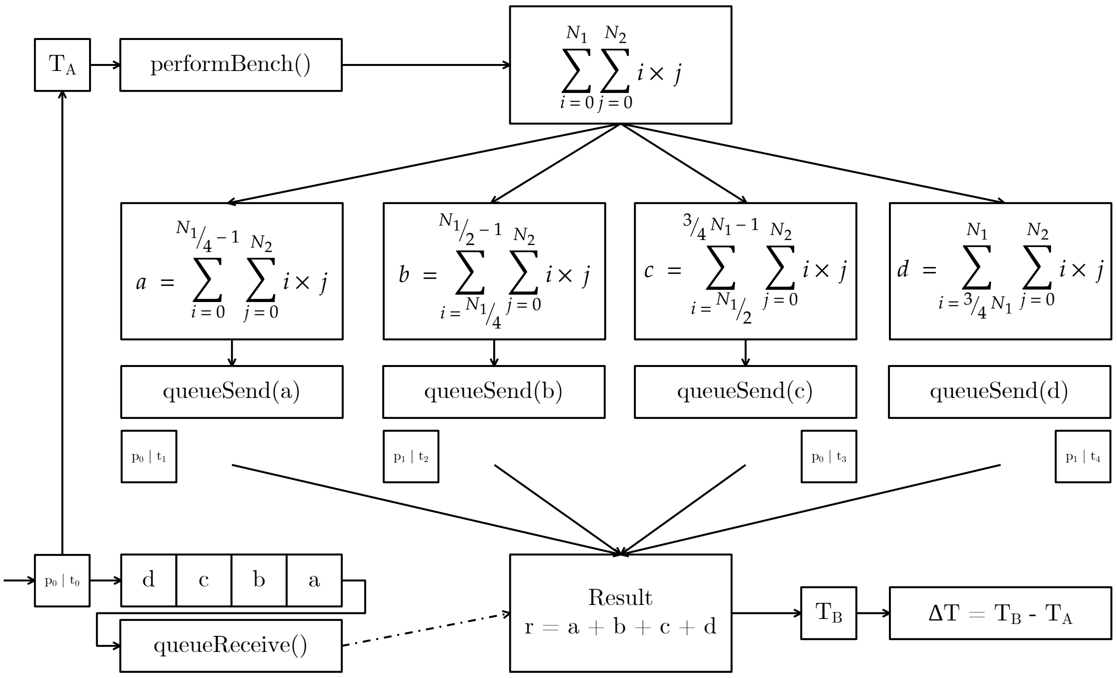
But what happened if we have an odd number of threads? Well in this case, the *Benchmark.h* class handle the issue, too. The upper limit will be divided into even part sum limits and for the last thread, the upper limit consists of the even upper limit plus the rest until he reaches 50000. This is done by an if clause and a "*modulo*" operator [see 4.1.2]. After determine the necessary limits and amount of threads, we are now able to create the tasks and assign them to a dedicated core. As mentioned before, real time parallelism can only be achieved by using the exact or less number of cores.

```
for (int j = 0; j < numOfThreads; j++) {
    ...

    c[j] = new Computation( lowLimSum, uppLimSum,
                            0, uppLimMul, *queue );

    xTaskCreatePinnedToCore(
        producerTask,
        (j+1)%2 == 0 ? "calcTask0" : "calcTask1",
        sizeof(c[j]) * 32 * 8,
        c[j],
        0,
        NULL,
        (j+1)%2 );
}
```

4.1.3 Implementation of the Message Passing Model



T_A
performBench()
$\sum_{i=0}^{N_1}\sum_{j=0}^{N_2} i\times j$
$a = \sum_{i=0}^{N_1/4-1}\sum_{j=0}^{N_2} i\times j$
$b = \sum_{i=N_1/4}^{N_1/2-1}\sum_{j=0}^{N_2} i\times j$
$c = \sum_{i=N_1/2}^{3/4 N_1-1}\sum_{j=0}^{N_2} i\times j$
$d = \sum_{i=3/4 N_1}^{N_1}\sum_{j=0}^{N_2} i\times j$
queueSend(a)
queueSend(b)
queueSend(c)
queueSend(d)
$p_0 \mid t_1$
$p_1 \mid t_2$
$p_0 \mid t_3$
$p_1 \mid t_4$
$p_0 \mid t_0$
d
c
b
a
queueReceive()
Result
r = a + b + c + d
T_B
$\Delta T = T_B - T_A$

Figure 4.2: Overview of the benchmark setup for the Computation example

If we now want to compare results regarding running the Computation on different cores, or even with multiple threads, we have to implement a time measurement functionality. This is done by the *Benchmark.h* class. Before starting a Computation, we are saving a time stamp, and after all part results are collected, we take another one. Basically the time needed performing the tasks is the delta time between those two time stamps. But this lead into one main problem: How can we determine we collected all results, because this is an asynchronous event.

Now our *QueueHandle_t* object will help once again. If we send back our results, the receiver will wait as long as a new object is pushed into the queue. As far as we know, we have to receive the passed *numOfThreads* amount of part results. Combining these both examples will result into the following implementation:

```
for (int k = 0; k < numOfThreads; k++) {
    long partResult;

    /* process communication -> receiving part results */
    xQueueReceive(*queue, &partResult, portMAX_DELAY);

    result += partResult;
}
```

The last parameter for the *xQueueReceive* function defines how long the receiver should wait until he should give up waiting. This is now set to a macro defining to wait as long as possible or until the queue is full. The passed reference will be used to add it to the result sum.

4.2 Introduction to the Mandelbrot fractal

The Mandelbrot set is generated by iterations, which means to repeat a process over and over again. In mathematics this process is most often the application of a mathematical function. For the Mandelbrot set, the functions involved are some of the simplest imaginable: they all are what is called quadratic polynomials and have the form

$$f(z) = z^2 + c \tag{4.4}$$

or even described with

$$M = \{ \quad c \in C \quad | \quad lim_{n \to \infty} Z_n \neq \infty \quad \} \tag{4.5}$$

where c is a constant number and M the complex series containing the values which are in the Mandelbrot set. Our main boundaries for the Mandelbrot set can be calculated as well, but we won't discuss this here. Let's say we take them for granted as shown in Figure 4.3. So what we have to specify now is the complex value c in formula 4.4.

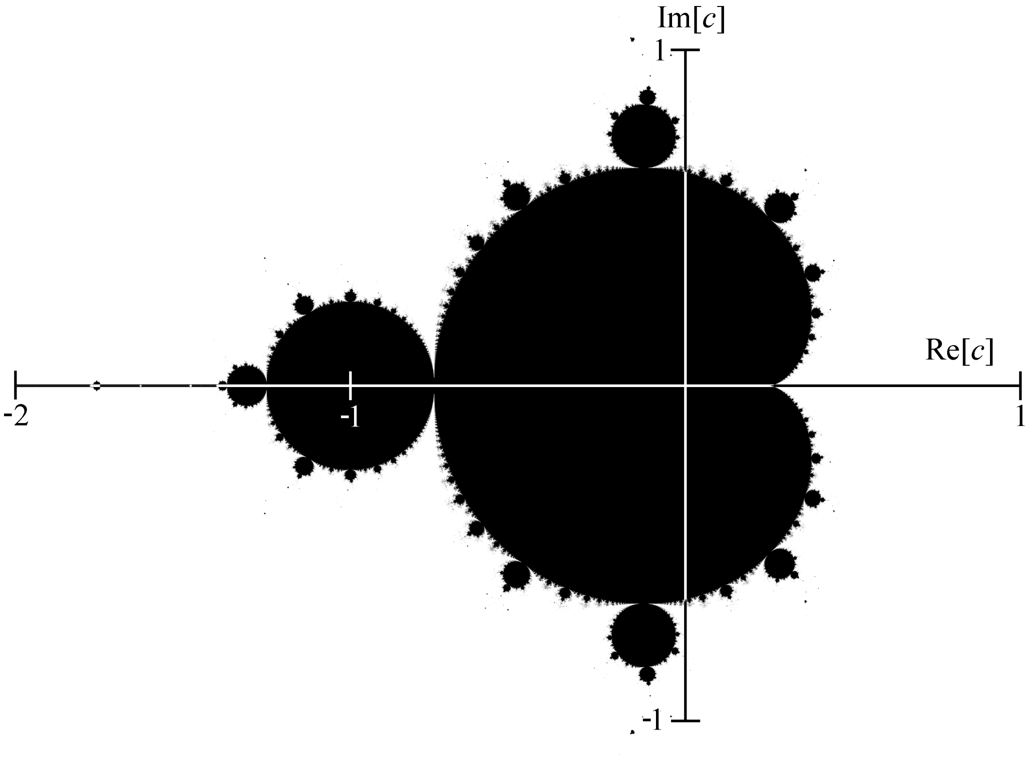


Figure 4.3: A mathematician's depiction of the Mandelbrot set M [10].

If we want to distinguish whether a point belongs to the Mandelbrot or not, we have to go through the mathematical description mentioned in formula 4.5. We start with a chosen complex value c_0 between our predefined boundaries, and pass this to formula 4.4 as c. The first result Z_0 will be now the passed value c ($f(0) = c$), and can be recursively passed as z again into the function to compute Z_1. Formula 4.4 can therefore be also described as follow:

$$\begin{aligned} Z_0 &= c, \\ Z_{n+1} &= Z_n^2 + c \\ &\vdots \end{aligned} \tag{4.6}$$

If Z_n, for c as our complex value under test, now goes to ∞, c doesn't belong to the Mandelbrot set M, if not, this is a point of our complex series.

The definition [see formula 4.6] in combination with formula 4.5 brings up two questions: Is it even possible to check whether the predefined complex value c passed to the Mandelbrot set function goes to ∞, especially on a micro controller? And if we are able to reach ∞ somehow, our computed value will be very big, so how can we deal with that on limited hardware?

These questions are more or less related, and not very complicated to solve. If we take a closer look into the general definition of a complex value, it can be proofed that "*if the absolute value of Z ever gets bigger than 2, it will never return to a place closer than 2, but it will actually rapidly escape to infinity*" [32]. In general that means we only have to check if the absolute value of our current Z_n goes bigger than two and therefore both problems are solved. But how often do we have to iterate through, to become more precisely, what is our threshold for n in formula 4.5?

Well, this can easily be shown if we calculate the Mandelbrot set for different maximal iterations for n [see Figure 4.4, n starting from 0 to 40 in steps by 10].

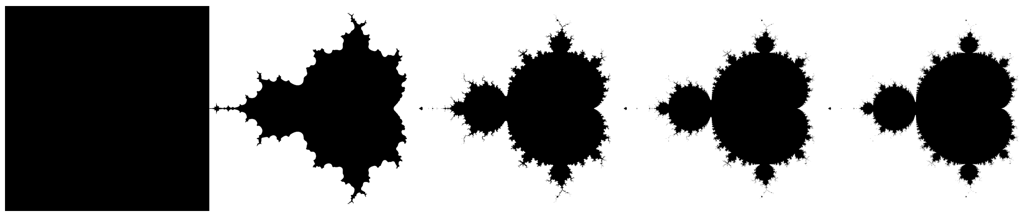


Figure 4.4: Mandelbrot Set using different numbers of iterations.

It is obvious, based on Figure 4.4, that the number of iterations we go through during checking if the absolute value of the complex number under test is less than two, regarding to the image resolution, has no major effect after more than 30 iterations. The image we are calculating is "*not infinitely accurate*", due to the fact that we are using pixel to plot the complex values. A considerable change in the image after a certain number of iterations [32] is not visible, therefore a larger number of iterations won't give us "*anything new to the image*" [32].

4.2.1 The Mandelbrot set implementation

As we already know, the Mandelbrot set M [see formula 4.5] can be calculated defining our main Cartesian complex boundaries while looping through all the points and test whether the complex value c belongs to the Mandelbrot set M or not. So basically our computation function consists of a part defining our complex boundaries

```
    /* Cartesian 2D boundaries */
    double MinRe = -1.2;
    double MaxRe = 1.2;
    double MinIm = -1.2;
    double MaxIm = 1.2;

    /* to get the complex value of a pixel point P(x|y) */
    double Re_facor = (MaxRe - MinRe) / (width - 1);
    double Im_factor = (MaxIm - MinIm) / (height - 1);

    /* implementation of the mathematical limes */
    /* -> to infinite (in this case 34) */
    unsigned MaxIterations = 34;
```

and how many iterations we want to go through, that meas how precisely we want to plot the image. After that, we have two for-loops going through all the pixel coordinates, which each has a corresponding complex value, to check whether they are in the Mandelbrot set or not [32]:

```
for (unsigned y = minY; y < maxY; ++y) {
    double c_im = MaxIm - y * Im_factor;

    for (unsigned x = minX; x < maxX; ++x) {
        double c_re = MinRe + x * Re_facor;

        double Z_re = c_re, Z_im = c_im;
        bool isInside = true;

        /* check if complex number Z belongs to M */
        for (unsigned n = 0; n < MaxIterations; ++n) {
            double Z_re2 = Z_re * Z_re, Z_im2 = Z_im * Z_im;

            if (Z_re2 + Z_im2 > 4) {
                isInside = false;
                break;
            }

            Z_im = 2 * Z_re * Z_im + c_im;
            Z_re = Z_re2 - Z_im2 + c_re;
        }

        /* draw pixel */
        if(isInside) { putpixel(x, y); }
    }
}
```

In this case, we used a lot of simplifications to avoid complex mathematical functions, which require including specific libraries we probably won't have on our *ESP32*. On of those are for example computing the absolute value of a complex number [see line 12, 4.2.1]:

$$\begin{aligned} abs(Z) &= \sqrt{Re\left\{Z\right\}^2 + Im\left\{Z\right\}^2} > 2 \\ Z^2 &= Re\left\{Z\right\}^2 + Im\left\{Z\right\}^2 > 4 \end{aligned} \tag{4.7}$$

After we found a complex value which fulfills the above condition *M*, we have to compute the next value regarding formula 4.4. The addition of two complex numbers is quite simple. However, complex number multiplication like z^2 in comparison to the addition is not that easy. But this process can be simplified as well.

$$\begin{aligned} Z_{n+1}^2 &= \left(Re\left\{Z\right\} + i * Im\left\{Z\right\}\right)^2 \\ &= \left(Re\left\{Z\right\} + i * Im\left\{Z\right\}\right)\left(Re\left\{Z\right\} + i * Im\left\{Z\right\}\right) \\ &= Re^2\left\{Z\right\} + i * Re\left\{Z\right\} Im\left\{Z\right\} + i * Re\left\{Z\right\} Im\left\{Z\right\} + \left(i * Im\left\{Z\right\}\right)^2 \\ &= Re^2\left\{Z\right\} - Im^2\left\{Z\right\} + 2i * Re\left\{Z\right\} Im\left\{Z\right\} \end{aligned} \tag{4.8}$$

One can conclude from this:

$$\begin{aligned} Re\left\{Z_{n+1}^2\right\} &= Re^2\left\{Z\right\} - Im^2\left\{Z\right\} \\ Im\left\{Z_{n+1}^2\right\} &= 2i * Re\left\{Z\right\} Im\left\{Z\right\} \end{aligned} \tag{4.9}$$

The corresponding implementation can be found in line 19 and 20 [see 4.2.1]. So far we discussed how to compute whether a complex point belongs to the Mandelbrot set or not. What is missing right now is how to combine that with calculating the Mandelbrot in parallel and how we can create a real printable image out of our computation as well as the whole benchmark of this setup.

4.3 Parallel Processing the Mandelbrot fractal

Basically, there are no huge differences between the *Benchmark* we used for the *Computation* and the one we need right now for performing the Mandelbrot. What changed are the class properties and instead of a *Computation.h* class we have to include the *Mandelbrot.h* class. For a graphical visualization we will use instead of the uart interface a webfronted hosted on the *ESP32* itself, so after the *Benchmark* is done, the micro controller will create a wireless network, on which the user can log into to call the *frontend* in the web browser.

4.3.1 Mandelbrot State Machine

To handle all the different tasks and stages the *ESP32* has to go through, we implemented a state machine.

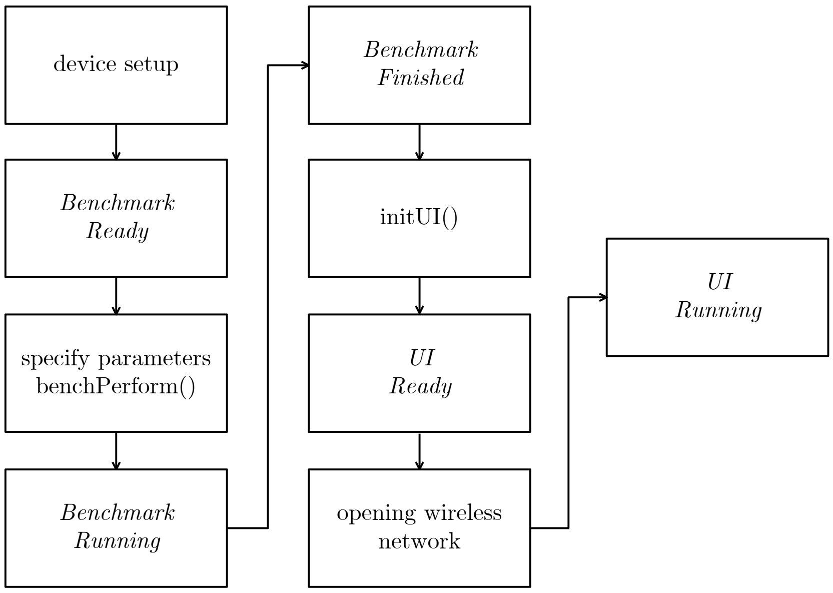


Figure 4.5: State machine implementation for handling the UI, Benchmark stages

Each state has its own set of parameters and will only switching the context to the next one if everything went fine. This is for preventing user errors by passing wrong arguments and to reduce runtime crashes. Nevertheless its also necessary because the UI needs several running threads in th background as well and this would lower the overall performance of the *ESP32*, which we actually need for the Benchmark. So the UI is only allow to start after the results are calculated and saved correctly.

The next chapter will now focus on the details of the C++ implementation of the *Benchmark* for performing the Mandelbrot computation [see Chapter 4.3.2]. Beside that, we will describe the *frontend* written in *javascript*, especially the communication and data processing part [see Chapter 4.3.5].

4.3.2 Mechanism to divide the picture into sub areas

If now want to compute the Mandelbrot set, as we already describe in Chapter 4.2, there are two possibilities to achieve this aim. And it definitely depends on the degree of complexity, which one we will choose. But first, let us describe the two main ways on how to compute the Mandelbrot set in parallel.

Our aim is to split the image into sub sections, as we did the same for the part sums computation. What probably comes first in our mind is to split the Mandelbrot image using a division like a chess board [see Figure 4.6].

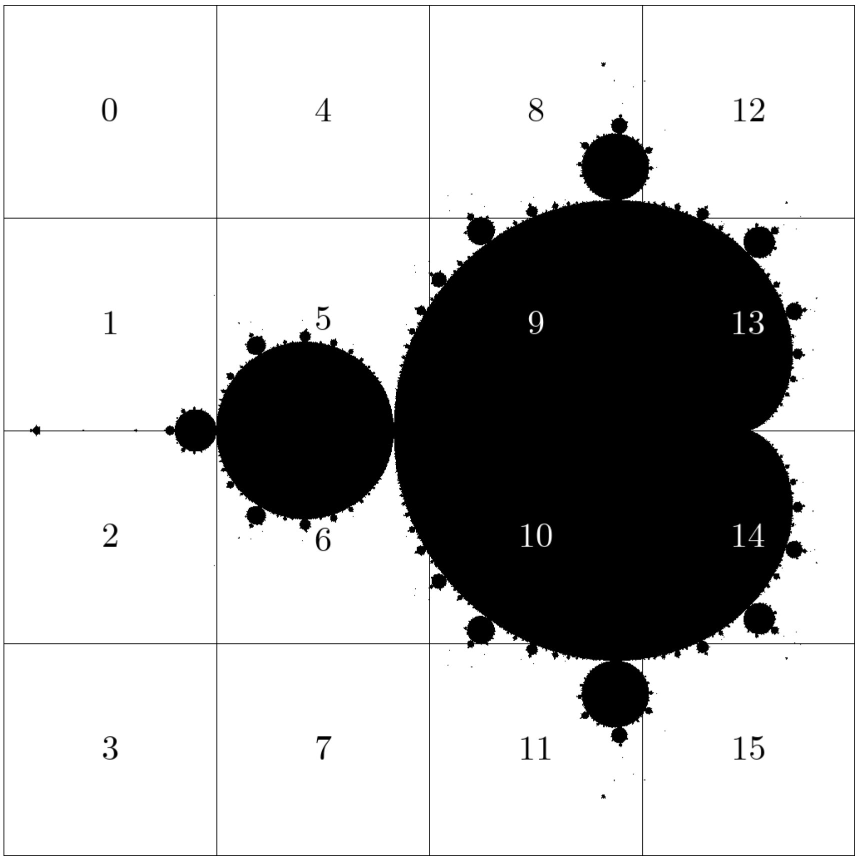


Figure 4.6: Split the Mandelbrot image using the chess board method.

Each sub image is defined based on its start and end point from where its located. If we assume a resolution of 600 by 600 pixel, sub image number 5 would have its start point at $P_{5_{\text{Start}}}$ (150, 150) and its end point at $P_{5_{\text{End}}}$ (300, 300). Our Mandelbrot function, which has to calculate the Mandelbrot set, should receive beside the main image resolution the boundaries of the sub areas, which have to be calculated dynamically as well, so in this case the user has only to specify the number of part images, which corresponds to the amount of parallel threads. The possible number of sub images are limited on two facts: for the chess board method the passed value for the number of sub images has to be a even square root and of course, the division of width or height with the root of the passed value has to result in a even number as well. Those two case have to be catch in the computation function of the Mandelbrot set.

Another method to split the image into sub areas would be using only rows. Each thread now has to calculate a part image with a resolution of width and height divided by the number of used threads. If want to split the work balanced regarding to the threads, its only allowed to specify an amount of threads which results in a even division of the height and this value.

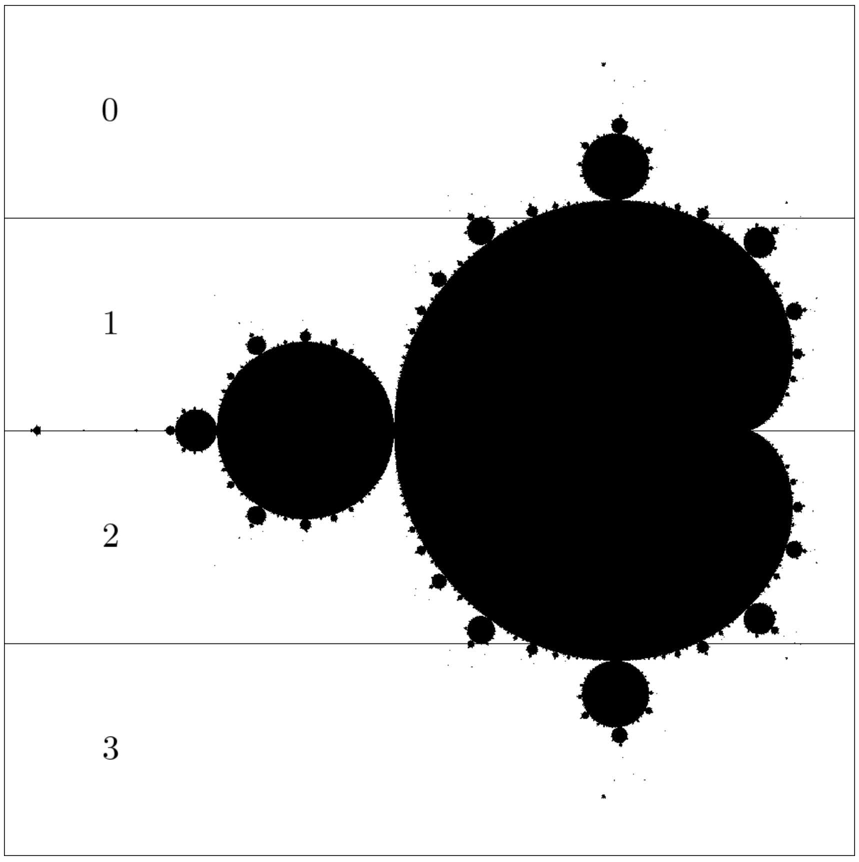


Figure 4.7: Split the Mandelbrot image using the row method.

To implement it more dynamically, we can assume also odd part images regarding to the height of the sub areas. In this case, the last row would have a smaller height. E.g for a main image with a resolution of 600 by 600 pixel, the rows would have a dimension of 600 by 85 pixel for seven threads, the last one in fact only 600 by 5 pixel.

A possible implementation for this scenario is seen in the following code snippet on the next page [see 4.3.2]. Regarding to the specified parameters *width*, *height* and the amount of threads, the start and end points are computed very similar im comparison to the chess board method. But due to the fact that we have a fixed width, our x coordinates are fixed too. So we only have to compute the start and end y coordinate for each sub area. In case of a odd result of the number of threads and the specified height, we have to add the rest of that division for the last end point y coordinate.

```
float Benchmark::runMandelbrotWith(unsigned int width, unsigned
    int height, int numThread) {

    ...

    for (int j = 0; j < numThread; j++) {
        unsigned minX = 0;
        unsigned maxX = width;
        unsigned minY = j * (height / numThread);
        unsigned maxY = (j+1) * (height / numThread);

        if(height % numThread != 0 && j == numThread - 1) {
            maxY += (height % numThread);
        }

        m[j] = new Mandelbrot(width, height, minX, maxX,
                                          minY, maxY, *queue);


        ...

    }
}
```

The mechanisms to benchmark this kind of computation are similar to the sum example [see Chapter 4.1]. A time measurement has to be implemented as well as a **Message Passing Model** to signal the end of the computation of every thread. After that the part results have to be collected to save them as a **.ppm* image file [see Chapter 4.3.3].

Before we describe the image format we store the Mandelbrot set into, we go into further details about calculating the Mandelbrot set. As already described in Chapter 4.2, the Mandelbrot set is based on a complex series defined by a simple function. Complex numbers are in fact 2-dimensional and they have an imaginary and real part. So in order to produce an image based on the complex series, we only have to plot the real part as x, the imaginary part as y coordinates on a 2D Cartesian system. If the calculated complex value belong to the Mandelbrot set, we will color this point red, otherwise black.

To get colors, we will use simple *RGB* values, so therefore each color can be created with mixing the three primary colors red, green and blue. To simplify the whole process, we will only limit ourselves as mentioned above to the colors red and black. And we will not implement gradations in the colors. So basically, for red we will save a one, and for black a zero. But how is it possible to generate a printable image based on the result matrix, where the point information are stored? For this we will use image format called *PPM*, the short form for "*portable pixmap format*" [see Chapter 4.3.3].

4.3.3 Image file format ppm

A *PPM* image is a uncompressed image file format in which the raw information of each pixel point is stored as a *RGB* value. The header section of the file will contain the width and the height as well as maximum color number, for a gradient color value range 255, for only two possible states zero or one.

```
P3                # ppm image format
3 3               # width x height
1                 # maximum color value (0-255 or 1), val(r,g,b)

0 0 0  => black             -
0 0 1  => blue              |   first row
0 1 0  => green             -
0 1 1  => magenta           -
1 0 0  => red               |   second row
1 0 1  => turquoise         -
1 1 0  => yellow            -
1 1 1  => white             |   third row
0 0 0                       -
```

Listing 4.1: Simple example for an ppm image file format

This will basically create a pixel matrix of three by three pixel filled with primary colors like shown in figure underneath.

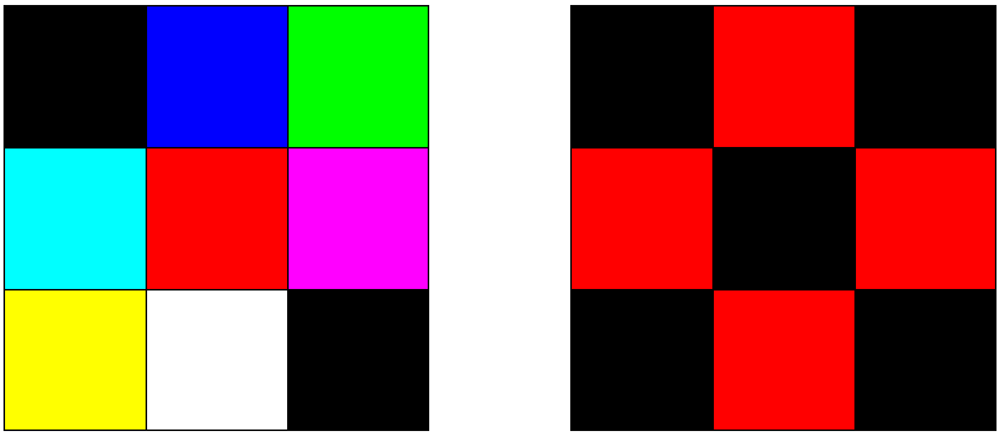


Figure 4.8: Colored and limited example of a simple pixel matrix image.

To interpret the *PPM* image file, we will read the header information, extract the width and height to go through the underneath stored *RGB* values. Each value will represent one pixel color and if a row is finished, a new starts; The pixel colors are simply listed one after the other.

If we now simplify that to the mentioned colors red and black, we would create a image only based on these colors. This is exactly what we are do when we compute the Mandelbrot set, and if we find a point which is inside the Mandelbrot set M, we assign a color, otherwise we will leave this point black.

4.3.4 Compression for reducing file size

So far, we haven't discussed the advantages or disadvantages of both methods to split the Mandelbrot image into sub areas mentioned in Chapter 4.3.2 and which one we will finally choose. Both methods are perfectly implementable on a machine with enough memory resources, especially *RAM* and *flash memory*. But if we think about our micro controller, the *ESP32*, we have indeed limited resources. So calculating and saving the Mandelbrot set on the *ESP32* is kind of different than on a usual x86 architecture with for example a powerful i7 processor in combination with 8GB of *RAM*.

If we want to have a precisely computation of the Mandelbrot image, it is necessary to have a quite high resolution, the colors for the sections inside or outside of the Mandelbrot set are not that important, to reduce file size, we limit them to red and black, because basically the *ppm* image is nothing more than a text file, so every character we save will decrease the file size.

```
P3              # ppm image format
600 600         # width x height
1               # maximum color value

0
0
1
1
0
1
0
1
```

Listing 4.2: Reduced ppm image file format

For an image with a resolution of 600 by 600 pixel we have 360000 characters to save including the header section and the carriage return line feed character for each new line. This will result in an overall file size of 2.2MB regardless of the pixel color information itself. The question arise if there is still a way to decrease the amount of characters we have to store. If we take a closer look to the example file above [see Listing 4.2], the information containing the Mandelbrot set is actually a bit stream, so what happen if combine a specific amount of those bits to a new value, probably one who has less characters overall.

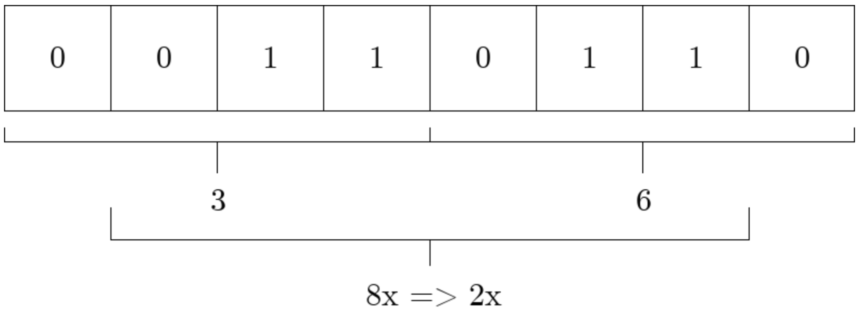
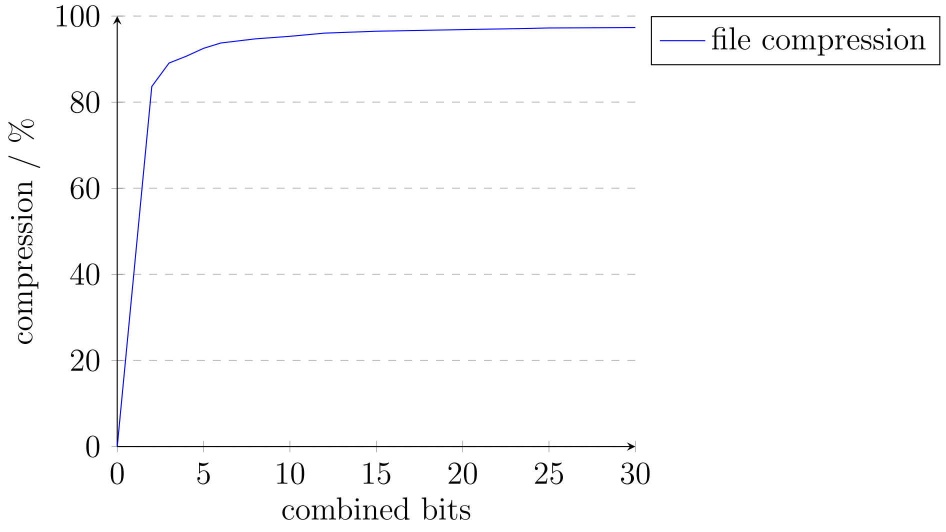


Figure 4.9: Method to reduce characters in a *PPM* file.

The procedure is quite simple: we just collect a specific amount of values - we will call them bits, in fact, they are true characters - and we interpret them as bits so that we can create based on these combinations a new real Integer value. A possible scenario is shown in Figure 4.9 were we combine each 4 bits to a new Integer value.

The question now could be how many bits we can possible combine? What are our limits? Well, first of all let's think in block buffers: If we have the first line of our Mandelbrot image, for a resolution of 600 by 600 pixel, we have 600 bits defining our first pixel row. To start a combining process, we have to check whether its possible to split those 600 bits into an even number of block buffers. So let's say into 120, that would mean we combine every 5 bits, because 600 divided by 120 result in 5. All odd number of block buffers are forbidden; this scenario i think is quite comprehensibly. Now its time to take into account both methods on how to split the Mandelbrot image into sub areas. However, we want to keep the compression and decompression process as simple as possible, that means only the last method is capable of achieving this aim, because using the first method, we have to be very carefully by combining the compressed part results to the final image; each compressed line form one part image belongs to several other lines on the same row.

Overall file size compression regarding the number of combined bits



Back to our combining problem: So how many bits can we combine as a maximum amount? We are literally limited to the hardware we are using. Based on the fact that we cast our bit stream to an Integer value, the maximum bits an Integer value can store depends on our hardware platform. For a 32-bit architecture, which also includes the *ESP32*, we have 32-bits for an Integer value.

Build on these facts, we are able to plot a graph which shows the relation between the result file size and the number of combined bits [see Graph 4.3.4].

We have to keep in mind that the number of combined bits depends on the specified resolution of the Mandelbrot image. Also, this method is not optimized in case of execution time, because the main bottle neck was storing the result into the *flash memory* and reducing *bandwidth* for transmitting the final result image to the webfrontend.

4.3.5 Graphical visualization

Our results we will receive from the *Benchmark* class are first of all the execution time values regarding how many threads we used to compute the problem in parallel (the *Computation.h* or the *Mandelbrot.h*), and of course, the Mandelbrot set as a *ppm* image.

A way to present those results state of the art is a visualization without any additional software which the user has to install. So we decided to implement a webfrontend, which can be easily accessed from the *ESP32* itself through logging into the created wireless network and opening the users web browser. The web page is written in *VueJS*, a *Javascript* framework, and uses asynchronous *http* requests and a websocket connection to communicate with the *ESP32 backend*.

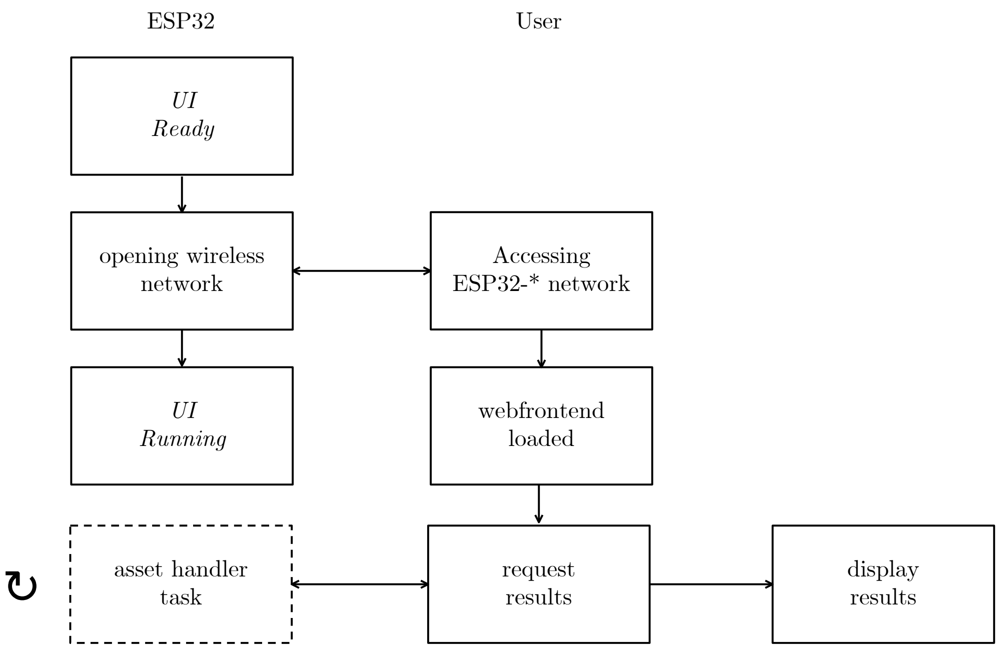


Figure 4.10: State machine implementation for frontend-backend communication.

The main part for the *frontend* is basically to decompress the received *ppm* image file and to plot the execution time results taken from the benchmark for the *Computation* and *Mandelbrot* example. Between the *frontend* and the *backend*, the number of combined bits are preset, because we are trying to combine as a maximum 30 bits regarding to the specified width, so this number can be less than 30, but we try to combine as much as possible. This number has to be preset, otherwise the decompression will fail. The received *ppm* image, after it is decompressed, will be printed on a canvas, so basically we loop through the *ppm* image and set pixel by pixel regarding whether we have to give them a color or not.

For displaying the execution time results, we use *ChartJS*, an additional component to plot charts with *Javascript*. We use four charts to display the execution time of both examples as well as the gained speed up factor. But we will only transmit to the *frontend* the execution time values, because the speed up factors can be calculated on *frontend* side to reduce bandwidth.

4.4 Benchmark setup of the Mandelbrot fractal computation

Now we will take into account how to finally start the whole benchmark process. This will include initializing the necessary parameters for the *Computation* and *Mandelbrot* example and how to enable the *frontend* to display the results graphically.

At First, we have to include the *BemchmarkHandler.h* class. This class will include the *Benchmark.h* file and serve all necessary functions to start and to handle the different states of the benchmark process. Due to the fact that we are using several threads for different task, we decided to have at least to global pointers to ensure access to the necessary resources. To avoid race conditions, most of the functions are covered with *Mutex's*, or are it was ensured that the function will only be accessed from one thread.

```
#include "libs/benchmark/BenchmarkHandler.h"
#include "libs/websocket/WebsocketHandler.h"

BenchmarkHandler *handler;
WebsocketHandler *wsHandler;

void setup() {
    /* serial communication */
    Serial.begin(115200);

    /* initialize internal storage */
    if(!SPIFFS.begin(true)){
        Serial.println("[main] SPIFFS mount failed.");
        return;
    }

    delay(1000);

    ...

}
```

Listing 4.3: Necessary include's for the benchmark setup

After defining the global pointers, we will initialize the *uart* interface to enable serial communication for debug purposes and the internal *flash memory* for saving our results to files. To start on of our benchmark examples, we have to call the related callback functions:

```
    handler = new BenchmarkHandler(16);

    handler->startComputationBenchmark();
    handler->startMandelbrotBenchmark();
```

Listing 4.4: Example on how to start the benchmark process

By setting up the object on the *heap* by calling the *new* operator, we pass through the constructor the amount of threads we want to create regardless which benchmark example we will execute, so this number of threads will be fixed.

Every time we start a benchmark example, the execution time results will be stored in the handler's properties. To display them, we have to call *displayResult()* method. In fact, the results are also stored after each benchmark run in a local *results.txt* file in the internal *flash memory*. The first line of the file will contain the results of the first run of the benchmark, the second one the ones of the second run of the benchmark.

```
void setup() {
    ...

    handler = new BenchmarkHandler(16);

    /* first we run the Computation example */
    handler->startComputationBenchmark();
    handler->displayResult();

    /* The Mandelbrot fractal will be stored in mandelbrot.ppm
   */
    handler->startMandelbrotBenchmark();
    handler->displayResult();

    handler->finalize();
}

void loop() {
    handler->displayUI();
}
```

Listing 4.5: The final benchmark code.

If the benchmark runs finished successfully, we are able to finalize the process by calling the method *finalize()*. This method will check whether all results are stored and benchmark finished successfully. Only in this case, it will enable the *UI_Ready* state, so the *ESP32* can create a wireless network and serve the necessary web files for the *frontend*.

Meanwhile, the infinite main loop while wait until the UI state changed to *UI_RUNNING*, after that the user is able to access the results from the *frontend* loaded in the users web browser.

Chapter 5

Conclusion

After running the benchmark functions mentioned in Chapter 4.4, the *ESP32* will serve a wireless network, on which the user can log into. While entering the default IP address (*192.168.4.1*) of the *ESP32*, the *backend* will serve all web files.

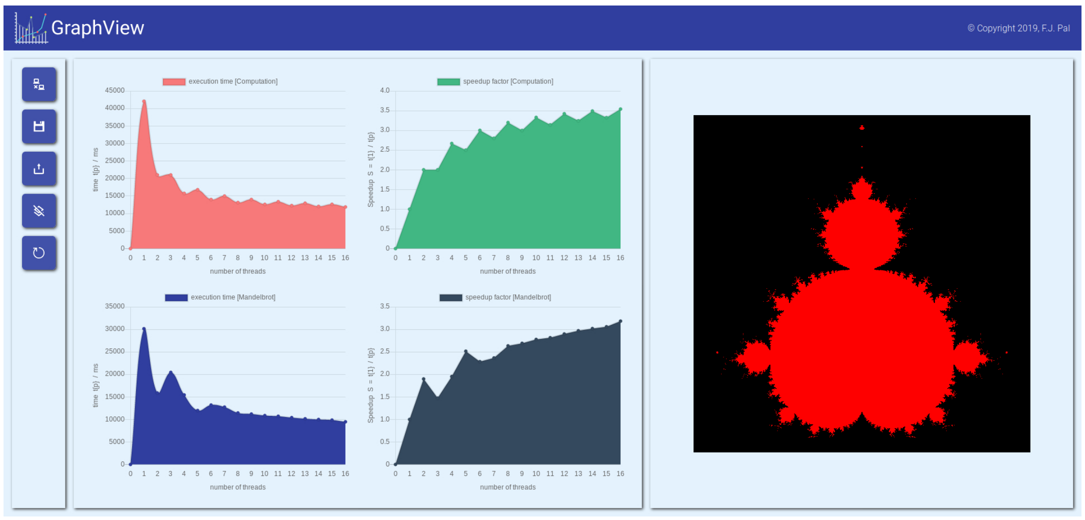


Figure 5.1: Successfully loaded *frontend* with all results.

The calculated results can now be loaded to the *frontend* using the buttons on the left. Therefore it is necessary to open the websocket connection to the *ESP32* (first button). After that we can proceed with downloading the results to our local drive by using the save button.

For both examples the chart results are quite impressive. Even after we left real time parallelism (limited to the real number of cores on the hardware system), in our case two threads running independently on different cores, it was possible to achieve a speedup. By using 62 threads for both examples, we were able to reach a maximum number of speed up of 3.65 (*Mandelbrot*) and 3.7 (*Computation*). Due to the fact that we run the *Benchmark* on a real time operating system (*FreeRTOS*), several runs of the same setup will result into almost the same results regarding execution time, because the execution time of each instruction is predictable.

The charts mentioned on the next page leads us to the assumption, that for our setup we will reach some kind of limit in speedup. But its not really comparable to the limit Amdahl's Law predicted, because we are not using real processor cores, we are dealing with thread level parallelism.

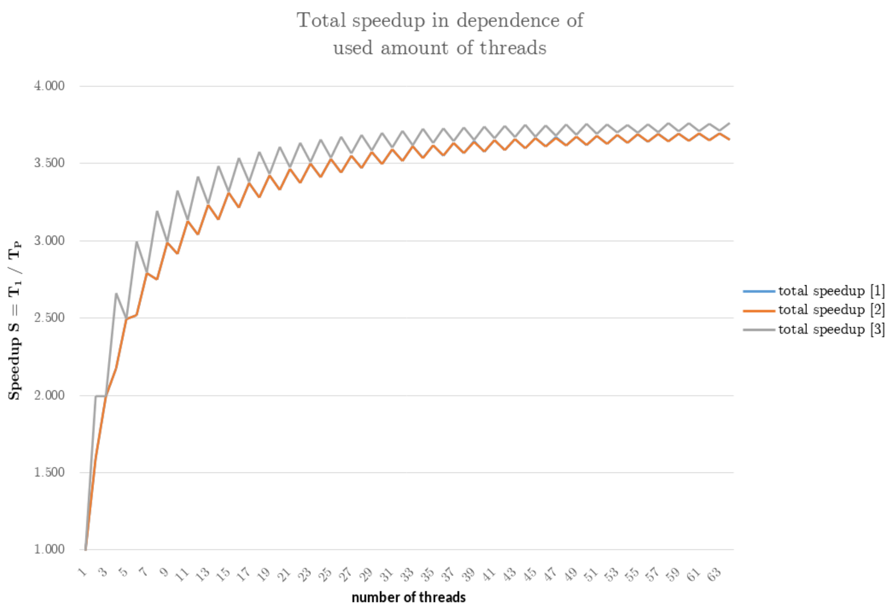


Figure 5.2: Speedup factor gain for the *Computation* example.

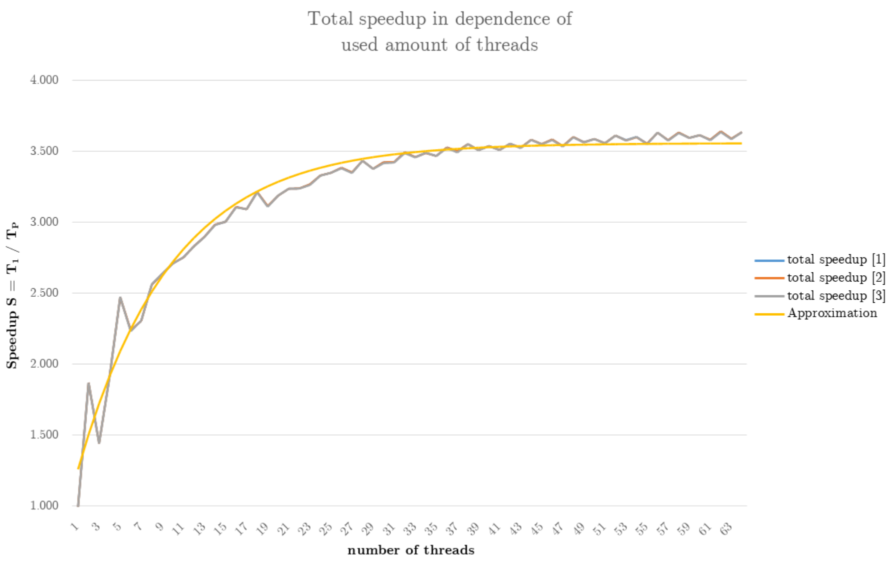


Figure 5.3: Speedup factor gain for the *Mandelbrot* example.

Bibliography

[1] Plattform Industrie 4.0. “Positionspapier, Leitbild 2030 für Industrie 4.0 - Digitale Ökosysteme global gestalten”. In: *Plattform Industrie 4.0* (Apr. 2019). URL: `https://www.plattform-i40.de/PI40/Redaktion/DE/Downloads/Publikation/Positionspapier%20Leitbild.pdf?__blob=publicationFile&v=5`.

[2] Plattform Industrie 4.0. *Was is Industrie 4.0?* 2019. URL: `https://www.plattform-i40.de/PI40/Navigation/DE/Industrie40/WasIndustrie40/was-ist-industrie-40.html`.

[3] Umut A. Acar and Guy E. Blelloch. *Algorithms: Parallel and Sequential.* Pittsburgh, USA: Carnegie Mellon University, Department of Computer Science, Feb. 2019.

[4] Tosiron Adegbija et al. “Microprocessor Optimizations for the Internet of Things: A Survey”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* PP (June 2017), pp. 1–1. DOI: `http://dx.doi.org/10.1109/TCAD.2017.2717782`.

[5] Vitorović Aleksandar. *Advances in Computers.* 2014. URL: `https://www.sciencedirect.com/topics/computer-science/parallel-programming-model/`.

[6] G.M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of Spring Joint Computer Conference.* New York: ACM, 1967, pp. 483–485.

[7] Goldberg David. *Multiprocessors and Thread-Level Parallelism.* 2003. URL: `http://www.csit-sun.pub.ro/courses/cn2/Carte_H&P/H%20and%20P/chapter_6.pdf/`.

[8] Frank Devai. “The Refutation of Amdahl’s Law and Its Variants”. In: Sept. 2018, pp. 79–96. ISBN: 978-3-662-58038-7. DOI: `http://dx.doi.org/10.1007/978-3-662-58039-4_5`.

[9] Gabriel Edgar. *An Introduction to Parallel Computing.* URL: `http://www2.cs.uh.edu/~gabriel/courses/mpicourse_03_06/Introduction.pdf`.

[10] Wikipedia Encyclopedia. *Mandelbrot set.* 2019. URL: `https://en.wikipedia.org/wiki/Mandelbrot_set`.

[11] Wikipedia Encyclopedia. *Parallel programming model.* 2019. URL: `https://en.wikipedia.org/wiki/Parallel_programming_model`.

[12] Prof. Robert van Engelen. *Parallel Programming Models.* URL: `https://www.cs.fsu.edu/~engelen/courses/HPC/Models.pdf`.

[13] Daniels220 at English Wikipedia. *SVG Graph illustrating Amdahl's law.* [Online; accessed October 29, 2019]. Apr. 2008. URL: `https://commons.wikimedia.org/w/index.php?curid=6678551`.

[14] Pawel Gepner and Michal Kowalik. "Multi-Core Processors: New Way to Achieve High System Performance." In: Jan. 2006, pp. 9–13. DOI: `http://dx.doi.org/10.1109/PARELEC.2006.54`.

[15] J Hennessy and D Patterson. *Computer Architecture. [Elektronisk Resurs] : A Quantitative Approach.* 2011.

[16] W. Daniel Hillis, Jr. Steele, and Guy L. "Data parallel algorithms". In: *Communications of the ACM* (1986). DOI: `http://dx.doi.org/10.1145/7902.7903`.

[17] Computer Hope. *Computer processor history.* 2019. URL: `https://www.computerhope.com/history/processor.htm`.

[18] Harald Koestler, C. Moeller, and F Deserno. "Performance Results for Optical Flow on an Opteron Cluster Using a Parallel 2D/3D Multigrid Solver". In: (Jan. 2006). URL: `https://www.researchgate.net/publication/236892123`.

[19] M. J Ladner R. E.and Fischer. "Parallel Prefix Computation". In: *Journal of the ACM* (1980). DOI: `http://dx.doi.org/10.1145/322217.322232`.

[20] Nigel Lesmoir-Gordon. "THE MANDELBROT SET, FRACTAL GEOMETRY AND BENOIT MANDELBROT - The Life and Work of a Maverick Mathematician". In: *Medicographia* 34 (June 2012), p. 353. URL: `https://www.researchgate.net/publication/270285889`.

[21] Sheik Dawood M. "Review on Applications of Internet of Things (IoT)". In: (Dec. 2018). URL: `https://www.researchgate.net/publication/329672903`.

[22] Belean Marian and Pal Franz Joseph. *esp32-BasicParallelProcessing.* 2019. URL: `https://github.com/josephpal/esp32-BasicParallelProcessing`.

[23] Khaled Mashfiq. *NONLINEAR EARTHQUAKE ENGINEERING SIMULATION USING PARALLEL COMPUTING SYSTEM.* Dec. 2012. DOI: `http://dx.doi.org/10.13140/RG.2.2.21215.87208`.

[24] Tim Mattson, Beverly Sanders, and Berna Massingill. "Patterns for Parallel Programming". In: (Sept. 2004).

[25] Zhang N. "A Novel Parallel Scan for Multicore Processors and Its Application in Sparse Matrix-Vector Multiplication". In: *IEEE Transactions on Parallel and Distributed Systems* 23.3 (Mar. 2012), pp. 397–404. DOI: `http://dx.doi.org/10.1109/TPDS.2011.174`.

[26] Greenlaw Raymond. *Limits to Parallel Computation: P-Completeness Theory.* New York: Oxford University Press, 1995.

[27] Pandey Siddharth. *Flynn's taxonomy.* URL: `https://www.geeksforgeeks.org/computer-architecture-flynns-taxonomy/`.

[28] Kota Sujatha et al. "Multicore Parallel Processing Concepts for Effective Sorting and Searching". In: *IEEE* (2015). DOI: `http://dx.doi.org/10.1109/SPACES.2015.7058238`.

[29] Karlsruhe Institute of Technology. “Multi-core processors for mobility and industry 4.0”. In: *PHYSORG* (2016). URL: https://phys.org/news/2016-12-multi-core-processors-mobility-industry.html.

[30] Klaus-Dieter Thoben, Stefan Wiesner, and Thorsten Wuest. “”Industrie 4.0” and Smart Manufacturing – A Review of Research Issues and Application Examples”. In: *International Journal of Automation Technology* 11 (Jan. 2017), pp. 4–19. DOI: http://dx.doi.org/10.20965/ijat.2017.p0004.

[31] Gudula Rünger Thomas Rauber. *Parallel Programming for Multicore and Cluster Systems*. Germany: Second Edition, Springer Verlag, 2013.

[32] Unkown. *The Mandelbrot Set*. 2019. URL: http://warp.povusers.org/Mandelbrot/.

[33] Balaji Venu. “Multi-core processors - An overview”. In: (Oct. 2011). URL: https://www.researchgate.net/publication/51945986.

[34] Dragan Vuksanović, Jelena Vešić, and Davor Korčok. “Industry 4.0: the Future Concepts and New Visions of Factory of the Future Development”. In: Jan. 2016, pp. 293–298. DOI: http://dx.doi.org/10.15308/Sinteza-2016-293-298.

[35] Benjamin Wah. “Computer Architecture”. In: *Wiley Encyclopedia of Computer Science and Engineering* (2008).

[36] Yousaf Zikria et al. “Internet of Things (IoT) Operating Systems Management: Opportunities, Challenges, and Solution”. In: *Sensors* 8 (Apr. 2019), pp. 1–10. DOI: http://dx.doi.org/10.3390/s19081793.