Parallel Programming on Embedded Multicore System ESP32

Universitatea Politehnica Timișoara

Marian Belean
Franz Joseph Pal

WS2019/2020
November 19, 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.

Declaration

I hereby certify that I have done the final thesis on my own, that I have completely and accurately stated all the aids I have used and identified everything individually, which was taken from the work of others unchanged or with modifications.

The topic of the submitted work was jointly with Mr. / Mrs. (...) (Bachelor and Master Thesis No. (...)).

Timișoara, the November 19, 2019

Signature:

Non-disclosure notice

This work contains confidential information. In spite of the anonymous presentation of the researched organisations, readers might conclude their identity. Therefore copying, quoting or publishing is not allowed without my explicit authorisation. Furthermore, disclosure of the information to anyone other than the examination board or lectors is not authorized.

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 29, p1] - and are playing an important role in data processing and process automation [see 32, p294] [or 28, p1]. On the other hand, in addition to efficiency in energy consumption, performance in terms of computation time [see 32, 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 34, 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 31, p4] increases the desired cost factor in terms of energy consumption [see 13, chapter 3] and computational efficiency [see 31, 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 [20]

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 19, 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 [31], the decade in which the microprocessor era started, the overall performance of a processor has increased [13]. This goal was achieved by several points, including "*sophisticated process technology, innovative architecture or micro-architecture*" [see 13, Chapter 1, p2]. In fact, increasing the clock speed of a single core processor, like Moore's Law predicted [31], was usually reached by increasing the number of transistors on the chip [31]. However, this go along side with the increase in complexity [see 31, Pollack's rule], which mean, that doubling the logic of a processor result in a performance boost of only 40% [see 31, Chapter 2].

Another huge problem chip manufacturers have to deal with is leakage power [see 13, Chapter 2, p3], because the "*transistor leakage current increases as the chip size shrinks*" [see 31, p2] [see Chart 3.1]. An increase of leakage current of the transistors also result in a increase of the die's temperature [13] along side the total power consumption as well.

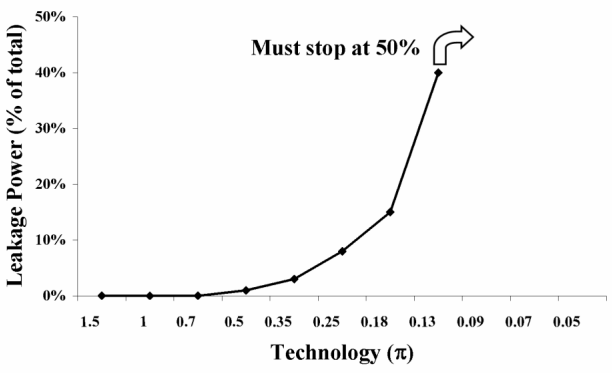


Figure 3.1: Leakage Power (% of total) vs. process technology [13]

Furthermore, a increase of the processor clock frequency to speed up the performance is only available to a suffisticating limit of 4GHz [31]. After this frequency threshold, also known as reaching the power wall, the "*power dissipation*" [see 31, p2] increases again.

Facing these types of problems such as "*chip fabrication costs, fault tolerance, power efficiency, heat dissipation*" [see 31, 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 **??**] was only available after dual or multi-core processors were invented in 2005 [16], the topic itself and efficient software implementations are still treated in scientific work [3] [27].

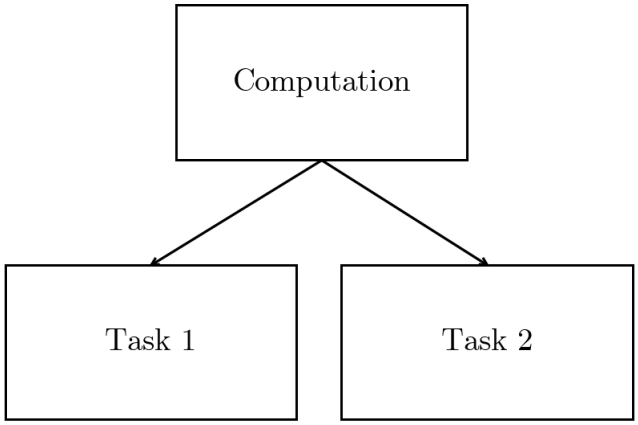


Figure 3.2: The basic concept of a simple concurrency computation.

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 22, 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].

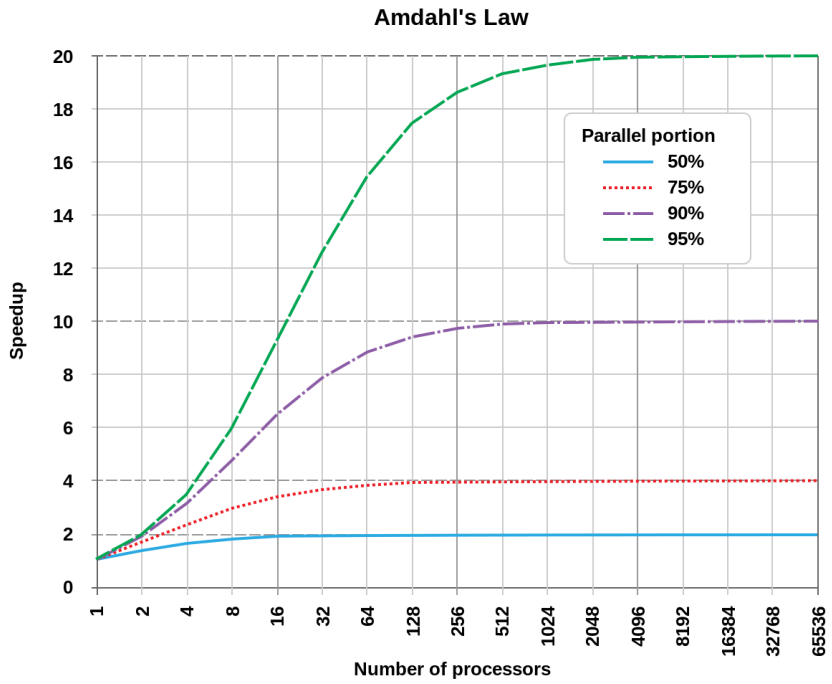


Figure 3.3: The limited speed-up of a program, which can be parallelized, depending on the number of parallel executions [12] [similar to 22, p4].

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 $(1\text{-}f)pt_p$. 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].

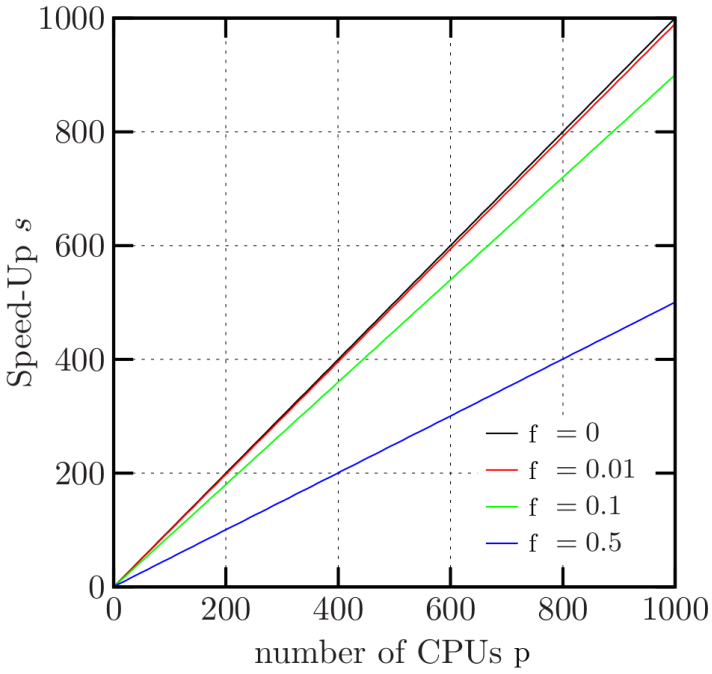


Figure 3.4: Gustafson's Law. In contrast to Amdahl we now have no upper limit to the speed up. f only determines the slope of the speed-up [17].

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 23, 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.

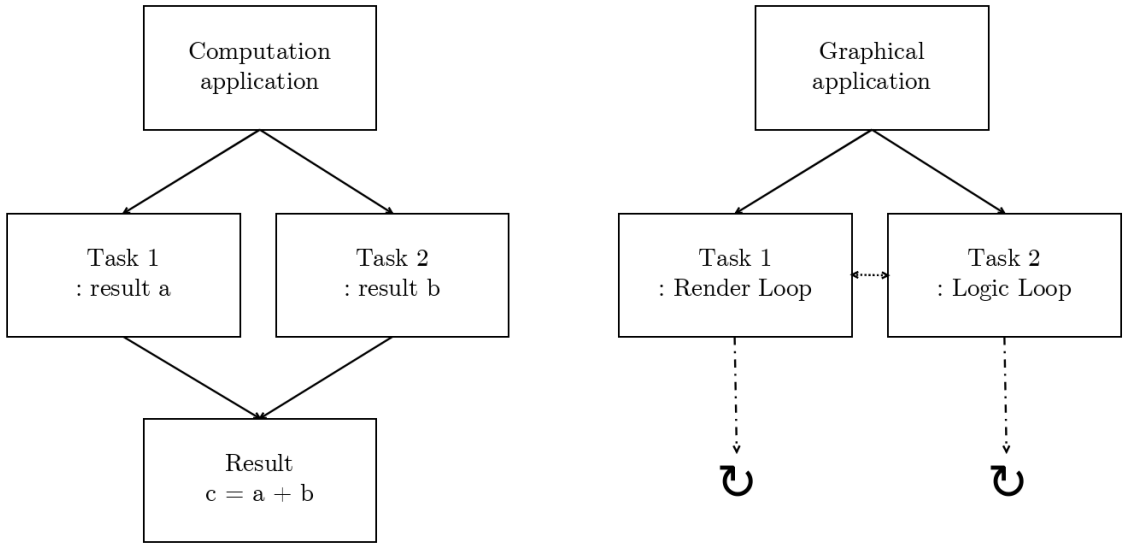


Figure 3.5: Comparison of independent and dependent tasks in a general concurrency application.

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 31, Fig. 3, p4], along side instructions per core (IPC) [13] [further inf. in Chapter 3].

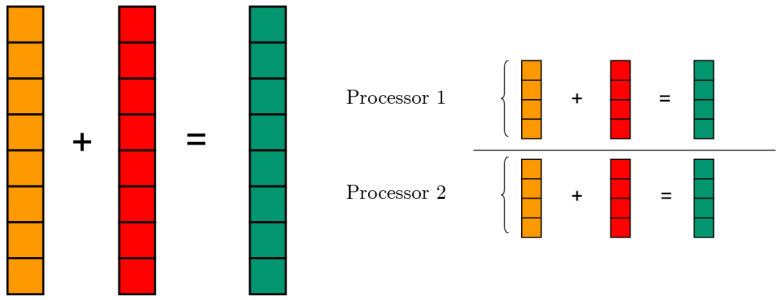


Figure 3.6: Adding to arrays of integers for speed up in concurrency [see 9, p3].

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 [31] [23]. 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 23, p15], alongside the ability to take less attention on control issues such as task orders, synchronization, (shared) memory access or even task communication [11]. 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 23, 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 23, p6].

4. Independence:

   To ensure maximum efficiency, parallel computations should be independent.

These principles result in four different **design patterns** [see 23, 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 23, p25] like Loop Parallelism, Fork/Join, Shared Data or Shared Queue [23].

- Implementation Mechanism:

  Using programming languages which offer the opportunity to real parallel computing (e.g. C++, OpenMP & Pthreads, MPI, OpenCL) [23].

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:

$$\begin{aligned} s &= \sum_{i=0}^{N/2-1} (a[i] \cdot b[i]) + \sum_{i=N/2}^{N-1} (a[i] \cdot b[i]) \\ &= \underbrace{\sum_{i=0}^{N/2-1} (a_{local}[i] \cdot b_{local}[i])}_{\text{p0}} \quad + \quad \underbrace{\sum_{i=N/2}^{N-1} (a_{local}[i] \cdot b_{local}[i])}_{\text{p1}} \end{aligned} \tag{3.7}$$

process communication

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] p_0 and 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* [25]. 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 [25].

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 [25], sequential models are in fact parallelizable, but this comes "*at the cost of an enormous number of processors*" [see 25, Chapter 5.5, p69], resources or even execution time [see 25, 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 [24]. 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 [15] or less parallelism [18].

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 [33].

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 [26].

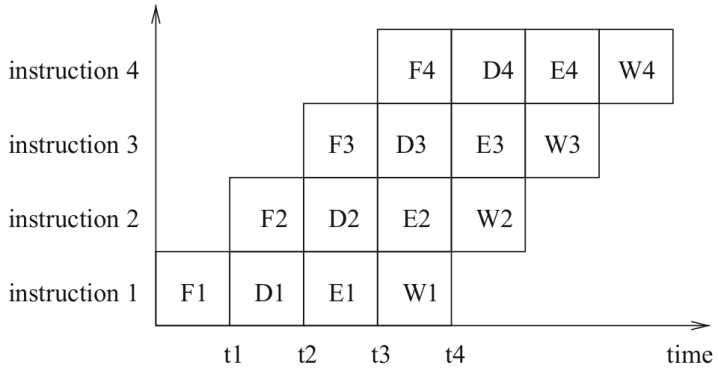


Figure 3.7: Overlapping execution of four independent instructions by pipelining. The execution of each instruction is split into four stages: *fetch (F)*, *decode (D)*, *execute (E)*, and *write back (W)* [see 30, Fig. 2.1, p11].

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] [30]:

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.

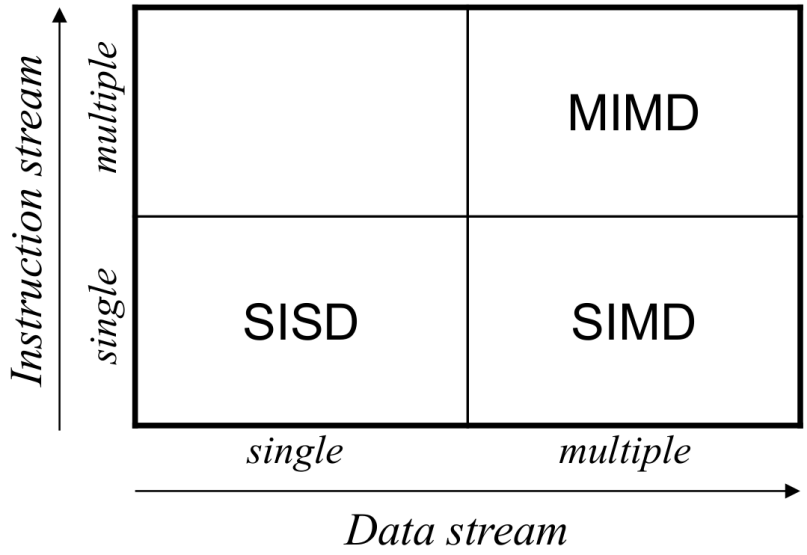


Figure 3.8: Visualization of Flynn's Taxonomy [see 11, p5].

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 [14].

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 [30]. 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 30, 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 30, 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 30, p106]. It also takes execution time along side with the providing of resources regarding to an architectural model into account [30]. 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 30, 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 30, 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 11, p4]. In general, we distinguish between three different process interaction models: the first one is called Shared address space programming (shared memory) model [11]. 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 [11] [10].

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] [10].

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] [10].

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] [10].

Chapter 4

Project documentation

4.1 Concept development

...

4.2 Project structure

...

4.3 Simple mathematical computation examples for Parallel Programming

As a first example, which we try to implement on our ESP32, we will discuss a similar computation as already mentioned in Chapter 3.2. 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.

$$\begin{aligned} s &= \sum_{i=0}^{N_1} \left(\sum_{j=0}^{N_2} (i \cdot j) \right) \\ &= \underbrace{\sum_{i=0}^{\frac{N_1}{2}-1} \left(\sum_{j=0}^{N_2} (i \cdot j) \right)}_{\text{p0}} \quad + \quad \underbrace{\sum_{i=\frac{N_1}{2}}^{N_1} \left(\sum_{j=0}^{N_2} (i \cdot j) \right)}_{\text{p1}} \end{aligned} \tag{4.2}$$

process communication

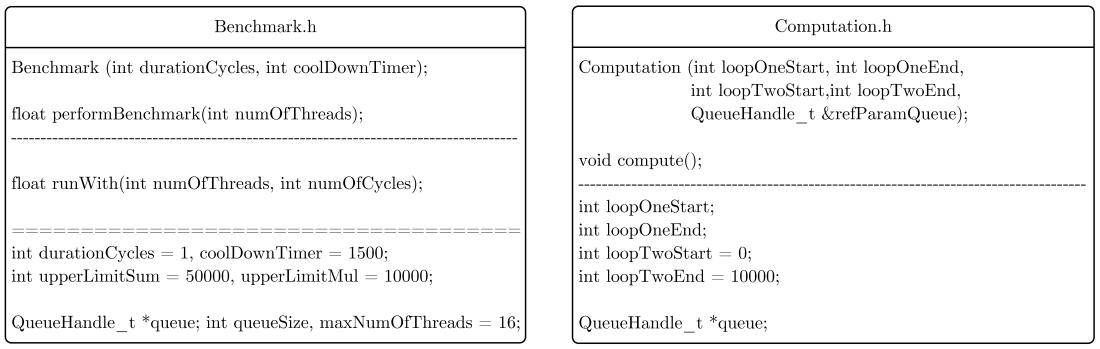
For collecting the part results from the parallel computed sums, some sort of process communication [see formula 4.2] between the threads is necessary, more then, its impossible without it. But actually, this is the task, we have to take care about most. Otherwise we will run into race conditions while 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 left 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.

4.3.1 UML diagram

This little mathematical example can be found on our GitHub repository [21] and is basically split into three parts. The main **.ino* file, and two classes, the *Computation.h* and the *Benchmark.h*.



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

Figure 4.1: Overview over both main implementations regarding Chapter 4.3

4.3.2 Implementation

The *Benchmark.h* class has several functions to set the necessary parameters to start a benchmark like mentioned at the end of Chapter 4.3. The computational problem discussed in formula 4.2 is implemented in *Computation.h* class regarding to the following function.

```
1  | void Computation::compute() {
2  |     long count = 0;
3  |
4  |     for (int i = getStartOne(); i < getEndOne(); i++) {
5  |         for (int j = getStartTwo(); j < getEndTwo(); j++) {
6  |             count += i * j;
7  |         }
8  |     }
9  |
10 |     /* process communication */
11 |
12 |     xQueueSend(*queue, &count, portMAX_DELAY);
13 | }
```

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 21, 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.

```
1 | Benchmark bench(1, 1500, 8);
2 |
3 | for (int i = 0; i < numOfRunningThreads; ++i) {
4 |     results[i] = new float(bench.performBenchmark(i + 1));
5 | }
```

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 () and two on core 1 (). The new limits for the part sums can be calculated using the formula below:

```
1 | float Benchmark::runWith(int numOfThreads) {
2 |     ...
3 |
4 |     for (int k = 0; k < numOfThreads; k++) {
5 |
6 |         /* outer sum i = 0 ... N1 */
7 |         int lowLimSum = (50000 / numOfThreads) * k;
8 |         int uppLimSum = (50000 / numOfThreads) * (k + 1);
9 |
10 |        /* inner sum j = 0 ... N2 */
11 |        int uppLimMul = 10000;
12 |
13 |        if(50000 % numOfThreads != 0 && k == numOfThreads - 1) {
14 |            uppLimSum += (50000 % numOfThreads);
15 |        }
16 |
17 |        c[k] = new Computation( lowLimSum, uppLimSum,
18 |                                0, uppLimMul, *queue );
19 |
20 |        ...
21 |    }
22 | }
```

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 down by and if clause and a "*modulo*" operator [see 4.3.2]. After determine the necessary limits and amount of threads, we are no 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.

```
1  | for (int j = 0; j < numOfThreads; j++) {
2  |     ...
3  |
4  |     c[j] = new Computation( lowLimSum, uppLimSum,
5  |                             0, uppLimMul, *queue );
6  |
7  |     xTaskCreatePinnedToCore(
8  |         producerTask,
9  |         (j+1)%2 == 0 ? "calcTask0" : "calcTask1",
10 |         sizeof(c[j]) * 32 * 8,
11 |         c[j],
12 |         0,
13 |         NULL,
14 |         (j+1)%2 );
15 | }
```

4.3.3 Implementation of the Message Passing Model

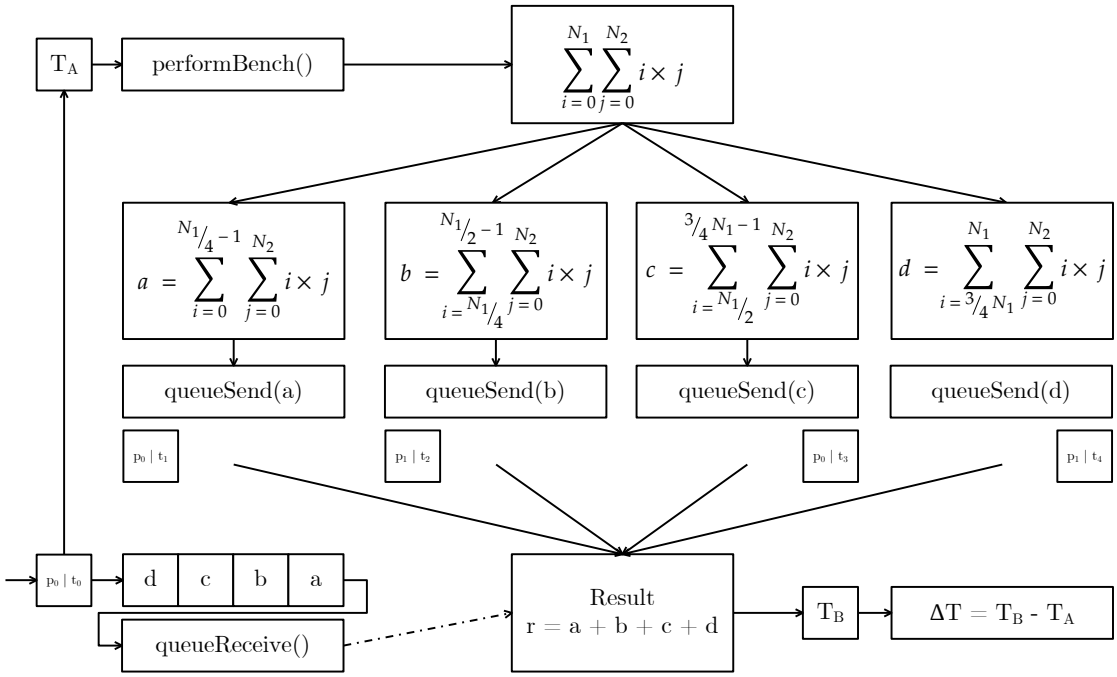


Figure 4.2: ...

4.4 Class diagramm

...

4.4.1 C++ Backend benchmark

...

4.4.2 Vuejs Frontend

...

4.5 Benchmark setup

...

Chapter 5

Conclusion

...

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. *Parallel programming model*. 2019. URL: https://en.wikipedia.org/wiki/Parallel_programming_model.

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

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

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

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

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] Benjamin Wah. "Computer Architecture". In: *Wiley Encyclopedia of Computer Science and Engineering* (2008).

[34] 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.