Mälardalen University
School of Innovation, Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Software Engineering
30.0 credits

RE-ENGINEERING SEQUENTIAL SOFTWARE TO INTRODUCE PARALLELIZATION

Andreas Granholm
agm11002@student.mdh.se

Examiner: Antonio Cicchetti
Mälardalen University, Västerås, Sweden

Supervisors: Federico Ciccozzi and Masoud Daneshtalab
Mälardalen University, Västerås, Sweden

Company supervisor: Johnny Holmberg
ABB Robotics, Västerås

September 6, 2017
Abstract

In the quest for additional computational power to provide higher software performance, industry have shifted to multi-core processing units. At the same time, many existing applications still contain sequential software; in these cases, multi-core processors would not deeply improve performances and in general would be under-utilized since software running on top of them are not conceived to exploit parallelization. In this thesis we aim at providing a way to increase the performance of existing sequential software through parallelization and at the same time minimizing the cost of the parallelization effort. The contribution of this thesis is a generic parallelization method for introducing parallelization into sequential software using multi-core CPUs and GPUs. As a proof-of-concept we ran an experiment in industrial settings by applying the proposed parallelization method to an existing industrial system running sequential code. Additionally, we compare the method we propose to existing methods for parallelization.
# Table of Contents

1 Introduction 3

2 Problem formulation 4

3 Background 5

4 Related Work and Comparison to our Contribution 9

5 Research Method 11

6 Results 13
   6.1 Analysis of existing methods for parallelization 13
   6.2 Definition of the parallelization method 14
      6.2.1 Specification phase 15
      6.2.2 Analysis phase 16
      6.2.3 Development phase 17

7 Evaluation 19
   7.1 Experiment 19
      7.1.1 Specification phase 21
      7.1.2 Analysis phase 21
      7.1.3 Development phase 25

8 Discussion 30

9 Conclusion 31

References 33
1 Introduction

With the increasing demand of computational power created by modern systems, multi-core processing units have become increasingly adopted in industry [1]. The reason is that increasing the frequency of the processor to achieve better performance is simply not viable due to physical limitations of the single-core chip [2]. Practically, the speed of a Central Processing Unit (CPU) is limited by both heat generation and gate delay of transistors. Instead, in order to increase the performance of digital systems, multi-core processing units are used to run computations in parallel, thus increasing the performance of software through parallelization. Today, besides multi-core CPUs, developers have an additional parallel computing platform for general-purpose programming at their disposal, namely Graphical Processing Unit (GPU). Initially designed for graphical computations, GPUs offer many more computational units than a multi-core CPU, but with the drawback of generally being slower in terms of clock frequency.

Despite the accessibility to these parallel hardware platforms, many software applications still run code sequentially. In order to utilize the full potential of parallel hardware, existing software needs to be re-engineered to exploit parallelization. As a response to this shift, methods for developing software for parallel as well as distributed systems have emerged [3]. However, parallel software is generally more complex than sequential software [4]. Because of this, developers need more guidance in finding concurrent tasks in existing software [5]. Although there are techniques for automated parallelization, they are not able to produce as good results as an engineer would manually achieve [6]. While there are many tools and programming guidelines available for parallelization of software, the increased complexity of ”thinking parallel” still poses a problem for software engineers [7].

The goal of parallelizing software is usually to increase the performance, but by doing this other quality attributes are usually affected, often negatively. According to Bass et al. [8] two quality attributes that affects the cost of a system greatly, sometimes even more costly combined than the development itself, are testing and modifying the system. Because parallel software is generally more complex than sequential software [4], the task of introducing parallelization to existing software can be a very costly form of modification. In this study we are interested in finding out which aspects of a system must be considered when parallelizing legacy sequential software and how this process can be simplified. This study therefore focuses on increasing the performance of existing software through parallelization, while at the same time aims to reduce the cost of the parallelization effort. In addition to this, we explore how this process can benefit from the utilization of both CPU and GPU when introducing parallelization in existing software. While the contribution of this work is meant to be generic, proofs of concept are provided for software parallelization using C++ as the main programming language.

The outline of the rest of the report will be as follows: in Section 3, a background will be presented, clarifying some of the concepts used in this report and further motivating the work. Next, the previous work related to the topic is presented and compared to the contribution of our work in Section 4. Following this, the method used to solve the problem will be presented in Section 5. In Section 6, the result of this thesis is presented. An evaluation of the results is shown in Section 7. Furthermore, in Section 8 there is a discussion of the work presented in this thesis. Finally, the thesis will end with a conclusion in Section 9.
2 Problem formulation

Today, many systems have at their disposal multi-core processing units, but run sequential software; this leads to an under-utilization of the available hardware. Therefore, in order to increase the utilization of the available hardware the software should be modified to introduce parallel computation, thus increasing its performance. However, due to the complex nature of parallel software in comparison to sequential software [4], there is a risk that the effort required to modify the software becomes high. While refactoring software comes with its own challenges, this study focuses on the process of introducing parallelization into existing sequential software. Therefore, this study tackles the following research challenges:

- Location of parallelization potential
- Utilization of both multi-core CPU and GPU when introducing parallelization

A limiting factor of the potential speedup through parallelization is that not all computations can run in parallel due to dependencies on previous computations. This introduces the first challenge, which is to locate parallel potential within the existing software. With the access to both multi-core CPUs and GPUs, it is important to maximize the utilization of the hardware in order to allow the software to reach its full potential. Due to the differences of the two platforms, on the hardware level, they are suitable for different problems and choosing the most suitable one can therefore yield more optimal performance. Choosing which processing unit type is most suitable for a specific parallel software piece is the second research challenge of this study.

The contribution of the study is a generic parallelization method for re-engineering existing sequential software. The method is meant to provide a structured and well-defined way to port legacy software from sequential to parallel platforms.
3 Background

Traditionally the way to improve performance of a processor was to increase its clock speed. However, due to physical limitations such as increased heat dissipation, this is simply not a viable option anymore. With the flattening single thread performance, IBM released the first multi-core processor in 2001 called Power4 [9] and the industry is shifting to multi-core solutions in order to gain more performance by utilizing parallelization. This means that performance of processors is now increased by the number of cores rather than by simply increasing their frequency [5]. In order to utilize the full potential of the hardware to achieve greater performance, the software must be able to run multiple instructions concurrently on these multiple cores. Much has happened since 2001; a comparison between Power4 and the currently top performing CPUs from Intel and AMD according to PassMark Software benchmarks [10] is shown in Table 1.

<table>
<thead>
<tr>
<th>Release</th>
<th>Unit</th>
<th>Frequency</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>POWER4</td>
<td>1.3 GHz</td>
<td>2</td>
</tr>
<tr>
<td>2016</td>
<td>Intel Xeon E5-2679 v4</td>
<td>2.5 GHz (3.3 GHz Boost)</td>
<td>20 (2 logical per physical)</td>
</tr>
<tr>
<td>2017</td>
<td>AMD Ryzen 7 1800X</td>
<td>3.6 GHz (4.0 GHz Turbo)</td>
<td>8 (2 logical per physical)</td>
</tr>
</tbody>
</table>

Table 1: Multi-core CPU then and now [9, 10]

Compared to Power4 the Intel Xeon E5-2679 v4 has increased the number of cores by a factor 10 while at the same time having a 2.5 times higher frequency when boosted. Besides multi-core CPUs, GPUs are another type of parallel processor that is usually present in modern computers. While they were invented for graphical computations, they can also be used for general purpose computations as well. The GPU is a highly parallel and many-core processor with great computational power in combination with high memory bandwidth. GPUs are designed to have more transistors for data processing instead of data caching and flow control which means they are specialized for highly parallel and intensive computations. Although GPUs were specifically designed for graphics rendering, it has become increasingly common to utilize them for general purpose computations. Figure 1 illustrates the difference in terms of hardware between CPUs and GPUs [11].

![Figure 1: Architectural differences between a CPU and a GPU](image1)

Its design makes a GPU ideal to handle data-parallel problems where the same computation is performed on each element of the data and not much control flow is needed [11].
Currently there are two major GPU providers, Nvidia and AMD. The first GPU was released by Nvidia in 1999 named Geforce 256 [12]. Much has happened for GPUs over the years since the release of GeForce 256 and in Table 2 a comparison between the currently most powerful GPUs released by Nvidia and AMD according to PassMark Software benchmarks [10] is shown.

<table>
<thead>
<tr>
<th>Release</th>
<th>Unit</th>
<th>Cores</th>
<th>Clock speed</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>Nvidia GeForce 256</td>
<td>4</td>
<td>120 MHz</td>
<td>Up to 128 MB @ 166 MHz 128-bit Memory interface</td>
</tr>
<tr>
<td>2016</td>
<td>Nvidia Titan Xp</td>
<td>3840</td>
<td>1582 MHz</td>
<td>12288 MB @ 11410 MHz 384-bit Memory interface</td>
</tr>
<tr>
<td>2016</td>
<td>AMD Radeon Pro Duo</td>
<td>8192</td>
<td>1000 MHz</td>
<td>8192 MB @ 500 MHz Dual 4096-bit Memory interface</td>
</tr>
</tbody>
</table>

Table 2: GPUs then and now [12, 13]

As Table 2 shows, the performance of the GPUs have increased dramatically since 1999, even more so compared to the difference in CPUs over time as shown in Table 1.

Although there is now access to these powerful multi-core processors, simply adding additional cores will not automatically make the software run faster. Significant effort must be put in to parallelize the software in order to achieve performance gains. In addition to this, a processor with 2 cores do not necessarily mean twice the performance. This is because the addition of cores introduces an overhead for the operating system (OS) to create and manage threads. In addition to this overhead, one of the bottlenecks to performance on a multi-core processor is simply that not all problems are parallelizable. Some problems may in fact run slower on multiple cores because of the introduced overhead by using multiple threads while the performance gains being too low. Therefore it is important to increase the processor utilization at the software level.

When evaluating a portion of software for its parallel potential, Amdahl’s Law [14] can be used to assess the potential speedup that can be achieved from parallel processing. The estimated speedup achievable from parallelization can be calculated using Amdahl’s law depicted in Equation 1.

$$Speedup(N)_{Amdahl's} = \frac{1}{(1 - P) + P/N + O_N}$$  \hspace{1cm} (1)

Where P is the parallel portion of the process and N is the number of processors. In addition to this, the overhead introduced from using N threads is noted as O. Applying Amdahl’s law in a case where the amount of parallel work is fixed will show diminishing returns from increasing the number of processors. Because of this, Gustafson revised Amdahl’s law for cases where the data set scales with the number processors used. Gustafson’s law is depicted in Equation 2 and can also be used to estimate potential speedup from parallelization [15, 16].

$$Speedup(N)_{Gustafson's} = (1 - P) + N * P$$  \hspace{1cm} (2)

The sequential fraction that limits the potential speedup from parallelization can reduce with the increasing number of processors, by keeping the sequential portions independent of the problem size. The results from the estimated speedup by applying Amdahl’s or Gustafson’s law can then be used to calculate the efficiency relatively to the ideal speedup by using Equation 3 [16].

$$Efficiency(N) = \frac{Speedup(N)}{N}$$  \hspace{1cm} (3)

These equations can be applied to a piece of software in order to evaluate its parallel potential. The results can then be used to decide whether it is worth parallelizing the software or not. In addition to this, to get close to these theoretical estimations the software will need to be parallelized effectively.
Due to the fact that all computations cannot be parallelized, software must be analyzed in order to locate which parts of the software have potential for parallelization. This can be done through both static and dynamic analysis of the code. The term static analysis means that the code is analyzed without running the software, for example manually or with a code analysis tool. Dynamic analysis is when the software is analyzed by executing it. Application profiling is a dynamic analysis which can be used to identify hot-spots in the software, i.e. code that consume much execution time. This is a commonly used technique when searching for optimization potential in code [17]. Once these hot-spots have been located, Amdahl’s law and/or Gustafson’s law can be used in order to estimate the potential speedup that the piece of software contains.

Within parallel computation, granularity can be explained as the amount of work performed by each task. Granularity can also be called grain-size of a task where fine-grained parallelism is when a program is broken down into a large number of small tasks. On the other hand, coarse-grain parallelism is when a program is split into larger tasks. Granularity can be measured in the number of instructions each parallel task executes but it can also be measured as execution time [16].

When it comes to parallel programming, there are two main types of approaches to approach a parallel problem: data parallelism and task parallelism. Task parallelism is when independent functional tasks are executed in parallel. The independent functionality is encapsulated in functions and then mapped to threads that executes the functions asynchronously [18, 16]. Data parallelism on the other hand, as previously mentioned, means that the same action is performed on different data [18, 16]. For example, when applying a filter to an image, the same task is applied for every pixel or group of pixels in the image. Task and data-parallelism can co-exist in the same application to solve different problems.

There are several different parallel computing Application Programming Interfaces (API) for expressing concurrency for CPUs and GPUs; a list of common programming models for parallel programming compatible with C++ is shown in Table 3.

<table>
<thead>
<tr>
<th>Parallel computing APIs</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ Thread support library</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Posix thread standard</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Win32 thread API</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CUDA</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OpenCL</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OpenACC</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 3: Examples of parallel computing APIs for expressing concurrency in C++

C++ Thread Support Library is a threading library available in the C++11 standard library [19] and allows developers to express concurrency through threads. Posix thread standard and Win32 thread API are similar to the C++11 thread library although with the drawback of being platform dependant to Unix and Windows respectively [20]. These parallel computing APIs are suitable to perform asynchronous tasks such as I/O and GUI updates that runs in the background of the main program [4].

Compiler directed parallelism such as OpenMP [21] is also a programming model to express concurrency on the CPU, however, it differs compared to the previously mentioned threading libraries in the way concurrency is expressed. The benefit of OpenMP is that the sequential code will not have to change in many cases and if the compiler do not support it, the directives will simply be ignored.
When it comes to parallelizing GPUs there are additional things to consider in comparison to CPU parallelization. One of the factors limiting potential speedup from parallelization on GPUs compared to CPUs, is that the GPUs usually do not share the same memory space as the CPU. This means that the data required for computations needs to be transferred to the GPU before starting the computations. In addition to this, the results must then be transferred back from the GPU to the CPU memory space. This introduces an additional overhead for parallelization on GPUs and may therefore limit the benefits of utilizing them.

CUDA is the parallel computing API invented by Nvidia which enables the developer to utilize the GPU for general purpose computations. Introduced in 2006, CUDA is a general purpose parallel computing platform as well as a programming model for Nvidia GPUs. The compiler NVCC, also developed by Nvidia, is used to build code for the GPU using CUDA. The CUDA code is built by NVCC while the CPU code, or host code in CUDA terms, is built by a compiler of choice such as the VC++ or GCC compilers [22].

Open Computing Language (OpenCL) is an open standard maintained by The Khronos Group [23]. Similar to CUDA it can be used to express parallelism on a GPU, however, OpenCL is not bound to a specific platform. Instead, OpenCL is a cross platform language able to express parallelism on both AMD and Nvidia GPUs as well as other platforms such as FPGAs.

OpenACC is another parallel computing API that utilizes compiler directives similar to OpenMP in order to express concurrency. OpenACC differs from the other approaches since it supports both CPU and GPU acceleration, allowing for a very portable code.
4 Related Work and Comparison to our Contribution

There has been much research within the field of parallel computation over the years. Since the contribution of this work is a method to re-engineering sequential software to introduce parallelization, the studied related work focuses on other development approaches or methodologies for parallel software rather than programming guides or generic software development methods.

In 2003 Intel [18] presented a method to introduce threading in sequential software using a generic development lifecycle consisting of six different phases. These phases are 1. Analysis, 2. Design, 3. Implementation, 4. Debugging, 5. Testing and 6. Tuning. The lifecycle is of a waterfall [24] nature with the exception of debugging, testing and tuning. These steps are intended to be performed iteratively until errors are found and fixed. The drawback of their method is that it targets developers using Intel threading tools only to introduce parallelization. Additionally, the parallelization is only considered for CPUs which limits the hardware utilization in systems containing both CPUs and GPUs.

The work presented by Tovinkere [16] is greatly similar to the approach presented by Intel 2003. The development lifecycle is also a general development lifecycle that has six different phases, although they differ slightly compared to the one in [18]. The steps in their method are as follows: 1. Analysis and Design, 2. Implementation, 3. Debugging, 4. Performance Tuning, 5. Re-mapping and 6. Unit Testing. Similar to the method presented in [18], the goal is to utilize threading to increase the performance of software. In comparison to Intel’s method [18], Tovinkere’s is more specific to parallel software development whereas Intel’s follows a general software development method. Tovinkere’s approach is less tool-dependent, although when applying it to a specific case they suggest Intel tools too. Both methods are specific to threading on CPUs and only consider the technical aspects of the development.

Jun-feng [20] provides an approach that consists of seven different steps, in which some of them also consist of several substeps. Their model is following a waterfall development cycle. During the initial phase of the development cycle they briefly mention financial costs in the form of hardware investment as well as development cost. The approach does not consider locating parallel potential in any way and initially assumes that there is code to perform a feasibility analysis on. Their model is also limited to parallelization on CPUs.

Christmann et al. [25] present a method for porting existing sequential software to a parallel platform. Their method takes into account financial aspects of the development costs of the parallelization of the existing software. Their approach consists of four different phases which are preparation, analysis, implementation and adaption. In their study the method is applied on a smaller program for evaluation, however, their results suggest that the method is more suitable for bigger programs.

The method proposed by [26] is based upon an iterative approach they call Assess, Parallelize, Optimize, Deploy (APOD). The idea behind this approach is to be able to deploy a parallelized portion as soon as possible to quickly create value for the users. In contrast to previously mentioned related works, this methodology considers parallelization on GPUs. However, this is also one of the drawbacks with their approach since it only targets parallelization on the GPUs.

The method presented in [27] proposes an iterative approach to introducing parallelization to legacy software. Their iterative process contains five different steps: Save current version, change, check new version, accept or reject and finally document. To introduce parallelization in to legacy Fortran software they propose the use of OpenMP and only consider CPU parallelization.
One of the major differences of the methods presented by [18, 16, 20, 26, 27] in comparison to ours is the consideration of effort estimation for the parallelization. During the analysis our method includes an effort estimation for the parallelization, which is further described in Section 6.2. This gives the opportunity to prioritize parallel potential portions of the software with higher speedup potential. This can potentially mean that a higher speedup can be achieved within a set budget of development effort. On the other hand, our parallelization method therefore introduces a slight extra overhead work to do this estimation. However, not taking this into consideration can potentially result in spending effort on less valuable parallelization potential regions. At the same time for our method, the extra overhead may be introduced for parallel potential regions that are not implemented in the worst case scenario. Because of this extra overhead, our method is potentially better suited for larger existing software codebases where this overhead would represent a smaller fraction of the total parallelization effort.

Another difference between the method presented by [18, 16, 20, 25, 26, 27] is that our method considers both CPU and GPU parallelization and theirs considers parallelization on either CPUs or GPUs only. Additionally, the method proposed by [26] is specific to CUDA enabled hardware while the methods proposed by [18, 16] targets Intel hardware and tools. This would also mean that the hardware is under-utilized if these methods are used in systems where both CPUs and GPUs are available. Instead, the method we propose is more generic and does not limit itself to specific hardware or tools and considers parallelization on CPUs and GPUs. However, since their methods are specific to certain hardware and tools, they are able to give more advice regarding implementation, potentially yielding more effective speedup on the specific hardware they are limited to when the suggested tools are available.

While there exists several different methodologies to the development of parallel software, there is still a risk of under-utilization of the available hardware since they do not consider parallelization on both CPUs and GPUs. Besides this, there is also a risk of the development cost running out of control if financial aspects of the development are not considered. Therefore, there is a need to consider technical aspects of parallelization, meant as utilizing both CPU and GPU, as well as the financial aspects of this parallelization process. This work addresses these two points.
5 Research Method

In this study the methodology used in order to approach the problem was the engineering design process [28], defined by the following seven steps:

1. "Define the Problem"
2. "Do Background Research"
3. "Specify Requirements"
4. "Create Alternative Solutions"
5. "Choose the Best Solution"
6. "Develop the Solution"
7. "Test and Redesign"

Before the work could start, the first step was defining the problem that needed to be solved. In order to tackle the problem described in Section 2 the goal of this study was to define a method for introducing parallelization to existing software.

With this goal in mind, the next step of the process was to do background research regarding state-of-the-art solutions to similar problems in order to gain knowledge of existing methods for parallelizing sequential software. This was achieved through an informal literature review searching relevant databases such as IEEE, ACM, Springer and Google Scholar [29, 30, 31, 32]. These databases were chosen because they include publications within the topic of computer science and the availability for the author. The purpose of this literature review was to gain knowledge on the following:

- Existing methods for parallel software development
- Introducing parallelization to existing software
- Localization of parallel potential in software
- Utilization of CPU and GPU for parallelization of software

With this in mind, keywords such as "Parallel Software", "Parallelization", "Software engineering method", "Multi-core CPU" and "GPU" were used. Papers were selected based on their relation to the topic of parallel software development and engineering. Although this study focuses on the introduction of parallelization into existing software, literature focusing on the development of parallel software was included. This was because literature focusing only on parallelization of existing software was limited. An analysis and comparison of the existing parallelization methods was performed in order to find their strengths and weaknesses. This was done by performing a Strengths, Weaknesses, Opportunities and Threat (SWOT) analysis [33].

The next step was to specify the requirements of the parallelization method. The identified strengths and weaknesses of existing methods were summarized and analyzed. Requirements were defined with the goal to inherit these strengths and at the same time avoid introducing weaknesses to the method. These requirements along with the initial goals of the study were combined into the requirements specification for the parallelization method.

The design of the parallelization method was started and the creation of alternative solutions was done by sketching down different ideas for a solution, based on the requirements defined in the previous step. Alternative methods were divided into different configurations of steps and phases. Once an idea for a solution was created, it was analyzed for strengths and weaknesses. In the end, the choice was between two fairly similar options but with one difference, further discussed in Section 8. Once one of these solutions were selected as the best solution it was taken to the
development phase, as described in Section 6.

Once the parallelization method was defined, the next step was to test and evaluate it. In several of the related works the evaluation was done by applying their method on smaller programs or in experimental environments [25, 34]. The drawback with doing this is that conclusions cannot be drawn for how the method performs when applied to larger systems in a real world environment. Instead, in this study the parallelization method was evaluated by applying it to an existing real-life industrial system. The details of the design of this process is described in Section 7.

Additionally, the developed method was also compared to existing methods. The data collected from the SWOT analysis of the state-of-the-art was compared to the data retrieved from the experiment in order to evaluate our parallelization method. This was done in order to show benefits and drawbacks of the derived parallelization method. This comparison is described in Section 4 and can be used to decide which parallelization method fits better on a case-by-case basis.
6 Results

In this section we present the thesis contribution. Section 6.1 starts with an analysis of the existing parallelization methods. Based on this analysis a set of requirements followed by the definition of the parallelization method is presented in Section 6.2.

6.1 Analysis of existing methods for parallelization

In this section we present existing parallelization methods as well as a SWOT analysis on each of them. This was done with the purpose of finding their strengths and weaknesses in order to refine the requirements for the parallelization method presented in this study.

As mentioned in Section 3, Intel presented a development method to introduce threading in software to increase performance in 2003 [18]. The Strength with this method is that the development methodology is simple to understand. However, it has a structure similar to the waterfall model [35] and therefore inherits its weaknesses. One notable Weakness of the method is that changes to requirement or discovering errors late in the development can be very costly [24]. In addition to this, the method targets users of specific hardware and tools (Intel processor and tools) only. Although, in the case where the specific hardware and tools are available, the methodology provides Opportunities to achieve better performance. At the same time, it does not consider utilizing ways of introducing parallelization other than through threading on CPUs, therefore there may be cases where this method does not utilize the full potential of the hardware which could be seen as a Threat.

In the method presented by Tovinkere [16] the Strength is that it is more specific to parallel software development in comparison to [18], but still remains simple to understand. The Weakness of this method is also that it targets specific hardware and tools only. Moreover, it does not consider financial aspects in terms of development costs of introducing parallelization into software. There is an Opportunity to fully utilize suggested tools to gain optimal performance from the software. Since the method only applies to specific tools and hardware, there is a Threat that the software lacks in portability. In addition to this, the method does not consider GPU parallelization which can potentially limit the performance gains.

In [20] the authors present a method whose Strength is that it takes the feasibility of a parallelization of a piece of software into account. On the other hand, its Weakness is that it does not consider locating parallel potential nor validating the software after the parallelization. Additionally, it only considers parallelization on CPUs. The Opportunity is that, due to the fact that the method only focuses on technical aspects of the parallelization it produces low overhead. The Threat with using this method is that the software may contain errors since software testing is not considered as part of the method.

Christmann et al.’s methodology [25] has the Strength that it considers the financial aspects of developing software. The Weakness of the method is that it does not consider GPU usage for parallel computation. An Opportunity due to the fact that this method considers financial aspects of development is that the costs can be kept under control. On the other hand, this also introduces additional overhead which may be seen as a Threat in cases where the overhead outweighs this benefit.

The Strength with the method proposed by Nvidia [26] is that it uses an iterative approach to development which allows for continuous deployment and considers the use of GPUs for parallelization. The Weakness is that it does not consider financial aspects nor utilization of CPUs and instead focuses only on optimization on GPUs. In addition to this, the method is also limited to parallelization on CUDA-enabled devices. The Opportunity with this approach is that since GPUs carry much computational power the potential for greater performance increases compared to limiting the parallelization to CPUs. Because the method is limited to GPUs, as well as limiting itself to CUDA enabled GPUs, its target audience is severely limited which poses a Threat when
the software needs to be available on multiple platforms with different hardware configurations.

In [27] the authors present a method whose Strength is that the iterative method focuses on managing changes to already working software without breaking it. The Weakness of the method is that it is limited to Fortran code and the use of OpenMP, meaning that a GPU would not be utilized if available. However, the Opportunity is that speedup may be achieved while maintaining the validity of the software, if the legacy is written in Fortran. Since parallelization on GPUs are not considered, there is a Threat that the hardware may be under-utilized.

A summary of some strengths and weaknesses identified in existing methods from the SWOT analysis is shown in Table 4

<table>
<thead>
<tr>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple to understand</td>
<td>Waterfall model</td>
</tr>
<tr>
<td>Feasibility analysis</td>
<td>Targeting specific tools</td>
</tr>
<tr>
<td>Considers financial aspects</td>
<td>No parallelization on both CPUs and GPUs</td>
</tr>
<tr>
<td>Iterative process</td>
<td>No validation</td>
</tr>
<tr>
<td></td>
<td>No parallelization potential localization</td>
</tr>
</tbody>
</table>

Table 4: Located parallel potential locations

From the results of this analysis it is clear that only one of the analyzed methods considers financial aspects in terms of additional development costs that parallelization introduces. Considering that developing parallel software is a more complex task than to develop sequential software [4], there is a risk that the parallelization of sequential software comes with a high cost. Effort of introducing parallelization is therefore an important consideration in order to keep the costs under control. In addition to financial aspects, the use of the GPU for parallel computation is also considered by one method only. At the same time there is no method that considers both parallelization on CPUs and GPUs together.

6.2 Definition of the parallelization method

Based on the analysis in the previous section the following set of requirements on the parallelization method was defined. The method should:

1. Consider the use of both CPUs and GPUs for parallelization
2. Consider effort required to parallelize sequential software
3. Include validation of the parallelized software
4. Provide an iterative development process
5. Not target specific tools

1. and 2. are direct consequences of the aforementioned research challenges and represent the novelty of this contributions since, as shown in the previous section, no existing approach provides both. 3. Validating that the parallelization does not introduce errors or unexpected behaviours must be part of the parallelization method, since it is instrumental in any software engineering process. 4. The parallelization method should provide an iterative development approach to incrementally parallelize software portions and continuously validate results. Additionally, this could also allow for continuous integration and deployment. 5. The parallelization method should not target specific tools, in order to be applicable by an as broad audience as possible.
The parallelization method that we propose consists of three different phases and is depicted in Figure 2. The first phase represents the definition of the goals of the parallelization as well as the tools to use. In the second one, the analysis phase, developers analyze the software for parallel potential and perform a feasibility analysis. The last phase is the development phase, where the feasible portions of the software are further analyzed and a parallel solution is designed, implemented and validated.

![Parallelization method diagram](image)

Figure 2: Parallelization method

In the following subsections a more detailed description of the phases is presented.

### 6.2.1 Specification phase

In this phase, the engineer specifies the goals of the parallelization and decides which tools to use. As shown in Figure 2, the first step is to decide what the goals of the parallelization are. The goals can be divided into two categories, financial and technical goals. In this method the financial goals are expressed in terms of development effort introduced by the parallelization. However, there are cases where not only the parallelization of the software itself affects the costs. For example, in some cases the parallelization may introduce increased costs in terms of energy consumption. In cases where such aspects are vital, these other forms of financial aspects and how they affect the
parallelization process must also be taken into consideration.

Technical goals can be expressed in terms of desired speedup as result of the parallelization. According to [25] there are two different approaches to choose between: one is to prioritize the cost before the performance while the other is to prioritize the performance above the cost. The first approach can be described as parallelizing a piece of software as much as possible within a certain budget, e.g. the development cost. Once this budget is used the parallelization stops no matter how much the performance has improved. The other is to set a target level of performance that the software needs to reach. The parallelization is carried on until this target is reached. In many cases the first option may be preferred since a "good enough" solution is more likely to be more cost effective. However, in some cases where there are timing restrictions that require a certain level of performance, the second option may be needed.

The next step is to specify what kind of tools to use for the parallelization. This step is very dependent on what the existing software’s characteristics are. For example, if the existing software is written in C++, then tools that support C++ have to be used. Although, if it would be beneficial to first modify the existing software in order for it to be used on a specific unsupported tool, it is recommended to do so. However, this will have to be decided on a case to case basis and is not covered by this work. One type of tools that should be chosen during this step is a profiling tool, to be used during the next phase to aid with parallel potential location. In addition to this, a timer to measure the execution time of portions of the software will be used for the feasibility analysis in the next phase. Additionally, in this phase the engineer selects which parallel computing APIs to use. One type of tools that should be chosen during this step is a profiling tool, to be used during the next phase to aid with parallel potential location. In addition to this, a timer to measure the execution time of portions of the software will be used for the feasibility analysis in the next phase. Additionally, in this phase the engineer selects which parallel computing APIs to use. This decision is also affected by the existing software, for example which platform it is intended for, used programming language and compiler. Since our method is meant to be generic and applicable to a wide variety of existing software, this choice also needs to be taken on a case to case basis.

6.2.2 Analysis phase

The purpose of this phase is to analyze the software to find potential for parallelization. Two techniques can be used to perform this task: manual and dynamic analysis of the software. In some cases, the parallelization is not necessarily performed by someone that has an in depth knowledge of the details of the complete software. In such cases it can be beneficial to perform additional manual analysis of the software to gain a deeper program understanding. Analyzing the control flow of the software can give additional program knowledge as well as revealing task-parallelism potential in the program flow. Manual inspection of the software can be a time consuming task, therefore, using dynamic analysis can help to point out where in the software to look for parallel potential. A profiling tool can be used to search for pieces of the software where much of the execution time is spent, i.e. hot-spots. The idea behind searching for hot-spots is that if these portions of the software are parallelized, the total gain of performance is greater than parallelizing a portion that is rarely used. Once this analysis is done and a set of hot-spots is defined, a manual inspection of the code should be performed. This is done with the purpose of deciding whether there is any potential for parallelization in that piece of software. Parallel potential is found by looking for computations that are not dependant of each other and can therefore be performed in parallel.

When locations for parallel potential have been found, it is time to do a feasibility analysis on them. The purpose of this is to find out if it is worth performing the parallelization on the piece of software. The first step is to evaluate how much of the code can be parallelized. Once this is defined, an estimation of the potential theoretical speedup from parallelization can be calculated by using Amdahl’s law or Gustafson’s law, as described in Section 3. Since these laws expect a number of processors to apply the parallelization on, a decision of whether to utilize CPUs or GPUs for the parallelization must be taken.
To decide whether CPUs or GPUs (or both) shall be employed, we first need to check whether a task-parallel or data-parallel approach is more appropriate. In the case where a task-parallel approach is selected, then CPUs are used for the parallelization. By considering the number of processors of the target hardware when applying Amdahl’s or Gustafson’s law we can estimate the potential speedup from parallelization.

On the other hand, if we decide to go for a data-parallel approach, an estimation of the potential speedup in case of CPUs or GPUs must be done in order to choose between them. Apply Amdahl’s or Gustafson’s law first targeting a CPU and then a GPU. Characteristics of the GPU, such as clock frequency and memory clock frequency, may differ from those of the CPU, this must be taken into consideration when analysing the results of speedup estimation. In addition to this, the grain-size of the parallel task plays a role in the selection of the processing unit type too. Since GPUs often have several hundreds and in some cases even thousands of cores, the parallel task should be fine-grained enough to utilize these cores. If the task is coarse-grained and unable to utilize the available cores, the GPU would be under-utilized; then it would be better to use a CPU instead. If the task is suitable for both CPU and the GPU, the results from the speedup analysis decide of whether to utilize the CPU or the GPU.

The second step of the analysis phase is to estimate how much it will cost to go through with the parallelization. There are three things that needs to be estimated: the time to design a parallel implementation, the time to implement the parallelization and to validate the parallelization. First, analyze the code size and complexity and estimate how much time it will take to design a parallel implementation. Next, analyze the code to see if there is a need for refactoring the existing sequential code in order to go ahead with the parallelization. Estimate the time it would take to refactor the code (if necessary) and implement a parallel implementation. The final step is to estimate how much effort the validation will take. Once these estimations are made they can be combined into a total estimation of the required effort to design, implement and validate the parallel potential piece of software.

When these two values are defined for a piece of software they can be combined to see how feasible it is for parallelization. If the potential gain is very small and at the same time the estimated cost is high, a decision must be taken whether it is worth even attempting to parallelize it or not. There can even be cases where the overhead introduced by the parallelization is greater than the performance gain, in this case the resulting software would turn out to be slower than the original. In this case, the choice to not parallelize this piece of software should be taken.

Once the feasibility analysis has been applied to all located parallel potential locations, it is time to prioritize them according to the available budget.

6.2.3 Development phase

This is the last phase of the parallelization method and it is performed iteratively. The following steps are applied to all chosen pieces of software selected for parallelization starting with the one set to the highest priority.

The first step of the development phase is to design how the selected piece of software should be parallelized. In the analysis phase an initial design decision was taken regarding which hardware platform to utilize. If the chosen hardware was the CPU the problem is either of a task-parallel or data-parallel nature. In the case of a task-parallel problem it is recommended to use a threading library such as the C++ Thread Support Library [18, 16, 36]. On the the other hand, when it comes to parallelize data-parallel problems on the CPU, the recommendation is to utilize a compiler directed approach such as OpenMP [18, 16, 20, 26]. If the chosen compiler does not support such a technology, a threading library can be used as an alternative. If the chosen approach was to utilize the GPU for the parallelization, we can simply use the parallel computing API selected in the specification phase.
Additionally, the piece of software selected for parallelization must be further analyzed in order to design the implementation of the parallelization. In some cases refactoring of data may be necessary in order to allow for better parallelization.

The implementation and validation step of the process was decided to be combined into one step. The implementation approach is freely chosen by the engineer since it does not affect the expected gains.
7 Evaluation

In order to evaluate our parallelization method, we set up an experiment exploiting a real-life industrial software system. The results from this evaluation were then used to compare the method to the other existing parallelization methods as described in Section 4. The goal of the experiment was to apply the parallelization method on an existing sequential software to introduce parallelization, as a proof-of-concept.

At ABB Robotics a topic that is considered quite extensively at the moment is 3D sensors and what they can bring regarding awareness of the environment for the robots. 3D sensors using structured light have opened up new possibilities for robotics and the provided 3D data can be used to describe a robot’s surroundings in more detail. That is, this kind of sensor can be used when scanning an environment in order to determine collision free paths in the 3D space. This effort has been going on for some time and sequential software has been developed yielding good performance regarding perception. The problem is that the execution time is too high and that today’s single computer cores are inadequate for the task. In addition to this, today’s 3D sensors don’t have any additional processing capabilities for vision algorithms, making the data acquisition slow. Therefore, to be able to obtain real-time behaviour of a 3D vision system, a processing device with higher computational power must be placed near the vision sensor. As previously mentioned in Section 3, GPUs are designed for highly parallel computation and leading graphic card vendors are currently offering small circuit boards with integrated processors and GPUs targeted to embedded environments [37, 38]. However, to run the existing software on such a platform, it needs to be re-engineered in order to introduce parallelization and thereby exploit the potential of parallel hardware.

In order to know what kind of data had to be collected during this experiment, a set of research questions was defined as follows:

1. How much overhead does the parallelization method introduce?
2. How much did the execution time of the parallelized software improve from the original?
3. Are there any limitations in using the parallelization method?

Data was collected in multiple ways, combining quantitative and qualitative measurements. In order to answer the first research question, the effort spent on each step of the approach was recorded with the purpose of tracking how much time is spent on preparation and planning compared to the actual implementation of the parallelization itself. Benchmarks was performed on the chosen pieces of software for parallelization before and after the parallelization method was applied. The data recorded from these benchmarks was used in order to answer the second research question. Additionally, we evaluated the parallelization method after the experiment with the purpose of collecting qualitative data. This was done in order to gain an understanding of how the experience of using the parallelization method was perceived by those applying it, with the goal of answering the third research question.

7.1 Experiment

The experiment was carried out at ABB Robotics. The parallelization method was applied on a palletizer application that is developed as a proof-of-concept for a 3D vision library at ABB Robotics. The application is quite simple in itself and the objective is to use a robot arm to pick and place objects from point A to point B. In this particular application the exact location of point B is unknown beforehand and only a larger area of where B’s location is, is known. The same goes for the location of the objects, whose exact position is unknown. In order to identify the location of the placing area (point B) as well as position and orientation of the objects to be moved, the application uses the 3D vision library. The control flow of the application is shown in Figure 3.

The first thing to do before the application can start using the robot arm to pick and place items is to model the static environment around itself using a 3D camera that uses structured light. The robot arm moves around and captures multiple images from different angles around itself and
stitches the images together. The static environment is then stored as a point-cloud [39] in the coordinate system of the robots base. Now that the static environment is modeled and stored, the first step of the main loop is to move to the general area where the placing area is and capture an image with the 3D camera. Using the stored model of the environment, everything except the placing area can be extracted from the point-cloud using a background extraction algorithm from the 3D vision library. The application is now aware of the exact location of the placing area and can move the robot arm to the picking area. An image is captured once the robot arm is above the picking area in order to locate objects to pick and place. The static environment can be extracted from the captured 3D image, in the same manner as it was done for the placing area, in order to remove everything except replaceable objects. The application now uses a 3D model in the form of a point-cloud to represent objects and locates them using another algorithm from the 3D library, which gives the objects location and alignment. Once the location and angle by which the robot should pick up the object have been identified, the robot arm can pick the object up and place it in the desired alignment on the placing area. This loop can now be repeated in order to pick and place the remaining objects.

Figure 3: Control flow of the Palletizer application
The issue with this sequential application is that the vision algorithms are very computational heavy. Once an image is taken, the application needs to wait for the algorithms to finish before it can move the robot arm again. This idle time increases the cycle time of the program and therefore the throughput is low. Our experiment consisted in applying our parallelization method to this application in order to increase its overall performance. In the following subsections the application of the parallelization method is described.

7.1.1 Specification phase

To specify the goals of the parallelization, a speedup maximization approach was used, meaning that the goal was to maximize the speedup within a given budget. The budget for the parallelization is specified in terms of time representing development effort. The budget for the development was set to five weeks of 40 hours each week, giving a total development time budget of 200 man hours. Besides maximizing speed-up, an additional goal was to make the parallelized software runnable also on platforms not including GPUs. This means that if the parallelization of a function was implemented on a GPU, the same function should also be parallelized on a CPU in order for the parallelization to yield speedup on platforms of different kinds. Additionally, this would allow a comparison between the results from parallelization on CPUs and GPUs in terms of development effort and achieved speedup. The decision between which implementation to use is taken at runtime depending on the platform configuration.

The development as well as the benchmarking was carried out using a Lenovo P50 laptop with Windows 7 as operating system. The specification of the hardware of the computer is specified in Table 5.

<table>
<thead>
<tr>
<th>Lenovo P50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>GPU</td>
</tr>
</tbody>
</table>

Table 5: Development and benchmarking hardware

The existing code base was developed using Visual Studio 2015 (v140 platform toolset) and Microsoft’s VC++ compiler and the code was compiled using the -O2 optimization flag. The C++ threading library and OpenMP was used in order to express concurrency on the CPU when applying the parallelization method. This was decided since the existing codebase was already using the C++ standard library and the compiler also supports OpenMP. The intention was to use C++11 threading library to express task-parallelism while OpenMP for data-parallel tasks. In order to express concurrency on the GPU, we used CUDA. This decision was based on the fact that the target system includes a CUDA enabled GPU. Time spent in this phase was 8 hours.

7.1.2 Analysis phase

As shown in Figure 2, the first step of the analysis phase is to locate a set of software pieces that could potentially be parallelized. In this experiment, the first step was performed through both a manual control flow analysis of the application and a dynamic analysis. The manual analysis was done to get a better understanding of how the application worked and to locate parallelization potential in the program flow. A high-level control flow of the application is depicted in Figure 3. Based on the control flow and dependencies between the actions, there were two visible locations for parallelization. The first one is in the environment modeling procedure, where there is no dependency between moving the robot and adding the point-cloud from the image to the point-cloud representing the environment. There is therefore potential for parallelization in this case, where the robot could move to the next position at the same time as the application models the environment. The second one is when a picture is taken of the placing area and the placing position is calculated, after which a collision free path is planned to the picking area. This path planning is not dependant of the result of the placing area image results, and so the actions could
actually be run in parallel.

In addition to the manual control flow analysis, a dynamic analysis was performed using Visual Studio Profiling Tools with the intention to find hot-spots in the code. The result of this analysis showed that many of the hot-spots were located inside third party libraries used by the software. However, since the goal was not to parallelize the third party libraries, the hot-spots located outside the target software were ignored. The result showed 13 functions, apart from the third-party functionality, that were accessed frequently and consumed much computational power. These functions were then inspected and analyzed manually for parallelization potential. After the analysis, 6 functions were identified as parallelizable. Table 6 summarises the located software portions with parallelization potential.

<table>
<thead>
<tr>
<th>Parallel potential</th>
<th>Location method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model environment</td>
<td>Manual analysis</td>
</tr>
<tr>
<td>Locate placing area</td>
<td>Manual analysis</td>
</tr>
<tr>
<td>Extract environment</td>
<td>Dynamic profiling</td>
</tr>
<tr>
<td>Align object</td>
<td>Dynamic profiling</td>
</tr>
<tr>
<td>Edge detection</td>
<td>Dynamic profiling</td>
</tr>
<tr>
<td>Stitch point-cloud</td>
<td>Dynamic profiling</td>
</tr>
</tbody>
</table>

Table 6: Located parallel potential locations

Once these locations of parallel potential software were located, we conducted a feasibility analysis on them to check whether they were actually parallelized.

Modeling the environment was the first potential location for parallelization and was found through the manual analysis of the software. This part of the code takes care of combining multiple images taken from the robot arms perspective. As shown in Figure 3, the robot arm moves to a set of positions and takes an image at each position. In this case, the potential for parallelism here is to add the previously taken image to the environment at the same time as moving the robot arm to the next position. The parallelization of this function would run on two parallel cores. This is because there are only two tasks that can run in parallel, meaning that it cannot scale to additional threads. In order to estimate the potential speedup achievable through parallelization, we used the following procedure: First, the total time it took to model the environment was measured. The total execution time for this was measured to 9822 milliseconds. Then, the individual times of the two tasks that could run in parallel was measured. For adding new point-clouds to the environment this was measured to 2816 milliseconds while moving the robot arm was measured to 6635 milliseconds. The remaining 371 milliseconds was spent on communication with the image sensor and the robot controller, which needed to be performed sequentially. Since adding images to the environment can be performed in parallel to moving the robot arm which is a longer process. Meaning that in theory, 5632 milliseconds (2*2816) of the sequential execution time could run on different cores at the same time. This is 57.34% of the total execution time. The results of applying these numbers into Amdahl’s law is shown in Equation 4.

\[
\text{Speedup}^{(2)}_{\text{Amdahl's}} = \frac{1}{(1 - 0.5734) + 0.5734/2} = 1.4
\]  

While the estimated speedup factor of 1.4 may not seem large, the execution time of modeling the environment is measured to 9822 milliseconds in total. In Equation 5 we apply this speedup factor to the measured execution time to estimate what the speedup would translate to in terms of time.

\[
\frac{9822}{1.4} = 7015.71
\]  

The result from this speedup estimation suggests that if this theoretical speedup could be achieved through parallelization, the execution time would be 2806.29 milliseconds faster than the
sequential code.

Locating the placing area was also found as a potential location for parallelization through manual analysis. In this case there were two different tasks performed sequentially that were not depending on each other: locating the placing area and calculating a collision free path to the picking area. The total execution time for these two tasks were 2205 milliseconds. Individually the tasks measured at 493 milliseconds for locating the placing area and 1712 for calculating the path to the picking area. It is therefore estimated that 44.72% of the code can run in parallel since 986 milliseconds \((493 \times 2)\) of the execution time can run on two separate cores. Equation 6 shows the potential speedup factor from applying these numbers using Amdahl’s law.

\[
Speedup(2)_{Amdahl's} = \frac{1}{(1 - 0.4472) + 0.4472/2} = 1.29
\]  

(6)

The estimated speedup factor achievable through parallelization was 1.29. In order to understand what this potential speedup translates to in terms of time, we applied these numbers in Equation 7.

\[
\frac{2205}{1.29} = 1709.3
\]  

(7)

These results also show a fair potential for speedup in terms of time. Since the total execution time is 2205 milliseconds, a parallelization that achieves this speedup would reduce the execution time by 495.3 milliseconds.

Extracting the environment from an image was another software piece with potential for parallelization. The sequential execution time of this function was measured to 295.4 milliseconds. An analysis of this piece of software showed that it contained code that could potentially cause race conditions if executed in parallel, by reading and writing to the output of the function. This code must therefore run sequentially in order to avoid corrupting the output. Additionally the execution time of this was measured to 4.14 milliseconds. Additionally, computations performed on the code that needs to run sequentially was measured at 4.14 milliseconds while the parallelizable portion of the function takes 291.26 milliseconds. This means that 98.6% of the execution time can run in parallel. The next step was to decide which parallelization approach to use. To do that, we first needed to classify the problem as task-parallel or data-parallel. In this case, the problem was classified as data-parallel because in order to extract the static environment from a point-cloud, the algorithm checks for every point in the point-cloud whether it exists in the environment or not. At this point, the speedup estimation was performed for both CPUs and GPUs according to the parallelization method. Equations 8 and 9 shows the results when applying these numbers to Amdahl’s law.

\[
Speedup(4)_{Amdahl's} = \frac{1}{(1 - 0.986) + 0.986/4} = 3.83
\]  

(8)

\[
Speedup(512)_{Amdahl's} = \frac{1}{(1 - 0.986) + 0.986/512} = 62.79
\]  

(9)

Note that the estimated speedup in Equation 9 is the theoretical speedup in the case when the sequential version is executed on the GPU. However, the GPU used in this experiment has a core frequency that is only 27.7% of the CPUs core frequency. The estimated speedup from parallelization using the GPU based on the difference in core frequency would be 16.5 \((62.79 \times 0.277)\). This means that in theory parallelizing it on the GPU would give a better speedup. Additionally, we analyzed the code to see how many computations may run in parallel to know if parallelization on GPUs is suitable. This analysis showed that 307200 (the size of the point-clouds) calculations can be performed in parallel, which means there are enough parallel computations for the 512 cores available on the hardware. For this specific experiment, as mentioned in Section 7.1.1, functionality should also be parallelized for a CPU in the case where a GPU parallelization approach is chosen.
Aligning object to find location and orientations of them inside a point-cloud was found as a location for parallel potential through dynamic analysis. Measuring this function showed an execution time of 1548.54 milliseconds. Similar to the previous piece of software, this function also contained code that had to run sequentially. The sequential portion of code was measured at 18.73 milliseconds and is therefore 1.2% of the whole execution time. This means that 98.8% of the code was parallelizable. Additionally, after analyzing this piece of software it was decided that a data-parallel approach was appropriate. The same action is taken for every data in the task of aligning a point-cloud in another and calculating the amount of points that matches. We estimated speedup by applying Amdahl’s law as shown in Equations 10 and 11.

\[ Speedup(4)_{Amdahl's} = \frac{1}{(1 - 0.988) + 0.988/4} = 3.86 \]  
\[ Speedup(512)_{Amdahl's} = \frac{1}{(1 - 0.988) + 0.988/512} = 71.79 \]  

The speedup estimations shows that aligning an object in a point-cloud has a good potential for speedup through parallelization. While the number of computations that can run in parallel in this case depends on the size of the point-clouds provided as input, we decided that this is a suitable piece of software to parallelize using GPUs. This is because the number parallelizable computations are usually in between 100000-500000. This would give the GPU in our benchmarking computer enough parallel work to utilize the available 512 cores. Additionally, even though the frequency of the GPU cores are only 27.7% of the CPUs core frequency, it is still estimated that the parallelization on GPUs would yield a better speedup.

Edge detection is used to locate edges in point-clouds. This functionality was measured and the execution time was 32.73 milliseconds. The reading and writing to the output was needed to occur sequentially and this was measured at 2.42 milliseconds. Meaning that 92.6% of the computations can be run in parallel. After further analysis, it was decided that this should be classified as a data-parallel problem, since the same action is performed for every point in a point-cloud to decide whether it is an edge point or not. However, the size of the point-clouds provided as input are usually around 200 points when used in this algorithm since the point-clouds are down-sampled. Since we have 512 cores at our disposal, more than half of those cores would have no computations to execute. This would mean that the GPU is severely under-utilized, therefore, the speedup estimation and parallelization should be done for CPUs and the result is shown in Equation 12.

\[ Speedup(4)_{Amdahl's} = \frac{1}{(1 - 0.926) + 0.926/4} = 3.27 \]  

From these results we can see that the potential speedup factor is 3.27 and makes this a good candidate for parallelization.

Stitch point-clouds is the process of combining two point-clouds into one and execution time was measured at 248.1 milliseconds. Further analysis of the code showed that the reading and writing data to the point-cloud had to run sequentially. This portion of the code was measured and the execution time was 54.09 milliseconds, meaning that 78.6% of the computations could run in parallel. Stitching point-clouds was also defined as a data-parallel problem since the same action is performed for all points in the point-clouds. These numbers were used in combination with Amdahl’s law in order to estimate the potential speedup, the results are shown in Equation 13.

\[ Speedup(4)_{Amdahl's} = \frac{1}{(1 - 0.786) + 0.786/4} = 2.44 \]  
\[ Speedup(4)_{Amdahl's} = \frac{1}{(1 - 0.786) + 0.786/512 + 0.1} = 4.64 \]
The results from the estimated speedup shows a decent potential for parallelization. While Equation 14 shows better potential for speedup for a GPU parallelization, the difference in the hardware must be taken into account. Since the frequency of the GPU cores are only 27.7% of the CPUs frequency in this experiment, we expect that the speedup achievable from parallelization on CPUs will be higher. Therefore, it was decided to perform the parallelization using a CPU.

Now that the potential locations for parallelization have been located and analyzed, we can estimate the effort required for the actual parallelization and prioritize them in order to know which one to parallelize first. In this experiment we prioritized the parallel potential locations based on their estimated speedup estimation achievable through parallelization. The priority of the functions can be seen in Table 7 and they range from high to low where a lower number means a lower priority.

<table>
<thead>
<tr>
<th>Function</th>
<th>Estimated speedup</th>
<th>Priority</th>
<th>Estimated development effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Align object</td>
<td>71.79</td>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td>Extract environment</td>
<td>62.79</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>Edge detection</td>
<td>3.27</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Stitch point-cloud</td>
<td>2.44</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>Model environment</td>
<td>1.4</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Locate placing area</td>
<td>1.29</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>160</td>
</tr>
</tbody>
</table>

Table 7: Priority and development effort estimation of parallel potential locations

After the specification phase and this analysis phase, the remaining budget in terms of man hours was 152. Based on this effort estimation, the effort (160) would exceed the remaining available budget (152). This means that the parallelization of the Modeling of the environment does not fit within the budget. However, it will still be kept as a planned parallelization in case it turns out that there is budget left when the implementation of the other parallel portions is completed.

7.1.3 Development phase

Once the analysis phase is completed, it is time for the parallelization development phase, which is done iteratively. More specifically, each step of this phase is performed on each included parallel potential location that fits within the financial budget starting from the highest priority.

Iteration 1

Based on the analysis in Section 7.1.2 the decision was to take a data-parallel approach using OpenMP and CUDA to parallelize the align object function. After further analysis of the code it was discovered that refactoring and implementation of several mathematical functions also had to be done for the GPU implementation. This was due to the use of third party libraries that were not callable from the CUDA code. This needed to be implemented in order to run the code using CUDA. Additionally, refactoring was needed to change some computation dependencies so that they could be performed in parallel. This resulted in the implementation taking more time than expected. The results of the parallelization are shown in Figure 4.

This gives us the speedup by a factor of 3.38 for the OpenMP implementation and 14.31 on the CUDA implementation. The speedup of the GPU parallelization is mainly limited by the need to transfer data between the CPU and GPU memory spaces. The validation was done through the use of a unit test, comparing the expected result to the result of the parallelized implementations. Time spent on this iteration was as follows; 16 hours on analysis and design of the parallelization implementation, 40 hours were spent on the implementation and validation.
Sequential
OpenMP
CUDA

Figure 4: Execution times of Align Object after parallelization

Iteration 2
The next parallel potential location on the priority list is the functionality of extracting the environment from a point-cloud. From further analysis of this code a potential risk for race conditions was found, due to reading and writing to a list containing the output of the function. This was handled in different ways for the CPU and GPU implementations. For the CPU implementation a critical section was added to protect from multiple access to the list. As for the GPU implementation, the decision was made to add an array of bools representing whether a point should be kept or not. This allowed each thread on the GPU to read and write its result independently without interfering with other threads or corrupting any data. However, this also introduced a small overhead after the function since it required a sequential iteration through this array in order to add elements to the output.

After further analyzing the code, it was noted that this function also contained third party code that was not usable from CUDA code running on the GPU. Once this additional functionality was implemented, the implementation of the parallel version of extracting the environment was carried out for the CPU and the GPU. The result of this parallelization using OpenMP showed a speedup of 3.06 times compared to the sequential solution. As for the CUDA implementation, a 13.07 times faster execution time was recorded in comparison to the sequential one and 4.19 times faster than the OpenMP implementation as shown in Figure 5.

Figure 5: Execution times of Extract Environment after parallelization

In Figure 5 the difference in execution time is shown in milliseconds where the sequential solution takes 295.4 milliseconds while the parallel solution takes 96.47 milliseconds. Additionally, the execution time of the CUDA accelerated implementation took 22.6 milliseconds. As previously mentioned this yielded a 13.07 times speedup which may not seem much considering the computational capability of the GPU in the benchmarking system. However, this execution time includes the overhead from transferring the data to and from the GPU as well as the added overhead of sequentially iterating through the output. In order to validate that the parallelization did not introduce errors, a unit test was implemented to compare the expected output and the actual output from the parallel implementation.
The time spent on each activity in the first iteration were as follows; 8 hours on analysis and design of the parallelization implementation and 8 hours for implementation and validation of the OpenMP implementation. As for the CUDA implementation it took longer time due to a more complex implementation. This resulted in 36 hours of work for the implementation and validation of the GPU parallelization.

**Iteration 3**

Detecting edges in a point-cloud was defined as a data-parallel problem in the analysis phase. Additionally, it was decided to parallelize it on the CPU using OpenMP. There was no additional refactoring necessary found through further analysis of the code. Although, it was found that there was a risk for race conditions which has to be protected using a critical section. The implementation of the parallel version was then carried out and the results are shown in Figure 6.

![Figure 6: Execution times of Edge Detection after parallelization](image)

The measurements show that the parallelized version of the code is 3.01 times faster than the sequential version. Validation was then carried out by implementing and running unit tests on the code. The time spent on the different parts of the development phase for this code was as follows: 2 hours on analysis and design followed by 4 hours of implementation and validation.

**Iteration 4**

During the analysis phase, it was decided that a data-parallel approach should be used for the parallelization of the point-cloud stitching functionality. The software was further analyzed to see if there was any required refactoring to allow for the parallelization. It was found that there was no major refactoring needed and the implementation could start. The results of the parallelization using OpenMP is shown in Figure 7.

![Figure 7: Execution times of Stitch point-cloud after parallelization](image)

After the parallelization the execution times was 2.31 faster than the sequential version which was considered as a successful speedup through parallelization. Again, unit tests were used in order to validate that the parallelization did not alter the output of the function. The time spent on this was 8 hours and was divided as follows; 3 hours of analysis and design of the parallel implementation followed by 5 hours of implementation and validation.
**Iteration 5**

Modeling the environment was defined as a task-parallel problem from the analysis, and therefore it was implemented using C++ Threading Library. The analysis showed that adding a point-cloud to the existing environment by stitching them together could be done in parallel to moving the robot arm. However, if the robot arm were able to move to the next camera position before the stitching was complete, there would be a risk of multiple threads trying to access the data representing the environment at the same time. In this case the data must be protected from race conditions using a mutex. The results from the implementation showed a speedup of 1.39 times faster than the sequential solution as shown in Figure 8.

![Figure 8: Execution times of Model Environment after parallelization](image)

These results show that the speedup factor is not as high as other parallelized code portions. In terms of time however, the execution is 2774 milliseconds faster which is more than previous parallelizations. The limiting factor of the speedup is that the majority of time is spent by moving the robot arm, therefore reducing the effect that the parallelization has on the speedup factor. Development effort spent on this iterations resulted as follows; 4 hours for design and 6 hours for implementation and validation.

**Iteration 6**

The analysis of the task to locate the placing area showed that it is task-parallel and that the C++ Threading Library should be used to express concurrency. After an additional analysis of the code, we realized that a synchronization must occur before calculating the path from the picking area to the placing area. This was because the path cannot be calculated unless the placing area is already known. Therefore, the thread that performs the calculations of the placing area should be joined with the main thread before this point. The implementation was then carried out and the results of the speedup can be seen in Figure 9.

![Figure 9: Execution times of Locate Placing Area after parallelization](image)

As Figure 9 shows, the speedup factor gained from parallelization was only 1.27. However, seeing as the idle time is reduced by 473 milliseconds and the implementation was fairly simple this can still be considered as a fair result. We validated the implementation and checked that the execution did not go past the synchronization without joining with the main thread and that the parallelization did not cause errors. The time spent on the activities was as follows; 4 hours on analysis and design, followed by 4 hours on implementation and validation.
The parallelization was completed 8 hours ahead of schedule and a summary of the results is depicted in Table 8.

<table>
<thead>
<tr>
<th>Function</th>
<th>Estimated Effort</th>
<th>Actual Effort</th>
<th>Speedup</th>
<th>Parallelization approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Align object</td>
<td>60</td>
<td>56</td>
<td>14.31 (3.38 with CPU)</td>
<td>GPU + CPU</td>
</tr>
<tr>
<td>Extract environment</td>
<td>40</td>
<td>48</td>
<td>13.07 (3.06 with CPU)</td>
<td>GPU + CPU</td>
</tr>
<tr>
<td>Edge detection</td>
<td>16</td>
<td>8</td>
<td>3.01</td>
<td>CPU</td>
</tr>
<tr>
<td>Stitch point-cloud</td>
<td>16</td>
<td>10</td>
<td>2.31</td>
<td>CPU</td>
</tr>
<tr>
<td>Model environment</td>
<td>16</td>
<td>12</td>
<td>1.39</td>
<td>CPU</td>
</tr>
<tr>
<td>Locate placing area</td>
<td>12</td>
<td>10</td>
<td>1.27</td>
<td>CPU</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>26.66</strong></td>
<td><strong>24</strong></td>
<td><strong>5.89</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Experiment summary

The numbers show that for the parallelizations using a CPU, the effort estimation was higher than what was required in reality. At the same time, one of the GPU parallelizations took less time than expected (Align object) while the other took longer than expected (Extract environment). Additionally, the results show that the speedup achieved from GPU parallelization was higher than that of the parallelizations on CPUs. However, this additional speedup came with a cost since they required additional effort to implement. The recorded effort spent on each phase when applying the parallelization method is shown in Table 9.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Effort spent (Man hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification Phase</td>
<td>8</td>
</tr>
<tr>
<td>Analysis Phase</td>
<td>40</td>
</tr>
<tr>
<td>Implementation Phase</td>
<td>136</td>
</tr>
</tbody>
</table>

Table 9: Time spent on activities

This shows that 26% of the effort was spent on preparation and location of parallel potential while 74% was spent on designing, implementing and validating the parallelization.
8 Discussion

As first step of this study we conducted an informal literature review in order to find existing parallelization approaches and to identify open research challenges. Conducting a systematic literature review could potentially have allowed us to further refine the research challenges to address. However, since our main goal was to provide a parallelization method and to evaluate it, and given the timeline for this work, we decided to go for a pretty detailed informal review and then focus the effort on the actual thesis contribution.

As for the definition of the parallelization method, we believe that all the steps of the parallelization method are necessary. Removing any of them could potentially affect the result of the parallelization negatively, either in terms of achieved speedup or the amount of effort required for the parallelization.

One of the steps of the research method used in this study, as described in Section 5, was to create alternative solutions and then choose the best one. In the end, the choice was between two fairly similar solutions. In this case, the choice was whether to have the design step of the parallelization method in the analysis phase or in the development phase. As Section 6.2 describes, the choice fell upon the latter. Defining the design of the parallel implementation in the analysis phase would potentially allow for better estimations regarding implementation effort. At the same time, this could potentially introduce additional overhead to the parallelization method. Particularly in cases where effort is spent on designing parallel implementations that are not actually implemented, because they do not fit within the set budget. The decision was therefore to have the design in the development phase instead so that time spent on designing parallel implementations does not go to waste.

We give examples of specific tools for parallelization on CPUs and GPUs using C++. Although, the parallelization itself is generic and can be applied on codebases with different languages and using different tools. However, this may introduce additional overhead during the specification phase of the parallelization method. The overhead would consist of searching for similar tools and parallel computation APIs to the ones that are suggested in this thesis. Once suitable tools required for the parallelization are identified, the parallelization method can be applied.

The evaluation of the parallelization method was done by conducting an experiment in industrial settings where the method was applied to an existing sequential software codebase. In the experiment, the results are affected by the experience and knowledge of the researcher running it. Instead, observing multiple developers applying the parallelization method and on different software codebases through case study would lower biases. This would potentially show a more accurate representation of how the parallelization method performs. As for the systematic literature review, a case study was out of the scope for this work for the same reasons.

In the experiment the effort required for the parallelization using GPUs was 4.75 times higher than the parallelization using CPU. At the same time, the achieved speedup was 4.35 times higher for the parallelization using GPUs compared to the CPU parallelization. While affected by knowledge and experience of the researcher running the experiment, these results suggest that the effort to speedup ratio are roughly the same with CPU parallelization being slightly more efficient. Additionally, in cases were extra speedup is necessary and GPU parallelization is feasible, it is suggested to exploit GPUs. However, more parallelizations performed independently by different persons with different experience and background, and using different computation APIs, would need to be measured in order to draw more conclusive results. A controlled experiment or case-study involving several researchers and several software codebases is left as a suggestion for future work.
9 Conclusion

In this study we have explored parallelization of existing sequential software focusing on locating parallelization potential in the existing codebase and investigating parallelization targeting both multi-core CPUs and GPUs. The contribution of this work is a generic parallelization method for introducing parallelization to sequential software. In the proposed method, dynamic analysis in combination with manual analysis is used to locate parallel potential in existing software. Additionally, a feasibility analysis is used to decide whether it is worth parallelizing a piece of software and to decide which hardware platform to utilize.

In order to evaluate the proposed method, an experiment was carried out in industrial settings. Three research questions were defined for the experiment and the outcomes are as follows:

1. **How much overhead does the parallelization method introduce?**
   26% of the effort was spent preparing, locating and analyzing parallel potential.
   74% of the effort for designing, implementing and validating of the parallelization.

2. **How much did the execution time of the parallelized software improve from the original?**
   The average case achieved a 5.89 times faster execution time compared to the sequential implementation.

3. **Are there any limitations in using the parallelization method?**
   The parallelization method does not consider other hardware platforms than CPUs and GPUs.

   Additionally, the experiment showed that parallelization using CPUs required less effort compared to parallelization using GPUs. However, the speedup gained from GPU parallelization was higher compared to what was achieved on a CPU. This resulted in the effort to speedup ratio being similar for both approaches. In some situations the added speedup from GPU parallelization may not be necessary because of the added effort required. However, when the speedup from CPU parallelization is not sufficient, GPU parallelization may be worth the additional effort.

   For future work to further explore this topic we suggest a study comparing existing parallel computation APIs in terms of achieved speedup and required development effort. The purpose of this would be to define suggestions regarding which one of them to choose depending on problem and goals. Another suggestion for future work is to conduct studies where multiple developers with different knowledge and experience apply the parallelization method in practice and evaluate the results. The results of such a study can be used to refine the phases of the parallelization method to reach the broadest audience possible. A final suggestion for future work is to conduct a study where the parallelization method is applied on multiple software of different size and complexity. The purpose of this would be to refine suggestions for in which cases the parallelization method is best suited.
References


