A Generic Approach to Efficiently Parallelize Legacy Sequential Software

59

Andreas Granholm and Federico Ciccozzi

Abstract

Multi-core processing units have been the answer to ever increasing demand of computational power of modern software. One of the main issues with the adoption of new hardware is portability of legacy software. In this specific case, in order for legacy sequential software to maximize the exploitation of the computational benefits brought by multi-core processors, it has to undergo a parallelization effort. Although there is a common agreement and well-specified support for parallelizing sequential algorithms, there is still a lack in supporting software engineers in identifying and assessing parallelization potentials in a legacy sequential application. In this work we provide a generic parallelization approach which supports the engineering in maximizing performance gain through parallelization while minimizing the cost of the parallelization effort. We evaluate the approach on an industrial use-case at ABB Robotics.

Keywords

Parallelization · Legacy · CUDA · OpenMP

59.1 Introduction

The greediness of modern software in terms of computational power has led to a wide industrial adoption of the so called multi-core processing units [1]. Besides multi-core Central Processing Units (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, with the drawback of a generally slower clock frequency.

In order to utilize the full potential of parallel hardware, existing sequential software needs to be re-engineered to exploit parallelization. Parallelizing sequential software mainly conceives two steps: (1) individuation of software portions with the best parallelization potentials in relation to a more or less fixed "parallelization budget".[1] While techniques for parallelizing sequential algorithms have been largely studied, there is lack of support for guiding the software engineering in identifying parallelization potentials [2].

Parallelizing software increases performance, but it can negatively affect other quality attributes, such as verifiability and maintainability [3]. Moreover, due to the intrinsic complexity of parallelizing software [4], introducing parallelization in existing sequential software can be a very costly task. In this study we are interested in finding out which software and process aspects shall be considered when parallelizing legacy sequential software and how to take them into account. Thus, we focus on providing a generic parallelization approach which aims at maximizing the increment of performance of existing software through parallelization while minimizing the cost of the parallelization effort. We consider both CPUs and GPUs as parallelization hardware targets to maximize the potential performance gain depending on the specific application type and we evaluate the parallelization approach on an industrial use-case.

The remainder of the paper is structured as follows. In Sect. 59.2, we provide a snapshot of the state-of-the-art in contraposition to our contribution. The parallelization approach and its details are unwound in Sect. 59.3, and its industrial evaluation is described in Sect. 59.4. The paper is

A. Granholm
ABB Robotics, Västerås, Sweden
e-mail: andreas.granholm@se.abb.com

F. Ciccozzi (✉)
Mälardalen University – IDT, Västerås, Sweden
e-mail: federico.ciccozzi@mdh.se

[1] The parallelization budget depends on several factors and it represents the budget in terms of maximum parallelization effort by which the parallelization itself is considered preferable to a re-implementation from scratch.

S. Latifi (ed.), *Information Technology – New Generations*, Advances in Intelligent Systems and Computing 738,
https://doi.org/10.1007/978-3-319-77028-4_59

concluded with a conclusive summary and identified future research directions in Sect. 59.5.

59.2 State-of-the-Art and Novelty

In 2003, Intel [5] presented a method to introduce threading in sequential software using a generic development lifecycle consisting of six different phases. This methods only targets developers using Intel threading tools and it only considers CPUs as target hardware. Tovinkere [6] presents an approach very similar to Intel's, also using threading to increase the performance of sequential software. Tovinkere's approach is less tool-dependent than Intel's, but focuses on CPUs only too. Jun-feng[7] provides an approach for parallelization, but without supporting the location of parallellization potentials and only targeting CPUs. Christmann et al. [8] present a method for porting existing sequential software to a parallel platform taking into account some financial aspects of the parallelization, but target CPUs only. The method proposed by Nvidia [9] is based upon an iterative approach able to deploy a parallelized portion as soon as possible to quickly create value for the users. However, it only considers GPUs. The method presented by Tinetti et al. [10] is iterative and focuses on parallelization of Fortran software using OpenMP on CPUs.

The novel aspect brought by our contribution is the provision and the interplay of the following four features:

1. consideration of effort estimation for the parallelization task;
2. support for identifying software portions with best parallelization potential;
3. consider both CPUs and GPUs as potential target parallel platforms;
4. not focus on specific languages or tools.

The combination of (1) and (2) gives the opportunity to prioritize software portions with parallelization potential that require lower effort and provide higher potential performance gain. With (3), we can maximize performance gains in case where both CPUs and GPUs are available. Through (4), we aim at providing a generic approach which can be instantiated using different technologies depending on the application scenario.

59.3 The Parallelization Approach

The parallelization approach that we propose consists of three phases and is depicted in Fig. 59.1. The first phase represents the definition of the goals of the parallelization as well as the tools to use. In the second phase, developers analyze the software for individuating parallelization potential and perform a parallelization feasibility analysis. The third and last phase concerns the actual parallelization, where the selected software portions are further analyzed and a parallel solution is designed, implemented, and validated. In the following subsections we describe the three phases in more detail.

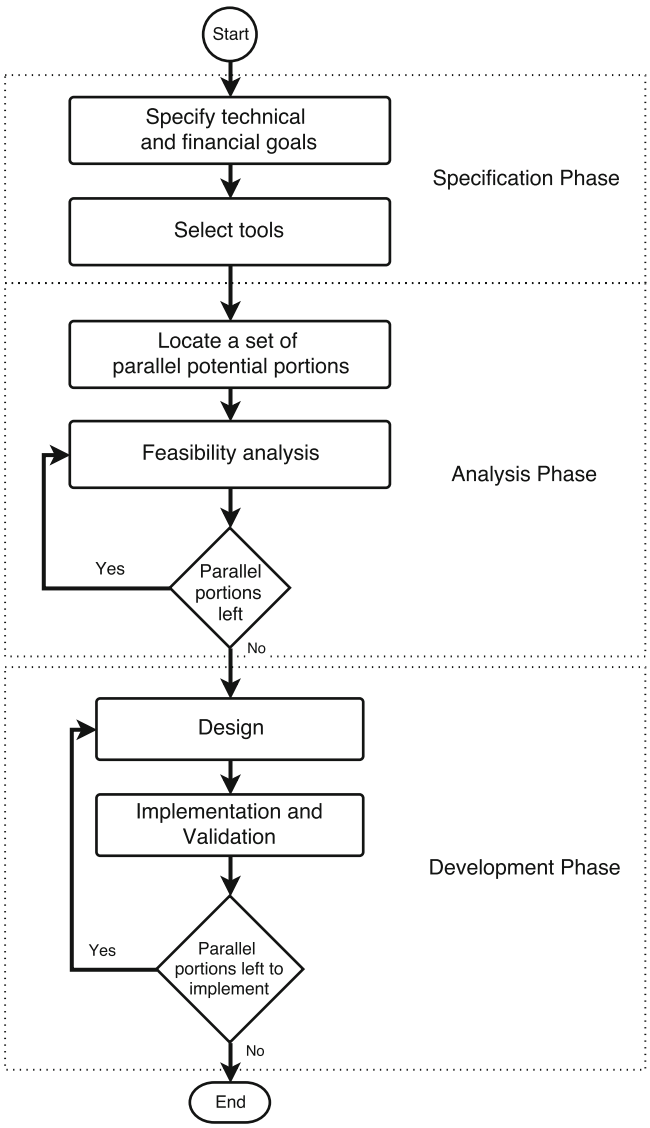

Fig. 59.1 Parallelization approach

59.3.1 Specification Phase

In this phase, the engineer specifies the goals of the parallelization and decides on which tools to use. Goals are of two kinds: financial and technical. An example of financial goal is development effort introduced by the parallelization. An example of technical goal is the desired performance gain in

terms of execution speedup as a result of the parallelization. In the industrial evaluation of our approach we focus on these two. Clearly, there are two ways to tradeoff them: either prioritizing cost over performance or the other way around [8]. In the first case, there is a fixed budget for carrying out the parallelization task and, once the budget is used, parallelization stops no matter how much performance has improved. In the second, we decide on a target level of performance to reach, and parallelization is carried on until that is reached, with looser constraints on cost.

Apart from goals, the engineer is supposed to select the tools to use for the parallelization, and this is usually very dependent on the characteristics of the software to parallelize. For instance, if the existing software is written in C++, then tools that support C++ need to be used. In some cases, it could be beneficial to first port the existing software in order for it to be used on a unsupported platform; this would require additional parallelization phases, and it is not in the scope of this work. A profiling tool, to be used during the identification of software portions with parallelization potential should be selected too, together with a monitoring tool to measure the software execution time to be used for feasibility analysis. Additionally, the engineer selects which parallel computing APIs to use, considering compatibility with the software to parallelize and the targeted parallel platform.

59.3.2 Analysis Phase

The purpose of this phase is to analyze the software to (1) find potential for parallelization, (2) estimation of maximum performance gain from parallelization, and (3) estimation of parallelization effort, in the given order. We address (2) and (3) as feasibility analysis.

59.3.2.1 Identification of Parallelization Potentials

Our approach indicates two complementary methods, to be exploited jointly, for carrying out this task: manual and automatic control flow analysis. Analyzing the control flow of the software to parallelize is needed to reveal parallelization potentials in the program flow. Manual analysis of an entire piece of software can be time-consuming, so automatic analysis can help in pointing out where in the software to look for parallelization potentials. The profiling tool selected in the specification phase can be used to search for pieces of the software where much of the execution time is spent, i.e. hot-spots. If these portions of the software are parallelized, the total gain of performance is greater than parallelizing "colder" portions. Once a set of hot-spots is identified, a manual inspection of their code needs to be performed to decide whether there is potential for parallelization. This is done by looking for independent computations that can thus be performed in parallel.

59.3.2.2 Feasibility Analysis

Once portions with parallelization potential are found, we advise to run a feasibility analysis to find out whether it is worth parallelizing them. The first step is to measure how much code can be parallelized. Once this is defined, an estimation of the potential execution time speedup from parallelization can be calculated by using well-defined mathematical equations. In our approach we suggest the use of Amdahl's law[2] [12] to assess the potential speedup that can be achieved from parallel processing.

$$Speedup(N)_A = \frac{1}{(1-P) + P/N + O_N} \tag{59.1}$$

The formalization of Amdahl's law is depicted in Eq. (59.1), where P is the parallel portion of the process, N is the number of processors, and O_N the overhead introduced from using N, if any. The estimated speedup factor calculated by applying Amdahl's can then be used to calculate the actual speedup gain in time through Eq. (59.2), where $Speedup_{time}$ represents the speedup gain in time and $SeqET$ the total sequential execution time.

$$Speedup_{time} = SeqET - \frac{SeqET}{Speedup(N)_A} \tag{59.2}$$

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. Since these laws expect a number of processors to apply the parallelization on, a decision of whether to utilize CPUs or GPUs (note that we suppose that both are available) for the parallelization must be taken beforehand. This decision can be taken looking at the type of parallelization we want to achieve: task parallelism, where the goal is to distribute task across processors running in parallel, or data parallelism, where data is supposed to be distributed. For task parallelism, CPUs are considered more suited due to higher clock frequencies and lower memory management overhead. For data parallelism, we suggest to make an estimation of the potential speedup in both the cases of targeting CPUs and GPUs. This is due to the fact that, besides varying processor's characteristics, such as clock frequency and memory clock frequency, 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 or

[2] An alternative to Amdahl's law is represented by Gustafson's law [11], which can be more suitable when parallelizing algorithms that can expand the amount of computation to fit the amount of parallelization available.

thousands of cores, the parallel task should be fine-grained enough to maximize the utilization of these cores; for coarse-grained tasks, it might be better to choose a CPU instead.

The second step of the feasibility analysis phase is to estimate the effort needed for the actual parallelization task in terms of time. There are three aspects that compose the total effort: (1) time to design a parallel implementation, (2) time to implement the parallelization, (3) time to validate the parallelized application.

Once estimated speedup is calculated for each candidate software portion, and depending on the set financial goals (i.e., maximize performance gain or minimize effort), the engineer decides whether to parallelize any of the candidates. The outcome of this step is a prioritized list of software portions to be parallelized within the given effort budget.

59.3.3 Development Phase

During this iterative phase, selected software portions are actually parallelized, in the prioritized order defined in the previous phase. 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 CPU, we are facing either task or data parallelism. In the case of a task parallelism, we recommend to use a threading library, such as the C++ Thread Support Library.[3] For data parallelism on CPU, the recommendation is to use a compilation-based approach, such as OpenMP.[4] If parallelization targets a GPU, the engineer will use the parallel computing API selected in the specification phase, such as NVIDIA's CUDA.[5]

It is worth mentioning that further analysis of the software portion to parallelize is amenable since activities, such as refactoring of data, may be necessary to maximize parallelizability. Actual implementation and validation of the parallelization are decided by the engineer; our approach does not pose any limitation on them since they do not affect the expected gains, if properly carried out.

59.4 Industrial Evaluation

In order to evaluate our parallelization method, we set up an experiment exploiting a real-life industrial use case focusing on 3D sensors using structured light at ABB Robotics (site of Västerås, Sweden). The goal of the experiment was to answer the following set of research questions:

RC1 *What is the effort of the parallelization approach overall and the effort of its phases individually?*
RC2 *How much did the execution time of the parallelized software improve from the original?*
RC3 *Are there clearly perceived limitations in applying the parallelization approach?*

In order to answer RC1, the effort spent on each step of the approach was recorded so to track how much time was spent on preparation, planning, and implementation activities. Benchmarking was performed on the software portions selected for parallelization, before and after the application of the approach in order to answer RC2. Additionally, we kept track of the "sensations" of the engineers applying the approach so to have an informal understanding on clear limitations of the approach (RC3).

The parallelization approach was applied on a palletizer application, developed as a proof-of-concept for a 3D vision library at ABB Robotics. The application makes a robot arm to pick and place objects from point A to point B. The exact location of point B is unknown; only a larger area of where B's location is known beforehand. 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 Fig. 59.2. Currently, the software running on the Palletizer is sequential, and the vision-related algorithms are so computational heavy that, once an image is taken, the application needs to wait for the vision algorithms to compute it before it can move the robot arm. This idle time increases the overall cycle time and limits throughput. The goal of the experiment was to apply the parallelization method to the sequential Palletizer application to improve its performance.

59.4.1 Specification Phase

The goal of the parallelization was to maximize performance gain in terms of execution time speedup; since the experiment's goal was to assess the parallelization method, we did not set any limit on parallelization effort, but rather measured it. Moreover, we aimed at providing parallelization for both targets (in case of data parallelism, where GPUs are favorable), CPU and GPU, in order (1) to compare results from the two, both in terms of effort and achieved speedup, and (2) to maximize portability (in case no GPUs are available). The development as well as the benchmarking was carried out using a Lenovo P50 laptop with Windows 7 as operating system. The laptop contains a CPU Intel Core i7-6820HQ vPro (8M Cache, up to 3.60 GHz) with a memory of 8 GB DDR4 2133 MHz, and a GPU Quadro M1000M,

[3] http://en.cppreference.com/w/cpp/thread.
[4] http://www.openmp.org/.
[5] https://developer.nvidia.com/cuda-zone.

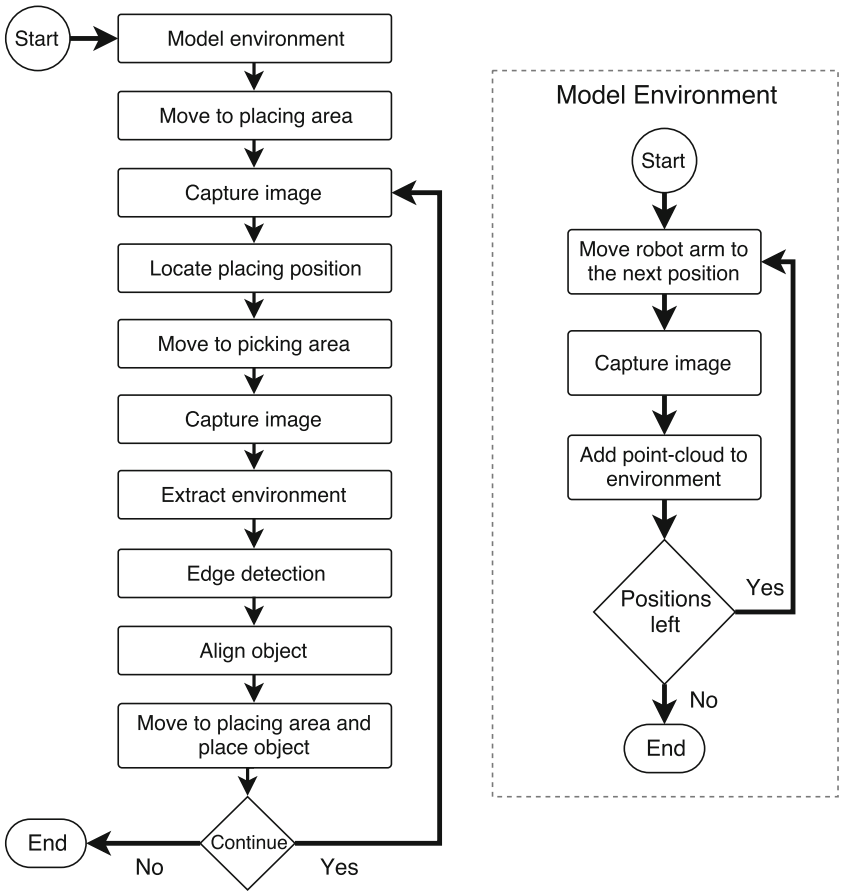

Fig. 59.2 Control flow of the Palletizer application

512 CUDA cores 993-1072 MHz with a memory of 4 GB GDDR5 5000 MHz and 128 Bit bus width.

The existing code base was developed in C++11 using Visual Studio 2015 (v140 platform toolset) and Microsofts VC++ compiler, and the code was compiled using the '-O2' optimization flag. The C++11 threading library and OpenMP were used in order to express concurrency on the CPU when applying the parallelization approach for task and data parallelism, respectively. To express concurrency on GPUs, we used NVIDIA's CUDA since the target system includes a CUDA-enabled GPU. Effort spent in this phase was 8 h.

59.4.2 Analysis Phase

The first step of the analysis phase is to locate a set of software portions that could potentially be parallelized. Based on the control flow of the Palletizer application and the dependencies between actions, there were two visible locations for parallelization. The first one was the `Model environment` procedure, where there were no dependencies between the robot moving and adding the point-cloud from the image to the point-cloud representing the environment. The two actions could be run in parallel. The second one was when a picture of the placing area is taken and the placing position is calculated, after which a collision-free path to the picking area is planned. The two actions are again independent and could be run in parallel.

To systematically identify hot-spots, automatic (dynamic) analysis using Visual Studio Profiling Tools was run on the codebase. The result showed 13 functions that were accessed frequently and consumed much computational power. These functions were inspected and analyzed manually for parallelization potential. After the analysis, the following 4 functions were identified as parallelizable: `Model environment`, `Extract environment`, `Align object`, `Edge detection`. Once these locations of parallel potential software were located, we conducted a feasibility analysis on them. `Model environment` takes care of combining multiple images taken from the robot arms perspective. The robot arm moves to a set of positions and takes an image at each position. What we can do 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 since only two tasks can run in parallel. `Model environment`'s sequential execution time was 9822 ms. Then, the individual times of the two tasks that could run in

parallel was 2816 ms for adding the image and 6635 ms for moving the robot arm. In theory, 5632 ms (2*2816, where 2816 ms is the shorter execution time), or 57.34%, of the total sequential execution time could run on two cores at the same time. Since we address task parallelism on two cores, we apply Amdahl's law only for a 2-core CPU. The application of Amdahl's law is shown in Eq. (59.3) gives an estimated speedup factor of 1.4, for a speedup time of 2806.29 ms (see Eq. (59.4)).

$$Speedup(2)_A = \frac{1}{(1 - 0.5734) + 0.5734/2} = 1.4 \quad (59.3)$$

$$Speedup_{time} = 9822 - \frac{9822}{1.4} = 2806.29 \quad (59.4)$$

The sequential execution time of `Extract environment` was 295.4 ms, with a parallelizable portion of 291.26 ms (98.6%). Since this is data parallelism, we estimate speedup on both CPU and GPU, using the maximum amount of available cores (4 for CPU and 512 for GPU). At this point, the speedup estimation was performed for both CPUs and GPUs according to the parallelization method. Absolute speedup factors for CPU and GPU are 3.83 and 62.79, respectively, as shown in Eqs. (59.5) and (59.6). Since the GPU used in this experiment has a core frequency which is 27.7% of the CPUs, the relative speedup for GPU is $62.79 * 0.277 = 16.5$.

$$Speedup(4)_A = \frac{1}{(1 - 0.986) + 0.986/4} = 3.83 \quad (59.5)$$

$$Speedup(512)_A = \frac{1}{(1 - 0.986) + 0.986/512} = 62.79 \quad (59.6)$$

The reasoning is similar for the remaining functions, whose estimated speedup gains are as follows (for the sake of page limitation we do not provide all the details of the related parallelization rationale).

`Aligning object` has an execution time of 1548.54 ms, with 98.8% of it parallelizable through data parallelism. Absolute speedup factors for CPU and GPU are 3.86 and 71.79 respectively, as shown in Eqs. (59.7) and (59.8), with a relative GPU speedup factor of 19.88.

$$Speedup(4)_A = \frac{1}{(1 - 0.988) + 0.988/4} = 3.86 \quad (59.7)$$

$$Speedup(512)_A = \frac{1}{(1 - 0.988) + 0.988/512} = 71.79 \quad (59.8)$$

`Edge detection` has an execution time was 32.73 ms, of which 92.6% parallelizable through data parallelism. Absolute speedup factors for CPU and GPU are 3.27 and 13.19 respectively, as shown in Eqs. (59.9) and (59.10), with a relative GPU speedup factor of 3.65.

$$Speedup(4)_A = \frac{1}{(1 - 0.926) + 0.926/4} = 3.27 \quad (59.9)$$

$$Speedup(512)_A = \frac{1}{(1 - 0.926) + 0.926/512} = 13.19 \quad (59.10)$$

As we can see speedup factor for CPU (3.27) is slightly lower to the one for GPU (3.65). This would suggest to target GPU. Nevertheless, since the size of the point-clouds provided as input to `Edge detection` are normally up to 200 points, which would take up to 200 GPU cores, we will have a minimum of 312 cores always 'idling', thus severely under-utilizing the GPU. CPU is thereby a better candidate in this specific case.

In this experiment we prioritized the software portions to parallelize based on their estimated speedup factor achievable through parallelization. A summary of the estimated speedup factor, priority, type of target platform, and effort spent for the analysis of the specific software portion is shown in Table 59.1.

59.4.3 Development Phase

In this phase the selected software portions are parallelized from highest to lowest priority until the depletion of the available effort budget (no budget limits in our case since we wanted to measure the needed effort). Moreover, each parallelized code portion was validated through canonical unit testing. In the following we describe the results of this hand-crafted phase in terms of achieved speedup contra estimations done in the analysis phase.

The `Align object` procedure was parallelized both for CPU, using OpenMP, and for GPU, using CUDA. After further analysis of the related sequential code, we discovered that re-implementation of several mathematical functions had to be done for the CUDA implementation, since used third-party APIs did not provide native support for CUDA. Additionally, refactoring was needed to loosen up a number

Table 59.1 Estimated speedup, priority, platform, and analysis effort

Function	Estimated speedup	Priority	Target	Analysis effort
Align object	3.86	5	CPU	16
Align object	19.88	5	GPU	-
Extract env.	3.83	4	CPU	8
Extract env.	16.5	4	GPU	-
Edge det.	3.27	3	CPU	2
Model env.	1.4	2	CPU	4

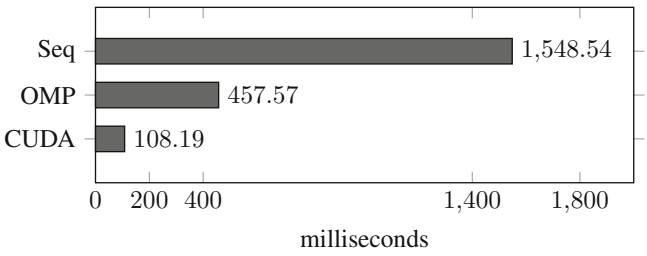

Fig. 59.3 Execution times of `Align Object`

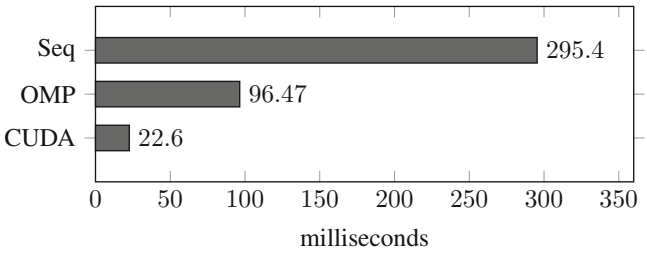

Fig. 59.4 Execution times of `Extract environment`

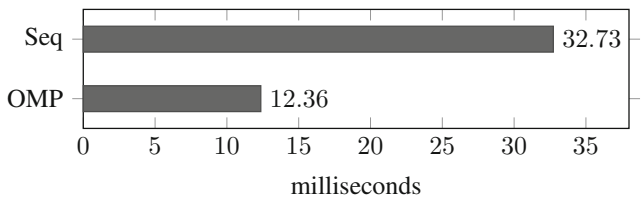

Fig. 59.5 Execution times of `Edge Detection`

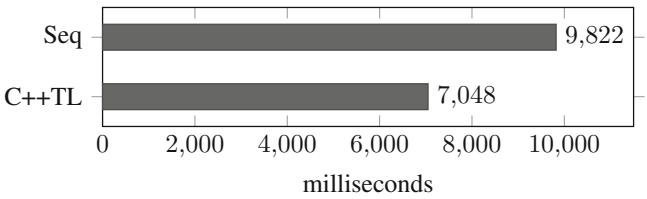

Fig. 59.6 Execution times of `Model environment`

Table 59.2 Experiment summary

Procedure	Estimated speedup	Actual speedup	Dev/val effort	Target
Align object	3.86	3.38	4	CPU
Align object	19.88	14.31	36	GPU
Extract env.	3.83	3.06	6	CPU
Extract env.	16.5	13.07	38	GPU
Edge det.	3.27	3.01	4	CPU
Model env.	1.4	1.39	6	CPU

of computation dependencies so that they could be performed in parallel. The results of the parallelization are shown in Fig. 59.3. Note that in Figs. 59.3, 59.4, 59.5, and 59.6, Seq, C++TL, OMP, and CUDA represent the execution times of the sequential application, the parallel application with C++ threading, with OpenMP, and with CUDA, respectively.

The achieved speedup factors were: 3.38 for CPU/OpenMP (expected 3.86), and 14.31 for GPU/CUDA (expected 19.88), as shown in Fig. 59.3. Forty man-hours were spent on implementation and validation.

`Extract environment` was also parallelized for CPU, using OpenMP, and for GPU, using CUDA. From further code analysis, we identified a potential risk for race conditions due to reading and writing to a shared list. This was handled in different ways for the two implementations: for OpenMP, we added a critical section to protect from multiple access, while for CUDA, we added an array of booleans representing whether a point should be kept. The achieved speedup factors were: 3.06 for CPU/OpenMP (expected 3.83), and 13.07 for GPU/CUDA (expected 16.5), as shown in Fig. 59.4. Fort four man-hours were spent on implementation and validation.

`Edge detection` was parallelized for CPU using OpenMP problem in the analysis phase. Additionally, it was decided to parallelize it on the CPU using OpenMP. Also in this procedure we had to add a critical section to avoid race conditions. The achieved speedup factor was 3.01 for CPU/OpenMP (expected 3.27), as shown in Fig. 59.5. Four man-hours were spent on implementation and validation.

`Model environment` was identified as candidate for task parallelization, thus on CPU using C++ threading. To avoid race conditions, we add to protect some of the shared data with a mutex. The achieved speedup factor was 1.39 for CPU/C++TL (expected 1.4), as shown in Fig. 59.6. Six man-hours were spent on implementation and validation.

A summary of the development activity results in relation to the estimations computed during the analysis phase is depicted in Table 59.2. In general we can observe that the actual speedup was indeed fairly close to the estimated one for all cases. As expected, speedups achieved targeting GPUs were higher than the one on CPUs. However, in all cases this came with an additional cost, since parallelization to GPU required additional much higher implementation effort. The overall recorded effort was 132 man-hours, of which 8 spent on the specification phase, 30 on the analysis, and 94 on development.

59.5 Conclusion

In this work we described and assessed a generic parallelization method for introducing parallelization in legacy sequential software. In the proposed method, automatic (dynamic) analysis in combination with manual analysis is used to locate parallelization potentials in existing software. Additionally, a feasibility analysis is used (1) to decide whether it is worth parallelizing a piece of software, and (2) to decide which hardware platform to target (CPU or GPU).

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:

RC1 *What is the effort of the parallelization approach overall and the effort of its phases individually?*

28.8% of the effort was spent preparing, locating and analyzing parallelization potential.

71.2% of the effort for designing, implementing and validating the parallelization.

RC2 *How much did the execution time of the parallelized software improve from the original?*

The average case achieved a 6.37 times faster execution time compared to the sequential implementation.

RC3 *Are there clearly perceived limitations in applying the parallelization approach?*

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, expectedly, higher compared to what was achieved on a CPU. This resulted in the effort to speedup ratio being similar for both approaches. A possible future research work would be 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 what API to choose depending on problem and goals.

Acknowledgements This research is partially supported by the Knowledge Foundation through the MOMENTUM project (http://www.es.mdh.se/projects/458-MOMENTUM).

References

1. E. Chovancova, J. Mihal'ov, Load balancing strategy for multicore systems, in *Proceedings of ICETA* (2015), pp. 1–6
2. M.A. Kiefer, K. Molitorisz, J. Bieler, W.F. Tichy, Parallelizing a real-time audio application – a case study in multithreaded software engineering, in *Proceedings of IPDPS* (2015), pp. 405–414
3. L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 3rd edn. (Addison-Wesley Professional, Reading, 2012)
4. H. Vandierendonck, T. Mens, Techniques and tools for parallelizing software. IEEE Softw. **29**, 22–25 (2012)
5. Intel, *Threading Methodology: Principles and Practices* (Intel Corporation, Mountain View, 2003)
6. V. Tovinkere, A methodology for threading serial applications. Intel White paper (2006)
7. B. Jun-feng, Application development methods based on multicore systems, in *2012 International Conference on Industrial Control and Electronics Engineering* (2012), pp. 858–862
8. C. Christmann, J. Falkner, A. Weisbecker, A methodology for porting sequential software to the multicore platform considering technical and economical aspects of software parallelization, in *Proceedings of ICSOFT-EA* (2014), pp. 551–559
9. Nvidia, CUDA C best practices guide. DG-05603-001_v8.0 (2017)
10. F.G. Tinetti, M. Méndez, A. De Giusti, Restructuring fortran legacy applications for parallel computing in multiprocessors. J. Supercomput. **64**(2), 638–659 (2013)
11. J.L. Gustafson, Reevaluating amdahl's law. Commun. ACM, **31**(5), 532–533 (1988). http://doi.acm.org/10.1145/42411.42415
12. G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in *Readings in Computer Architecture* (Morgan Kaufmann, San Francisco, 2000), pp. 79–81