ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

OPTIMIZACIÓN DE UN ALGORITMO DE TOMA DE DECISIONES BAJO INCERTIDUMBRE PARA PLATAFORMAS HETEROGÉNEAS

OPTIMIZATION OF A DECISION MAKING ALGORITHM UNDER UNCERTAINTY FOR HETEROGENEOUS PLATFORMS

Realizado por
Denisa-Andreea Constantinescu

Tutorizado por
Dr. Mª Angeles González Navarro
Dr. Juan Antonio Fernández Madrigal

Departamento
Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE 2017

Dra. M. Ángeles González Navarro.
Profesor Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

Dr. Juan Antonio Fernández Madrigal
Catedrático de Universidad del Departamento de Ingeniería de Sistemas y Automática de la Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada "Optimization of a Decision making Algorithm under Uncertainty for Heterogeneous Platforms", ha sido realizada por Dña. Denisa-Andreea Constantinescu bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y constituye la memoria del Trabajo Fin de Máster que presenta para optar al título de Máster en Ingeniería Informática.

Málaga, Septiembre de 2017

Dra. M. Ángeles González Navarro.
Codirectora de la tesis.

Dr. Juan Antonio Fernández Madrigal.
Codirector de la tesis.

Acknowledgements

I would like to thank to my advisors, Professors M. Ángeles González Navarro, Juan Antonio Fernández Madrigal and Rafael Asenjo Plaza, for their endless support and encouragement. I especially thank my hard working mum for her unconditional love and care.

Resumen

Este trabajo de fin de máster presenta nuestra experiencia optimizando un algoritmo de toma de decisiones para que un agente robótico navegue en un espacio cerrado en el interior de un edificio. Nuestro escenario de caso de uso se basa en CRUMB, un robot móvil, y en una plataforma de computación heterogénea. Este tipo de plataformas se caracterizan por integrar procesadores con distintas capacidades de cómputo y diferentes requerimientos energéticos. Programar de manera eficiente este tipo de sistemas requiere de un importante trabajo de re-ingeniería de los algoritmos. En nuestro caso, el algoritmo con el que implementamos la tarea de navegación se basa en un proceso de decisión de Markov, y, en particular la *Iteración de Valores* para calcular la política de navegación óptima. Utilizando como punto de partida el algoritmo clásico de Iteración de Valores, proponemos una serie de implementaciones diferentes que optimizan el tiempo de ejecución y/o el consumo de energía para un tipo de sistemas heterogéneos que son objeto de nuestro estudio.

Palabras clave: MDP, Iteración de valor, Programación heterogénea, CRUMB, V-REP, MATLAB

Abstract

This master thesis report presents our experience optimizing a decision-making algorithm for an autonomous robotic agent to navigate in an indoor environment. Our use-case scenario is based on CRUMB, a mobile robot, and a heterogeneous computing platform. This kind of platforms usually incorporate several processors with different computing power as well as energy requirements. Efficiently programming these systems calls for a complex re-engineering work of the original algorithms. In our case, the algorithm used for implementing the navigation task has is based on a Markov Decision Process; we use the *Value Iteration* algorithm. Using as a starting point the classical Value Iteration algorithm, we propose a number of implementations that are optimized for time and/or power efficiency targeting a class of heterogeneous systems.

Key words: MDP, Value Iteration, Heterogeneous computing, CRUMB, V-REP, MATLAB

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

The aim of this thesis is to optimize a decision-making algorithm under uncertainty for an autonomous robotic agent to navigate in indoor environments. For our study we used CRUMB [1], a relatively low cost mobile robot and a heterogeneous computing platform. Porting an application to this type of heterogeneous systems is not exempt of challenges. It requires significant hardware expertise as well as complex re-engineering work of the original algorithms to identify the part of the application and corresponding data that must be offloaded to the accelerator. Moreover, the programmer must handle the appropriate runtime calls to orchestrate the work among the fundamentally different computing devices in order to efficiently exploit the capabilities of the system.

In this work, we have chosen the *Value Iteration* (VI) algorithm for computing the optimal navigation policy in the context of a Markov Decision Process (MDP). Using as a starting point the classical VI algorithm, we have first developed a sequential implementation and then proposed a series of variants that optimize for time and energy efficiency on our target heterogenous platform. Implementing a MDP with a large number of states and actions requires increasing computer memory and computation power according to the problem dimension in order to offer real time performance. In our use case we consider a mobile robot that has to navigate to a target without hitting walls nor the obstacles present in its environment. The application must provide the best action the robot can take, given its current state. We consider that the robot has knowledge of the environment a priori, (it knows where the objects and walls are beforehand). Moreover, the robot is the only agent that modifies the environment. This problem will be solved using the Value Iteration algorithm under a MDP framework to compute the optimal sequence of actions the robot can take at each time step.

For the software implementation we have considered an Odroid XU3 - a Heterogeneous Multi-Processing (HMP) System-on-Chip (SoC)- and an Intel Core i7. As all mobile systems, CRUMB is bound in power, memory and computational capabilities. In this context, our main challenges are (1) optimizing the application by making an efficient use of the available resources (multi-core CPU and GPU) in the previously mentioned heterogeneous architecture, and (2) identifying how big can be the problem that we can be solved using these resources.

1.1 State of the art

MDPs are a powerful formalism to model decision problems in order to to compute optimal policies. Although MDP problems are widely found in several application domains such as security systems, defense, healthcare applications, or ambient monitoring [2], the algorithm studied in this work (Value Iteration) has not been fully exploited in mobile

robotics for practical situations [3]. This is mainly due to its high computational requirements when a lot of states are needed.

In the literature we have encountered mostly GPU-based optimizations of the Value Iteration algorithm that take advantage of SIMD parallelism using CUDA implementations [4], [5], OpenMP implementations [4] and OpenCL implementations [6]. The applications vary in purpose. For examle, Ruiz et al. [4] define a MDP solver in terms of matrix multiplication for crowd simulation. Their algorithm was tested on embedded systems, desktop CPU, and GPUs (CUDA) for real-time crowd simulation and allows embodied agents to navigate on a virtual hexagonal grid environment. Cheng and Lu [6] provide an optimal path solution for multiple agents in a "where to go" problem and use coarse grained state representations in order to reduce the number of computations, while Noer's method [5] offers an approximate path solution over POMDP (Partially Observable MDP). Solution [5] is a GPU point-based Value Iteration algorithm implementation that features belief state pruning to avoid unnecessary calculations. Lacerda *et al.* [7] propose a soft real-time multi-objective sequential decision making technique to generate MDP policies that combine the achievement of a primary task with the execution of as many secondary tasks as possible. This technique assures a high probability of achieving the primary task within a user-defined time-bound. Another very interesting approach for implementig Value Iteration is the one of Zhang et al. [8] that introduces an algorithm (iCORPP) that uses common-sense reasoning to dynamically construct (PO)MDPs for adaptive robot planning.

As far as we know, Ruiz et al. [4] are the only ones that consider real-time restrictions for embedded systems, while Laceda's implementaion [7] assures only soft real-time. Thus far we have not encountered a heterogeneous implementation of the Value Iteration algorithm for navigation of a real robot under real-time and power consumption constraints, which is the main focus on this work.

Mobile autonomous robots have limited resources such as processing power, memory and energy availability, and they are bound to real time response restrictions. Yet, implementing a MDP with a large number of states and actions requires an increasing amount of computer memory and computation power, which jeopardizes the real time performance. There are multiple approaches to overcome these obstacles, such as using improved algorithms for planning, approximating the optimal policy, or using very customized hardware. We propose here the use of heterogeneous architectures in order to deal with low power consumption and real-time requirements, at the cost of adding a new level of complexity to programming. In particular we will explore the implementability of the Value Iteration algorithm with the previously described restrictions.

1.2 Objectives

This Master Thesis has two main objectives. The first one is implementing an efficient application for navigation of a real robot in an indoor environment based on the MDP formalism, which will be able to exploit heterogeneous computing platforms. The second one is to characterize the scenarios that can be solved in a reasonable amount of time on our systems and study the feasibility of adapting the optimal actions of our robot in runtime in cases of environmental changes. We will focus our work on developing and implementing the described objectives for the following general set up: the robot must navigate to a given geometrical target without hitting walls or obstacles present in its environment.

The contribution of this work is two-fold: i) porting the algorithm to modern heterogeneous architectures by using the appropriate programming models, namely OpenMP [9],

TBB [10] (Threading Building Blocks) and OpenCL [11] technologies; and ii) exploring the size of the different dimensions that affect the complexity of the problem to characterize the scenarios that can comply with real-time and power consumption restrictions on the studied heterogeneous architecture.

1.3 Methods and plan of work

The V-model has been used in order to define the project requirements, plan, design, implement, test and validate the results. The proposed work plan has been developed during the third semester of the Master's studies and it corresponds to 12 ECTS credits. The plan of work has included the following tasks:

1. Study of two different heterogeneous architectures (Odroid and Intel Core i7): from the memory and computational resources to the different programming models to exploit them. We have chosen these platforms because they are suitable for a wide range of robotic platforms.
2. Define a MDP model for the robotic problem.
3. Implement a sequential solution in the form of the VI algorithm for the Odroid and Intel Core i7. The data structure for the probability matrix should be sparse tridimensional.
4. Port the sequential solution to the heterogeneous systems using the appropriate programming models, for example, OpenMP [9] and TBB [10] for the multi-cores CPU and OpenCL [11] for the GPU accelerator.
5. Explore the computational burden of different MDP problems (for different number of actions and states):
   - Find out the average "adaption time" when there is a change in the environment and study in which of the previously explored problems it is feasible to recompute the optimal actions for our robot.
   - Assess the obtained solutions on both heterogeneous platforms and evaluate their efficiency and power consumption.
6. Write this thesis.

1.4 Material resources

The following material resources have been available for the student to carry out this Master Thesis:

- CRUMB robot simulator in V-REP.
- Odroid XU3 (Samsung Exynos 5 Octa (5422) featuring a 2.0Ghz Cortex-A15 quad-core and a 1.4Ghz Cortex-A7 quad-core CPUs, ARM's big.LITTLE architecture. The Exynos 5 includes the GPU Mali-T628 MP6).
- Laptop HP Pavilion 15-n230ss featuring an Intel i7-4500U (1.8 GHz, 4 MB cache, 2 cores), an Intel HD Graphics 4400 and 8GB of RAM.
- Tower PC featuring a 4.0GHz Intel i7-6500K quad-core and 16GB of RAM.
- SW Development Kit: OpenMP, TBB, gcc compiler, OpenCL libraries and runtime.

1.5 Thesis structure

This thesis is composed of five chapters. The first one is an introduction chapter containing the objectives of this work, background information summarizing the theoretical aspects which we consider, and a short review of related work.

The second chapter presents an abstraction of the CRUMB robotic system using a Markov Decision Process representation. It includes the state and action definitions, etc., and introduces the Value Iteration algorithm.

The third and fourth chapters are the core of this thesis. In the third chapter we describe the process of obtaining the numerical parameters of the problem for testing the Value Iteration algorithm. We explain why we decided to use a simulation to acquire the needed data to compute a probabilistic model of the robot interaction with the world instead of using the real robot. Also, we present the physical and functional characteristics of the CRUMB robot.

In the fourth chapter we explore different ways to implement the Value Iteration algorithm in order to compute the optimal policy (i.e., figure the best action for every state) as fast as possible while taking into account the power consumption required for it. The algorithms are optimized using two programming models: OpenMP [9] and TBB [10] for mult-icore execution, and OpenCL [11] for GPU accelerated execution. In this chapter we also:

- Explore the computational burden of different MDP problems for different sizes of the input, i.e., different number of actions and states.
- Assess the obtained solutions for the available heterogeneous platform(s) and evaluate their efficiency and power consumption using the library presentend by Corbera *et al.* [12].

The last chapter presents the conclusions of this work, followed by an annex with the configuration of the development environment and bibliography.

CHAPTER 2
Problem definition as a MDP

We have worked with CRUMB, a non-holonomic wheeled mobile robot. Our main use case scenario consists in making the robot to navigate and reach a certain target in a structured indoor environment. From a MDP perspective, the goal is to find an optimal policy to perform such navigation, in the sense of reaching the target (in any orientation) while avoiding obstacles. The target could be the recharging station location or maybe a desk where the robot has to leave mail.

We formally define the MDP problem by describing the robot states [2.1] and actions [2.2], the transition probabilities [2.4] of reaching a state from any other state by taking a particular action, and the rewards [2.3] or cost of doing a certain action in the current state. The robot has some knowledge about the environment, gathered from simulations; this knowledge is in the so-called transition matrix. The desired agent behaviour is modelled by the rewards matrix. In our use-case, the robot is the only agent that modifies the environment. It does not possess a map of the environment, but it is capable of sensing it.

For this problem we need to describe the robot, that is our agent, and the scenario. The robot is defined by its *states* and *actions*, while the way the robot interacts with the scenario is determined by the *rewards* and the *transition probabilities*.

2.1 States

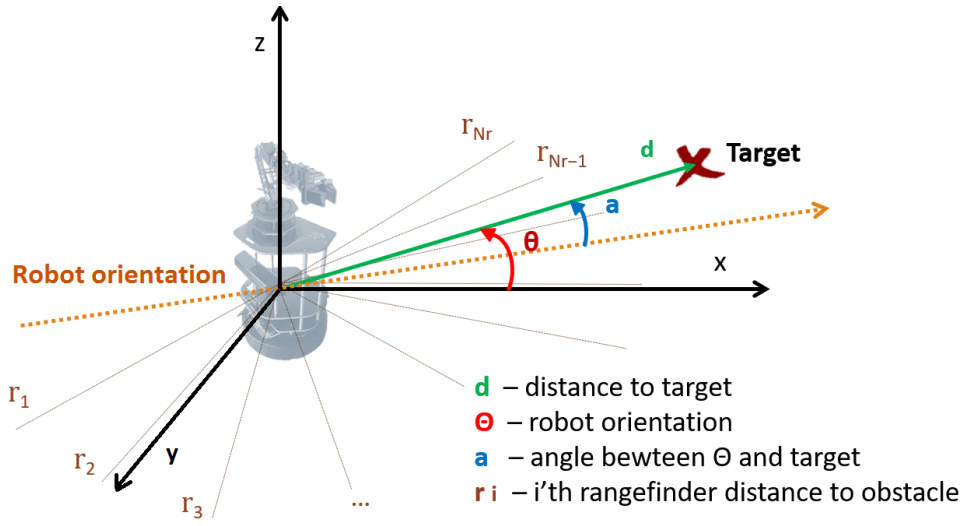

Figure 2.1: Components of the state of the robot.

We consider the robot state completely (deterministically) observable by the values of its sensor readings. Fig. 2.1 illustrates these state components.

The sensors allow the robot to measure its orientation in the universal frame of orientation, given by an angle θ , that will be discretized into NT values. The rangefinder will provide NR measurements of distances to obstacles, equally spaced in the frontal area of the robot (egocentrically), each one discretized into NG values. The robot will know the distance d to its target, that can be discretized into ND values. The robot will also calculate the angle a between its orientation and the relative location of the target, that can be discretized into NA values. The number of states is, thus: $|S| = NT \cdot NR \cdot NG \cdot ND \cdot NA$, where S represents the state space.

2.2 Actions

The robot can take any of the following basic six actions at any iteration:

1. Stay still at the place it is (a_0).
2. Move forward for a given, fixed time (a_1).
3. Move along a curved trajectory for a given, fixed time, either to the left (a_2)
4. or to the right (a_3).
5. Turn around without displacement (a_4).
6. Move backward (a_5).

All action effects last for a given, fixed time. The number of actions is $|A| = 6$, where A represents the action space.

2.3 Rewards

The reward for being in a state s (equation 2.1), $R(s)$, must have a positive value when the robot reaches the target (or possibly when getting close enough to it), a negative value if it collides with obstacles, and a value in between the two otherwise. In our case, z could easily take a neutral value, such as zero. We do not consider rewards depending on the starting state or the action themselves. The first case is identified by the fact that the robot state has a distance to the target close or equal to zero (s.d = 0). The second case (negative reward) happens when the ending state is close to an obstacle (i.e., the robot just hit an obstacle or it is about to do it), and it is identified by the fact that one or more of the rangefinder measurements have a "close obstacle" reading. In other words, any rf(i) reads that there is small or zero distance to an obstacle (in equation 2.1, if $s.rf(i) = 0$).

The R matrix, $R(s, a)$ (equation 2.2), quantifies the expected reward for taking action a while being in state s. It influences greatly in the resulting policy: the higher the values of a state-action pair reward, the greater the probability of the robot taking that action when it is in that particular state. The previous definition of the rewards is crucial for making the robot take the optimal set of actions in order the get to its target as fast and safely as possible. For example, if we would set high positive rewards for the states when the robot

is close to obstacles then the robot would be attracted towards those states - a behaviour that is not desired; very negative rewards would repel the robot from the obstacles.

$$R(s) = \begin{cases} x : x < 0, & \text{if } s.rf(i) = 0 \\ y : y > 0, & \text{if } s.d = 0 \\ z : x < z < y, & \text{otherwise} \end{cases}, \tag{2.1}$$

$$\forall i \epsilon [1, NR], s \epsilon S, a \epsilon A$$

$$R(s,a) = \sum_{s' \epsilon S} P(s' \mid s, a) R(s') \tag{2.2}$$

2.4 Transition probabilities

A most important element of the Value Iteration algorithm is the T matrix (transition probabilities matrix). It contains the probability of reaching any state s' provided that the agent is in a state s and executes action a, i.e., it models the behaviour of the world when the robot acts in it.

Since it holds probabilities, the T matrix must satisfy that:

$$\sum_{s' \epsilon next(s,a)} T(s,a,s') = 1, \forall s, a, s \epsilon S, a \epsilon A \tag{2.3}$$

In equation 2.3, $next(s,a)$ is the set of states that can be reached from state s by executing action a.

We have considered three ways of filling in T:

a) *By intuition:* a human provides values for the probabilities directly, hoping that they capture all the influences between different parts of our structured state correctly. In order to do this, a systematic method for abstracting states, assigning probabilities in the resulting (smaller) transition graph, and then refining that graph, can be very helpful, especially when the structure of the state has many parts (more than 2 or 3), as it is our case.

b) *By simulation:* run the robot in a simulator in such a way that it visits all states often enough to get statistically significant data, and finally process those data to obtain the probabilities of T. The simulator can be implemented in basically two exclusive ways: either it simulates directly the abstract world described by the MDP (this may make more difficult for the human to implement probabilities in the simulator since it is farther from the real world and therefore from the human intuition) or it carries out a (physically) realistic simulation. In the latter case, a procedure to abstract the simulation data into the abstract world is needed, as it also happens in the next alternative.

c) *By gathering data from a real scenario:* a human sets up a real robot to operate in a scenario that can be translated into the abstract world we have depicted in chapter 2, make the robot work until each possible state of the abstract world is visited often enough (a state of the abstract world will correspond to a continuous sub-space of the scenario configuration space), and then proceeds as in the previous paragraph.

In cases b) and c), a procedure to obtain T from the statistical data obtained from the simulation or from the real world experiments must be implemented. Such a procedure will have as inputs a large number of tuples, each one corresponding to a moment where

the state of the system and the action that lead to that state were observed, thus having a number of values assigned to different variables, e.g.: executed action, robot position on the floor, robot orientation, target position, measurements of the sensors, ... First of all (specially for case c), those tuples must be translated into states as they are defined in the MDP. These abstract tuples will be the ones used for calculating T.

The probabilities of T can be obtained then through a simple frequentist approach whereby the probability P of an uncertain event E, written P(E), is defined by the frequency of that event based on the observations. More specifically, the probability of reaching state s' provided that the agent is in state s and executes action a is equal to the number of tuples where the agent was in state s' after executing action a divided by the total number of tuples where the agent was in state s

In this work we have chosen to realistically simulate the robot for data acquisition (option b). Data acquisition via simulation is potentially faster than using the real robot. The main reason for dismissing option a) is because it would be quite complicated for a human being to accurately imagine the values for T matrix when we have a multidimensional state space like ours.

2.5 Policy Values

The policy values needed to select actions are stored in the so-called matrix Q during the execution of the VI algorithm. This matrix is defined in a $|S| \times |A|$ real space and it contains values which indicate the expected reward that the robot is likely to obtain by choosing action a when in state s and follow and optimal policy after that. Bigger values in Q indicate that the robot is closer to achieve its task when taking that action in that state. Negative values indicate that the state-action pair will probably lead the robot to an obstacle.

2.6 The Value Iteration Algorithm

The pseudocode for the VI algorithm [13] needs the following inputs (see Fig. 2.2) :

- S - the set of all states
- A - the set of all actions
- T - the state transition function specifying $P(s' \mid s,a)$
- R - a reward function R(s,a) that associates a reward to every state-action pair
- θ - a threshold, $\theta > 0$, that specifies whether we consider that the algorithm has converged to the optimal policy
- γ, $\gamma \in [0,1]$ - is the discount factor and represents the difference in importance between future rewards and present rewards

and provide teh following outputs:

- $\pi[S]$ - an approximately optimal policy
- $V[S]$ - the value function, i.e., a vector that contains the value of the current policy when starting at each state

- $V_k[S]$ - a local real array that stores the sequence of value functions that the algorithm has found before convergence.

1: assign $V_0[S]$ arbitrarily
2: $k \leftarrow 0$
3: **repeat**
4: $k \leftarrow k+1$
5: **for each** state s **to** [Evaluate policy]
6: **for each** action a **do**
7: $V_k[s] = \sum_{s'} P(s'|s,a)\,(R(s,a)+\gamma V_{k-1}[s'])$
8: **for each** state s **do** [Improve policy]
9: $\pi[s] = \text{argmax}_a \sum_{s'} P(s'|s,a)\,(R(s,a)+\gamma V_k[s'])$
10: **until** $\forall s\ |V_k[s]-V_{k-1}[s]| < \theta$
11: **return** π, V_k

Figure 2.2: Pseudocode for the Value Iteration algorithm.

We will analyse the implementability of the Value Iteration algorithm for navigating a real robot through this use-case study: The system has only one agent, CRUMB, that has to navigate through an indoor environment and reach a given target while avoiding obstacles. The robot can use any combination of the actions described in the previous section. The environment is defined as a five by five square meters space bounded by walls that can contain any number of obstacles.

CHAPTER 3

Robot simulation for estimation of problem parameters

In the previous chapter we explained why we decided to compute the state transition function P(s' | s,a), i.e., the T matrix, by simulation. This chapter presents the elements of the simulation conducted to gather data in order to make up that probabilistic model of the robot interaction with the world. This simulation and data acquisition phase provides the data for testing our Value Iteration algorithm implementations.

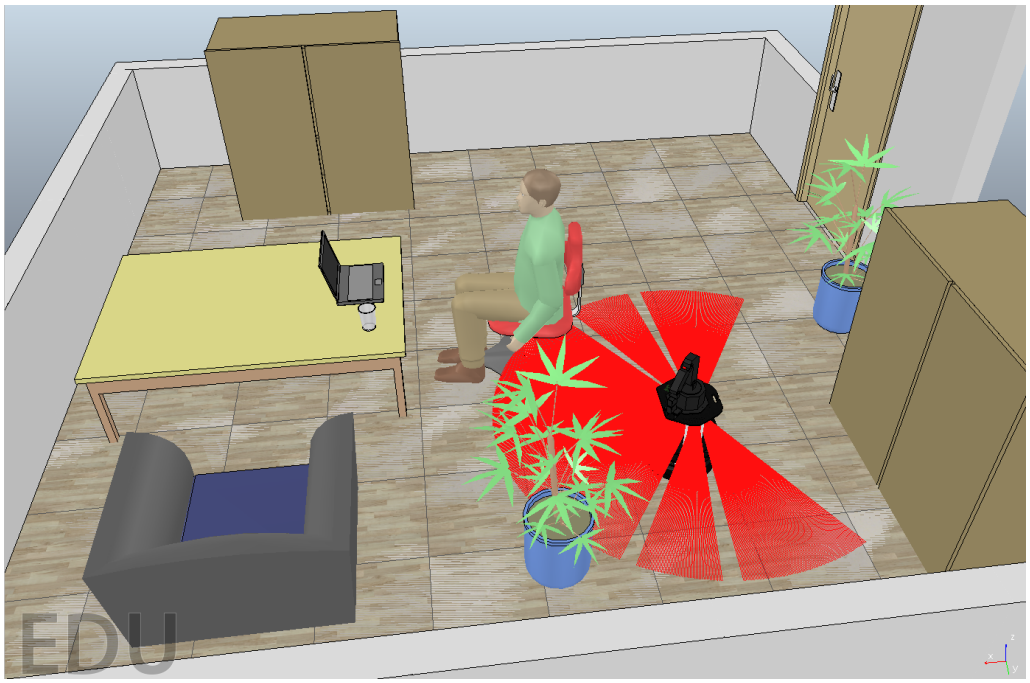

Figure 3.1: A V-REP simulation scenario example.

For this task we have used the educational version of V-REP (V-REP PRO EDU, version 3.3.2.)[14] robot simulator in order to realistically simulate the robot, which can be integrated with the Matlab (R2016b) development environment. The communication between the V-REP and Matlab is made possible with a ROS (Robot Operating System) API that is available freely for Matlab. ROS [14] is a programming framework that enables different parts of a robot system to discover, send, and receive data. Matlab support for ROS is a library of functions that allows the programmer to exchange data with ROS-enabled physical robots, or with robot simulators such as Gazebo [15] or V-REP[16].

We have chosen V-REP because it is an excellent platform for creating and simulating highly realistic robots, it is available for free for educational entities, and because it is easy to use and we had some previous experience with it. The V-REP features that we used in this work are the following [14]:

- Crosss-platform & portable content: V-REP is cross-platform, and allows the creation of portable, scalable and easy maintainable or modifiable content: a single portable file can contain a fully functional model or scene, including control code. This characteristic was specially useful for running simulations on different platforms. We used it successfully on Windows 10 and Ubuntu 14.04 platforms.
- Dynamics/Physics: It offers fast and customizable dynamics calculations to simulate real-world physics and object interactions (collision response, grasping, etc.). We use the Bullet physics library: an open source physics engine featuring 3D collision detection and rigid body dynamics. It is used in games, and in visual effects in movies. This library is often considered as a game physics engine.
- Remote API: allows to control a simulation (or the simulator itself) from an external application or a remote hardware (real robot, remote computer, etc.) and offers support for six different programming languages, including Matlab. The remote API functions interact with V-REP via socket communication. The remote API can let one or several external applications interact with V-REP in a synchronous or asynchronous way, remote control of the simulator is supported (e.g. remotely loading a scene, starting, pausing or stopping a simulation for instance).
- Can be executed in headless mode (no graphical interface): the execution of a simulation in graphical mode plus the Matlab controller consumes quite many resources. In order to reduce the simulation time it is useful to be able to execute multiple instances of the simulation in headless mode, which almost half as heavy as the graphical mode for the system.

We have developed the robot controller using a Matlab based CRUMB toolbox for V-REP [17] that allows easier communication between V-REP and Matlab than the Matlab API for ROS, and facilitates the programming process. Our controller has three functions: to send a command to the robot to take certain action, to receive the raw data from the robot sensors and convert it into a discrete state, and to store the state - action command double into a log file. The log file data is further used to compute the T matrix.

In figure 3.1 we present an example of a V-REP environment configuration for the simulation of the CRUMB robot in an indoor scenario. We will present the CRUMB robot and the configuration of the simulation in detail in the following sections.

3.1 The robot

The CRUMB robot (figure 3.2) (Cognitive Robotics sUpporting Mobile Base) [1] is a result of the CRUMB research project developed at the University of Málaga, Spain. CRUMB is a relatively low-cost, personal robot kit with open-source software for education and research, compatible with ROS, that is essentially a Turtlebot-2 mobile robot with a WidowX articulated arm. Turtlebot-2 features a ROS architecture. It integrates a Kobuki base, the Turtlebot structure, a Microsoft Kinect sensor and a netbook (running ROS). It also comes with a docking station for battery recharge. In figure 3.3 the mechanical characteristics of Turtlebot-2 are presented [1]. In the following we present its physical and functional specifications:

- Dimensions: 31,5 x 43 x 34,7 cm
- Weight: 6.3 Kg
- Maximum translational velocity: 70 cm/s
- Maximum rotational velocity: 180 deg/s
- Payload: 5 kg (hard floor), 4 kg (carpet)
- Cliff: will not drive off a cliff with a depth greater than 5 cm
- Threshold Climbing: climbs thresholds of 12 mm or lower
- Rug Climbing: climbs rugs of 12 mm or lower
- Expected Operating Time: 3/7 hours (small/large battery)
- Expected Charging Time: 1.5/2.6 hours (small/large battery)
- Docking: within a 2m x 5m area in front of the docking station

The Turtlebot-2 hardware specifications are [1]:

- PC Connection: USB or via RX/TX pins on the parallel port
- Motor Overload Detection: disables power on detecting high current (>3A)
- Odometry: 52 ticks/enc rev, 2578.33 ticks/wheel rev, 11.7 ticks/mm
- Gyro: factory calibrated, 1 axis (110 deg/s)
- Bumpers: left, center, right
- Cliff sensors: left, center, right
- Wheel drop sensor: left, right
- Power connectors: 5V/1A, 12V/1.5A, 12V/5A
- Expansion pins: 3.3V/1A, 5V/1A, 4 x analog in, 4 x digital in, 4 x digital out
- Audio : several programmable beep sequences
- Programmable LED: 2 x two-coloured LED
- State LED: 1 x two coloured LED [Green – high, Orange – low, Green & Blinking – charging]
- Buttons: 3 x touch buttons
- Battery: Lithium-Ion, 14.8V, 2200 mAh (4S1P – small), 4400 mAh (4S2P – large)
- Firmware upgradeable: via USB
- Sensor Data Rate: 50Hz
- Recharging Adapter: Input: 100-240V AC, 50/60Hz, 1.5A max; Output: 19V DC, 3.16A
- Netbook recharging connector (only enabled when robot is recharging): 19V/2.1A DC
- Docking IR Receiver: left, centre, right

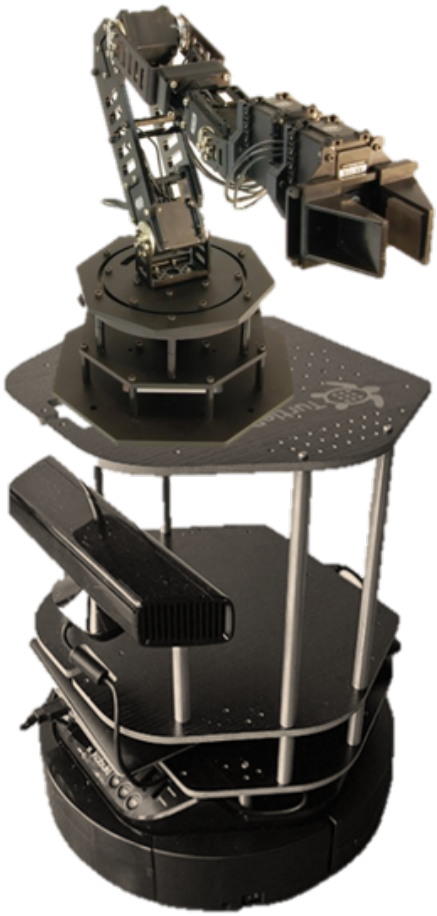

Figure 3.2: The CRUMB mobile robot

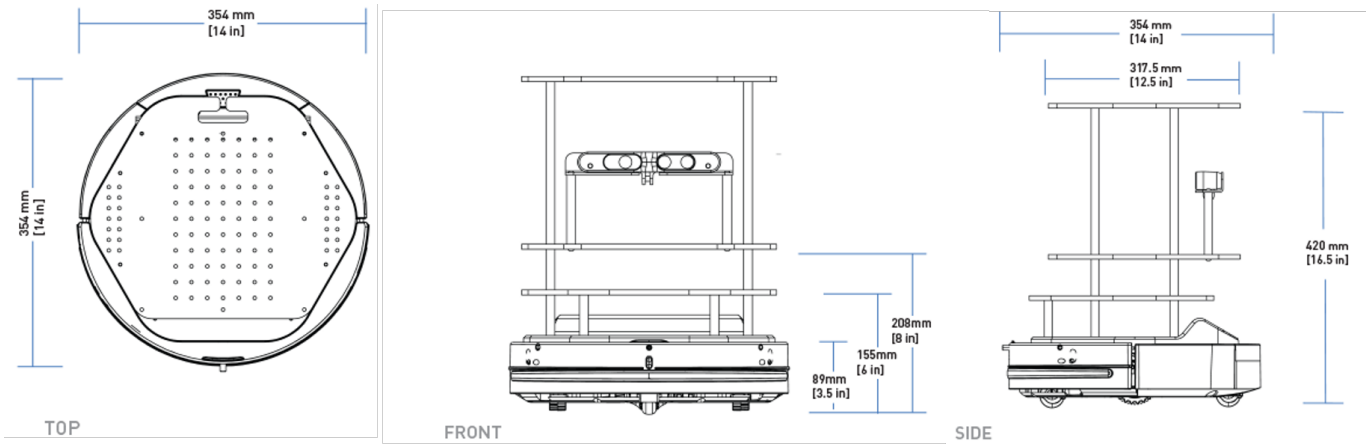

Figure 3.3: CRUMB base specifications.

- Diameter : 351.5mm / Height : 124.8mm / Weight : 2.35kg (4S1P – small)

The WidowX arm specifications are [1]:

- Weight: 1400 g
- Vertical Reach: 55 cm
- Horizontal Reach: 41 cm
- Strength (no rotate):
  - 30 cm -> 400 g
  - 20 cm -> 600 g
  - 10 cm -> 800 g
- Gripper strength: 500 g Holding
- Wrist Lift Strength: 400 g
- Servos:
  - 2x MX-64 Dynamixel Actuators
  - 2x MX-28 Dynamixel Actuators
  - 2x AX-12A Dynamixel Actuators
- Arbotix Robocontroller:
  - ATMega644p Microprocessor
  - 8 Analog & 8 Digital IOs
  - Physical, Xbee Wireless, & USB/TTL Serial control options
  - Arduino IDE compatible
  - Custom firmware capable
  - ROS Ready

Our robot will be represented by a model in V-REP robot. The V-REP CRUMB model also includes a Hokuyo URG-04LX scanning laser rangefinder, featuring a detectable range from 20mm to 4000mm, 100 msec/scan and with a 240° scanning range with 0.36°

angular resolution. We use the Hokuyo laser sensor for the rangefinder component of the robot state. Fig. 3.1 can give you an idea about how it is placed and about its reach.

The CRUMB robot model has been previously designed in the System Engineering and Automation Dpt. with V-REP (Virtual Robot Experimentation Platform) to accurately match the physical description of the real robot. V-REP allows us to easily simulate the robot behaviour in any environment setting when we give it specific commands. The model can be controlled via an embedded script, a plugin, a ROS node, a remote API client, etc., and the controllers can be written in C/C++, Python, Java, Lua, Matlab or Octave [16]. Our controller is developed in Matlab as a remote API client and uses the programming toolbox developed by Iván Fernández Vega in his final bachelor thesis [17] in collaboration with the CRUMB research team. This toolbox is an extension of the Matlab Robotics System Toolbox and allows us to easily connect to the robot and run the same program (controller) on both the simulator as on the actual robot.

3.2 The simulation

The general workflow of the simulation is represented in figure 3.4. As we may see in the diagram, we use a Matlab controller client that sends commands to the V-REP simulated CRUMB robot. We have simulated the robot in multiple scenarios; this is indicated by the outer loop in the diagram. It is important to remark that this is a distributed system: the controller could be running on a remote computer and send the commands to the simulator or/and to the real robot through the network.

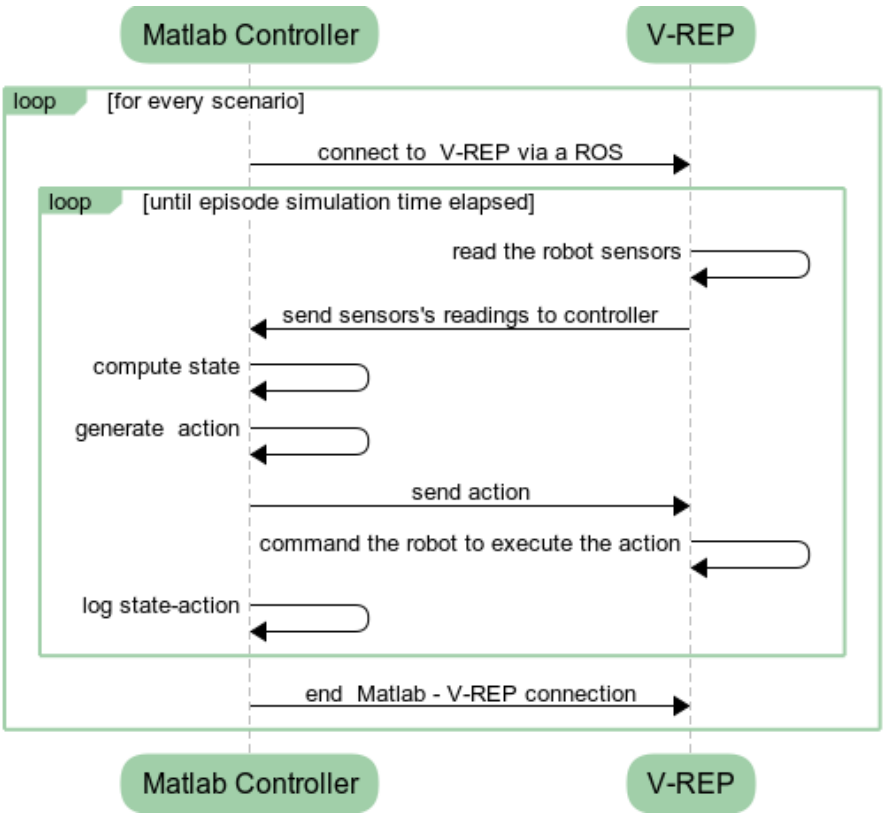

Figure 3.4: Matlab - V-REP simulation workflow.

Before attempting to start the communication between the Matlab controller and the V-REP simulator we have to initiate the V-REP simulation of the CRUMB model and open a port for incoming client connections. Only afterwards we may successfully connect

the Matlab controller client to the V-REP simulated robot via a ROS (Robot Operating System) based interface.

In our setting, we simulate 30 episodes of 2000 seconds each for every scenario. The beginning of a new simulation episode is marked by the change of the target position to a new random one. The simulation of an episode corresponds to the second loop in the diagram: while the simulation of the current episode is not ready (the 2000 s have not elapsed), the simulator computes the current sensor readings for the robot and send it to the Matlab Controller via ROS publishers and subscribers, and at its turn, the Controller computes the current state of the agent from those sensor readings, and generates and sends a command to the V-REP simulator to make the robot execute a given random action. Then the controller stores in a log file that state - action tuple. When the simulation of the current episode has ended, the position of the target changes.

When all episodes have been simulated for the current scenario the Controller client ends the connection. We can manually launch the same process for another scenario once the previous is ready or we can launch several simulations for different scenarios in parallel if we have a resourceful enough computer. We have used both methods in our work.

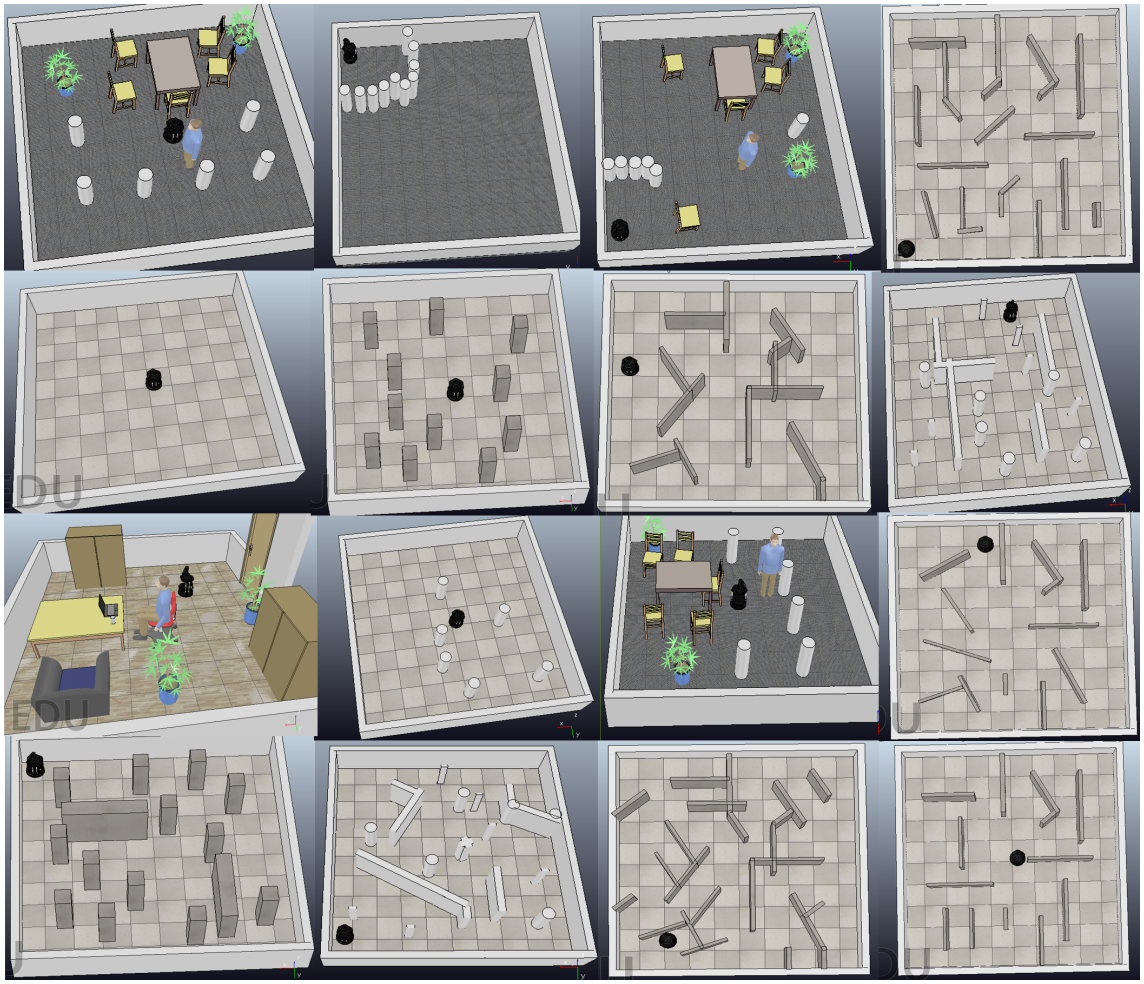

Figure 3.5: Several V-REP 5x5 m scenarios.

In figure 3.5 we can see a sample of the scenarios used in simulation. They are all 5×5 m room like scenarios, surrounded by a one meter tall wall that prevents the robot from "seeing" and falling outside of it. They all contain static objects and obstacles placed in a manner that promotes the exploration of all the state space. Each scenario was used

several times in simulation with small modifications on the objects positions and orientations and in the robot initial pose.

3.2.1. V-REP simulator configuration

In this section we discuss the decisions that we made for the simulated experiments, such as the duration of an action, the duration of the experiment, the V-REP parameters settings, etc.

We needed to study what configuration is more appropriate for the simulation so that we explore all the reachable states of the robot while making a reasonable compromise between the precision of the simulation and the duration of the simulated experiment. In order to achieve this we have made a study of the simulation time required to take an action by varying two parameters in the V-REP simulation environment and comparing the distribution of the physical time needed to simulate an action with those settings. The mentioned parameters are Dt - the simulation time step, and pps - the simulation phases per frame. Dt indicates how often the simulator computes the dynamics of the system; a smaller Dt means a more accurate simulation. pps sets how many simulation time steps are executed per frame. The rendering operation will always increase the simulation cycle duration, slowing down the simulation speed. In this case, a higher pps indicates that the the rendered simulation has inversely proportional less frames per second, thus poorer image quality but faster simulation. Our top priority is having the dynamics simulation as accurate as possible.

Table 3.1: Average execution time of an action in a PC with 16 RAM and i5 processor.

Dt (ms)	***ppf***	***Duration (s)***	***Observation***
5	**1**	**8.5**	The most precise simulation
10	1	3.7	
	256	3.6	
25	**1**	**1.82**	Acceptable compromise
	128	1.68	
50	1	1.17	
	64	0.84	
100	1	0.58	
	32	**0.56**	The fastest simulation

For this experiment we have set the action duration in the simulation to $0.5s$ and we simulated for $50s$ (simulation time), the equivalent of 100 actions. The executions have been made in a 16 GB RAM and a i5 processor computer. We remark that by fixing one of the two parameters we limit the range of the other. The results of the experiment are shown in table 3.1. As it can be seen, we have fixed the Dt value and then measured how much time it takes to simulate a 0.5 s action (Duration column) for the minimum and maximum possible values of ppf. We can notice that for the largest possible Dt ($Dt = 100ms$) we have almost real time simulation of an action; unfortunately, this setting does not produce an accurate enough simulation of the dynamics of the system. On the other extreme we have the 5 ms Dt, that offers the most precise simulation, but with the drawback that simulating a sole action of 0.5 seconds would require the simulator to work for 8.5 seconds on average, which is totally unacceptable. We have made a compromise between simulation accuracy and simulation duration and chosen the combination of $Dt = 25ms$ and $ppf = 1$ for that. In figure 3.6 we show the duration distribution for

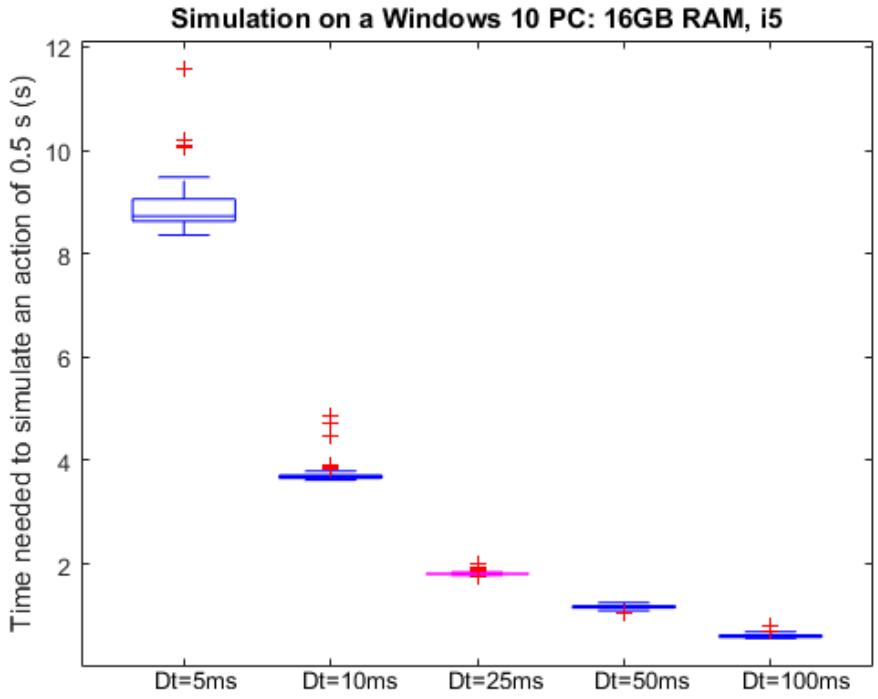

Figure 3.6: Simulation time boxplots for 100 actions of 0.5s for pps = 1 and varying Dt.

all possible configurations of Dt and $ppf = 1$. The normal distribution of the duration reinforces our previously made choice.

In addition to this, after implementing the algorithm and studying its behaviour we observed that a 5 ms action does not make the model of the system fully comply with the markovianity requirement; as a result, we slightly adjusted how the actions are taken. This matter will be discussed later in the section about "Considerations regarding the simulated experiment".

3.2.2. Deciding the simulation duration

The first question that comes to mind about the simulation is "For how long should we simulate?" A partial answer could be "Until every single state was visited often enough". Let us say that 100 times is enough times. In this case $SimulationTime = Ta \times |S| \times |A| \times 100s$, where Ta represents the time needed to simulate the robot taking an action from A; Ta varies in function of the PC hardware and the simulator settings.

So far we know that the number of actions, $|A|$ is fixed to 6, because we have 6 actions, but what about state space dimension, $|S|$? The state space dimension depends on the discretization granularity of the continuous state space.

Having all this in mind we have to decide on the state discretization and figure out how long should an action take so that the total simulation time does not surpass two weeks, which is the longest time we can dedicate to simulation. For this task we have a server that can run two instances of the simulation in parallel at most, and a computer that can only run one at a time and is not always available.

State discretization granularity

In table 3.2 it is presented a summary view of the state representation in our MDP model. It is composed of four elements: d, t, a, and r. The rangefiner, r, has a number NR of nested elements. The value for NR can be no higher than the physical resolution of the Hokuyo sensor, which is equivalent to its total number of laser beams, equal to 684 (i.e. the number of times the the angular resolution of the sensor, 0.36 degrees, is encompassed in the scanning range of the sensor).

Table 3.2: MDP state representation.

State		*Discretization*
d	distance from robot pose to target pose	ND
t	robot orientation, θ	NT
a	angle between θ and the target pose	NA
r	set of rangefinder distances to surrounding obstacles	NR $\times$ NG

We have tried a number of combinations for this discretization and computed how much time it takes to gather enough simulation data to compute the T matrix. As we mentioned previously, $SimulationTime = Ta \times |S| \times |A| \times 100$. We know that $|A| = 6$ and that it takes around 1.8 s to obtain the transition (Ta = 1.8 s). The corresponding duration time of the chosen discretizations of the state can be seen in table 3.3. The fist option for our discretization is shown in the third column (ND = 20, NT = 72, NA = 32, NR = 5, and NG = 10) and it corresponds to an ideal discretization that would realistically approximate the real continuous state of the robot with little error. Unfortunatelly this discretization would require literally a lifetime of simulation (about 80 years of continuous simulation) with our current resources. By lowering our standards on how accurate we want our model to be (ND = 10, NT = 32, NA = 16, NR = 5, and NG = 5) we still have the same problem - the simulation requires more than four years of continuous execution, which is not feasible. Thus, we decided to give up on the realism of our model and make a quite rough dicretization of the state (ND = 4, NT = 5, NA = 4, NR = 3, and NG = 3), that would require only nine days of continuous simulation.

Table 3.3: Different discretization options and their simulation durations for a $Ta = 1.8s$.

State	*Discretization options*			
d	ND	20	10	**4**
t	NT	72	32	**5**
a	NA	32	16	**4**
r	NR $\times$ NG	5 $\times$ 10	5 $\times$ 5	**3 $\times$ 3**
Simulation time (days)		28800	1600	**9**

3.3 Considerations regarding the simulated experiment

There are two very important requirements that we must be aware of for the simulation so that the produced policy by the VI algorithm is suitable for the real robot, not just for the approximation given by the MDP model: first, we should obtain a model of the robot interaction with the environment that is as accurate as possible; second, we need to model our system so that it satisfies the Markov property.

3.3.1. Requirement 1: obtain sufficient experimental data to compute the T matrix

In order to obtain a good approximation of the state transition function we need to make sure that the agent visits every single state and takes every possible action from it enough times, that is, until we have visited every state-action-state triple a statistically meaningful number of times. We attempt to achieve the complete exploration of the state space through simulation by using a large number of scenarios that are as varied as possible.

A sampling method is called biased if it systematically favours some outcomes over others. The idea here is to avoid the visit of some states over other or choosing some action more frequently then other during the episodic exploration experiments. For this purpose we have taken the following measures:

- Prepare multiple simulation scenarios that are as varied as possible (figure 3.5).
- Generate the actions randomly so that we do not promote one action over another. There is a fair chance of trying all possible actions for all possible states for a comparable number of times.
- Periodically modify the target position (random new position inside the scenario perimeter) during simulation, so that the robot can get to all reachable states from the given scenario.
- Evaluate the distribution of state-action visits, and, according to the evaluation, create new simulation scenarios to compensate for the state-action cases that are not properly covered (visited). We evaluate the uniformity of the state-action visits distribution by visualising the number of transitions realised through the simulation. We will explain how it works next.

Visualisation of the transitions

To be able to represent the transitions from a state to the next in a 2D image we have associated to every state a unique id. The state id is computed as follows:

```
StateId = (s.t-1)   * NG * NG * NG * ND * NA +
          (s.r(1)-1)* NG * NG * ND * NA  +
          (s.r(2)-1)* NG * ND * NA +
          (s.r(3)-1)* ND * NA +
          (s.d-1)   * NA +
           s.a;
```

Using this representation, every state is given an integer number between zero and the total number of states minus one:

$StateId \in [0, NS-1), NS = NT \times NG \times NG \times NG \times ND \times NA$

The transitions matrix T is three dimensional, so we have decided to use heatmaps to represent the number of transitions for every state-action-state triple.

We asses on the uniformity of the state space exploration during the simulation and figure out where there is a problem by visualising the transitions:

1. Global view of the transitions:
    - First, we check that we have taken every possible action from every state at least once by drawing a 2D graph for every action a where the the axes represent the current state id and the next state id. An example can be seen in

figure 3.7. A blue dot on the graph indicates that there is at least one transition from s_1 (current state) to s_2 (next state) by taking action a, where a takes values from 1 to 6, each corresponding to action *stay*, *move forward*, *move to the left with displacement*, *move to the right with displacement*, and *move back* respectively. The value of nz indicates the number of nonzero state-action-state triples (the number of blue dots). This test is passed if for every state s_1 we can reach one or more states s_2 by taking any action.

- Next, we look at similar graphs to the ones from figure 3.7 but using a range of colours to quantify the number of transitions, not only to indicate that there is a transition. We can check that the robot has taken every possible action from every state at least 100 times by drawing a heatmap for every action a. In this representation the axes represent the current state, s_1, and the next state, s_2 (like in the previously described graph) and the color indicates how many times we have had a transition between s_1 and s_2 by taking action a from s_1. Everything is OK if the color indicates more than 100 transitions for every state-action tuple. An example of such graphs is shown in figure 3.8.

2. Decomposed view of the transitions: In some regions we cannot tell if there are enough transitions or if there are any transitions at all, and, most importantly, why. For these cases we use a new variation of the graphs described previously that details if there is at least one transition (like in the graph from point 1) and how many transitions are there (like in the graph from point 2), but now for every state component and action. These graphs can also give us a clue about why we have few transitions from one or more states or why some states are unreachable. In figures 3.12, 3.13, 3.14, 3.15, 3.16, and 3.17 can be observed such graphs.

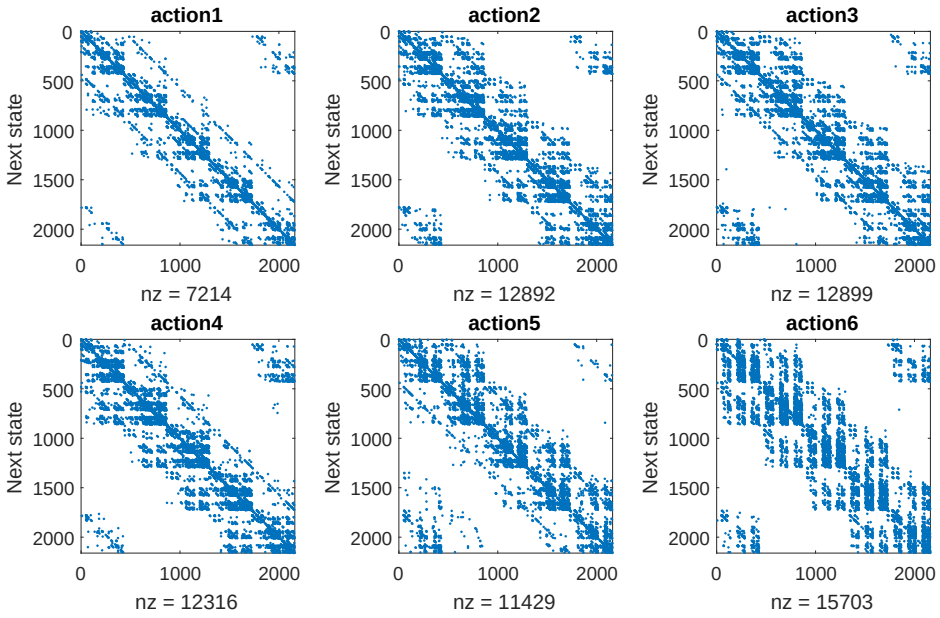

Figure 3.7: Transitions existence from the current state to the next state.

By inspecting the corresponding graph (Fig. 3.7) we can confirm that the first test has been passed. We also noticed that the vast majority of the transitions are actually auto-transitions (the great majority of the transitions are concentrated on the diagonal of the graphs). This is explained by the rough discretization that we had to use. Even though the robot clearly moves, the imprecise discretization makes the the robot end in the same state most of the time.

Whenever we have fewer or no transitions for some state-action tuples than for others than normally means that our scenarios promote visiting some states more than others, so we need to create new ones accordingly to compensate the unbalance. To better identify where exactly is the problem we look at the number of transitions grouped by state components (Figs. 3.12 - 3.17). We do so because it is hard to figure out whether all the states were visited at least once and why some states were visited few times only from looking at the global state transition graphs. The state transitions are grouped by the discrete values of its components: theta, the three rangefinders, the distance to target and the relative orientation of robot to the target.

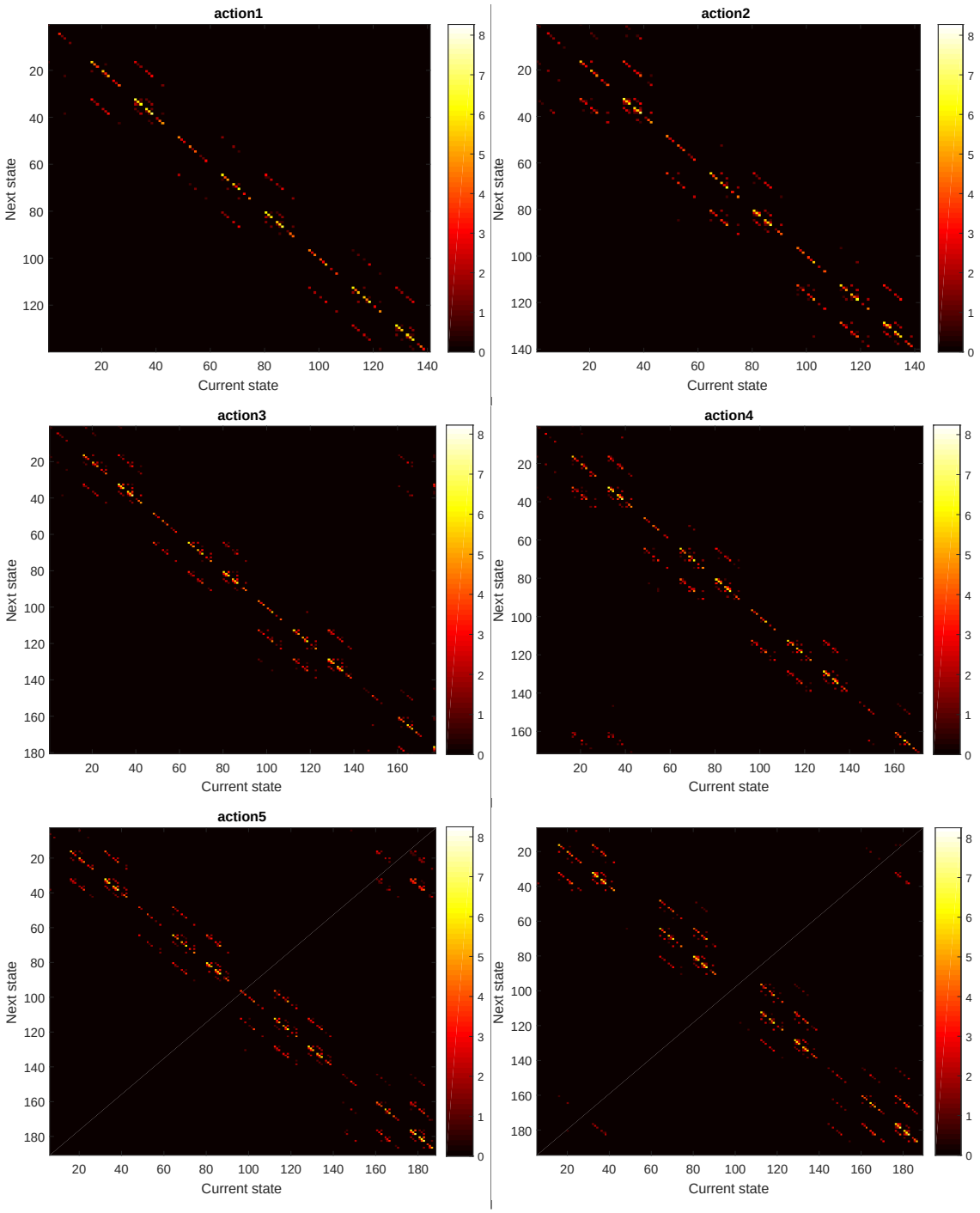

Figure 3.8: Transitions number from current state to next state using log2 scaling. Every image represents only a section of the whole graph to facilitate the visualisation of the transitions.

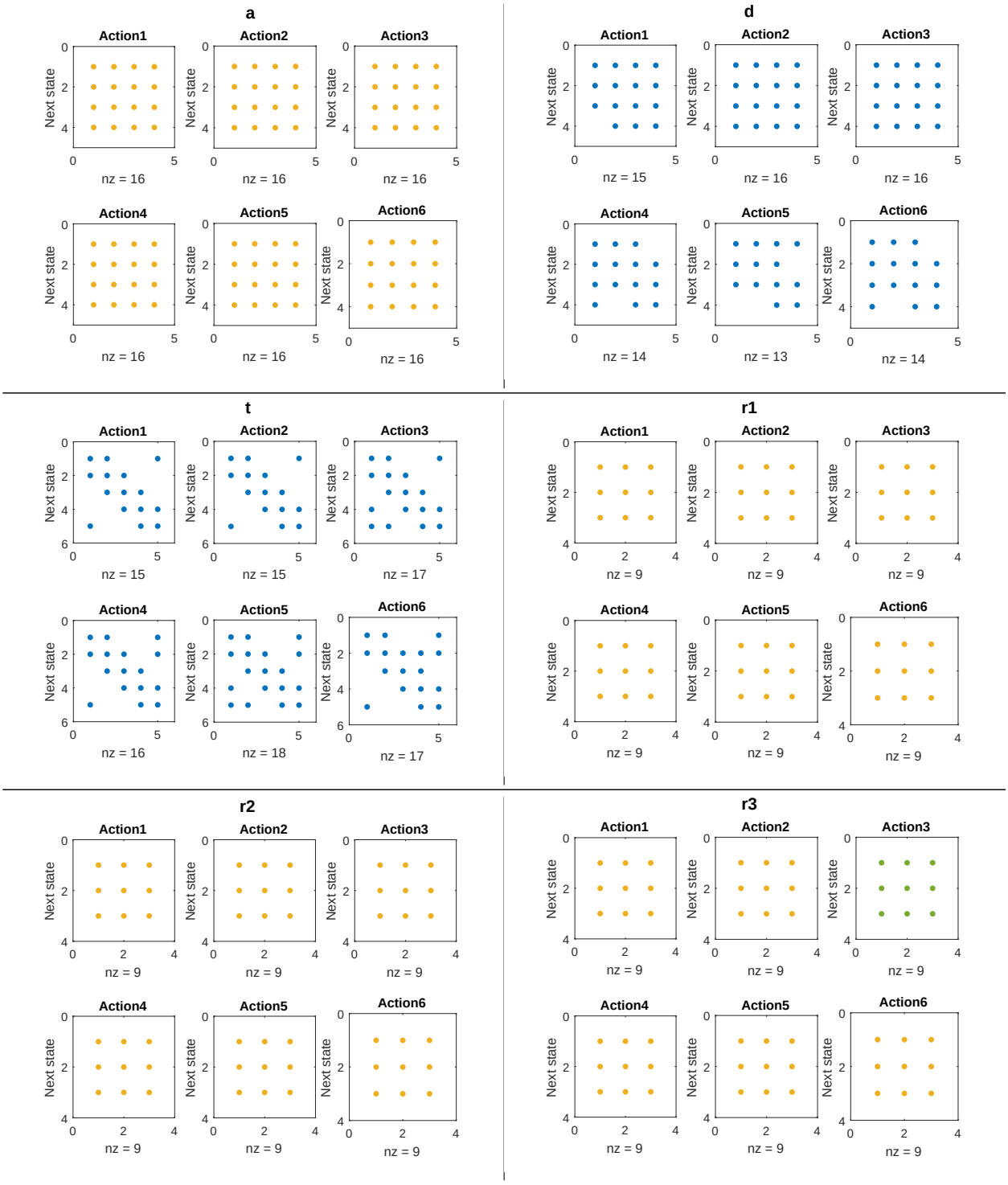

Figure 3.9: Detailed transitions existence from current sub-state (state component) to next sub-state. We use orange markers for the sub-states that are fully visited for all possible actions and blue markers for the ones that are not.

An example of the use of these graphs is the following: in early stages of the simulation we realized that the states with the highest id had no or very few transitions in comparison with the rest of the states. By looking at the graphs of the transition count for the state components (like in Figs. 3.12 - 3.17) we noticed that there were almost no transitions corresponding to the d component of the state for any action for $d = ND - 1$. Fig. 3.10 illustrates the case. In practice this means that the robot did not explore the case when the target is placed in one corner of the scenario and it is placed in the opposite

corner of the scenario, that is at the largest distance possible. Having observed this, the solution to balance the number of transitions from the states with high state id is obvious: explicitly include a simulation episode with the robot initial position in one of the corners and the target placed in the opposite corner for every simulation scenario. Fig. 3.13 shows that after forcefully including the new scenarios the problem is solved.

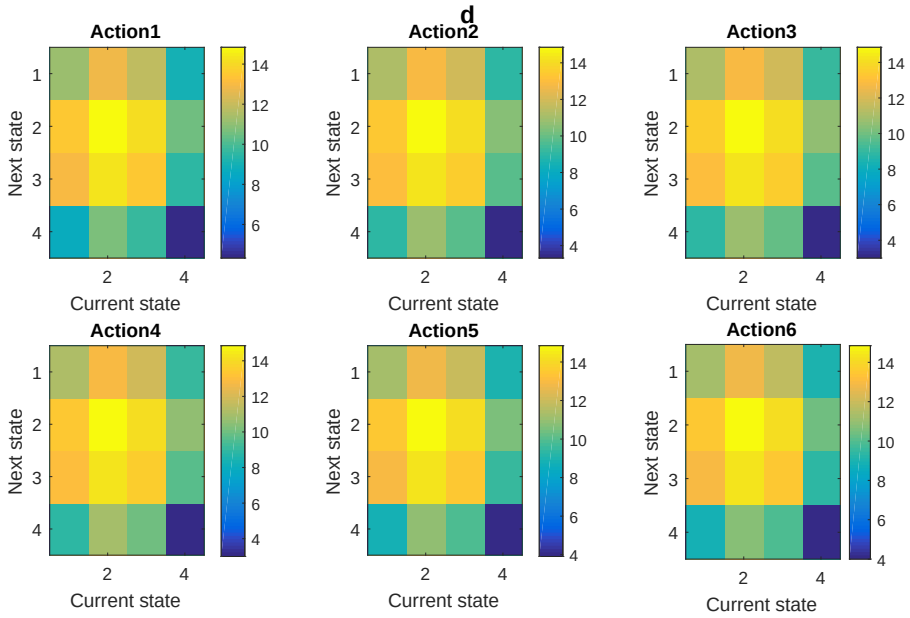

Figure 3.10: Example of biased simulation: the robot hardly ever explores the case when the target is the farthest possible ($d = ND - 1$).

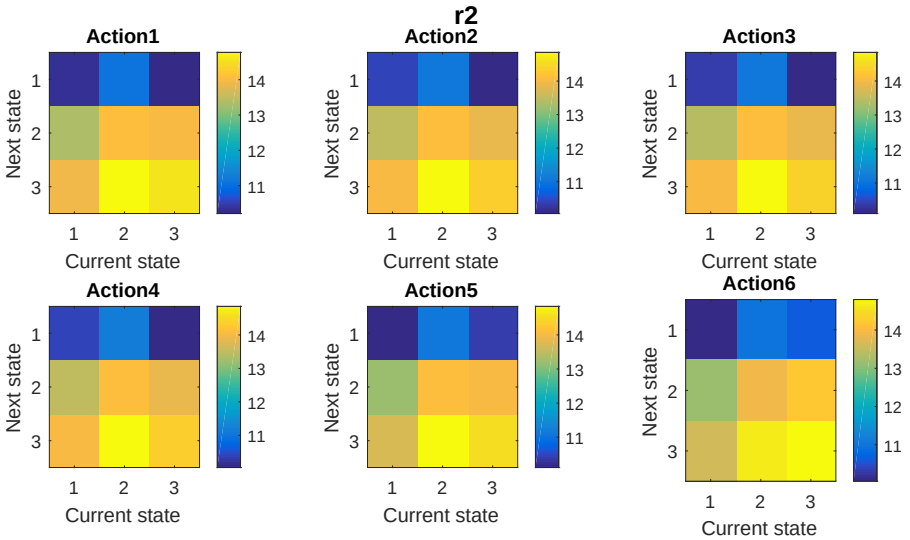

Figure 3.11: Example of biased simulation: the robot experiences the case when it hits or gets really close to and obstacle after taking any action far less than when it has an obstacle farther away.

Another case where we noticed that our state space exploration was biased and we fixed it by including new scenarios is the following: by looking at the total number of visits of all state components corresponding to the rangefinder (close obstacle-1, middle range obstacle-2, far obstacle-3) we observed that the robot reaches a state with a close obstacle far less than the other cases (middle range and far obstacle). Fig. 3.11 illustrates the case. Our solution for this problem is to include more stretched, maze-like scenarios

so that the robot has more of opportunities to hit or be very close to obstacles. Fig. 3.16 shows that after including new scenarios with stretched paths the problem is solved.

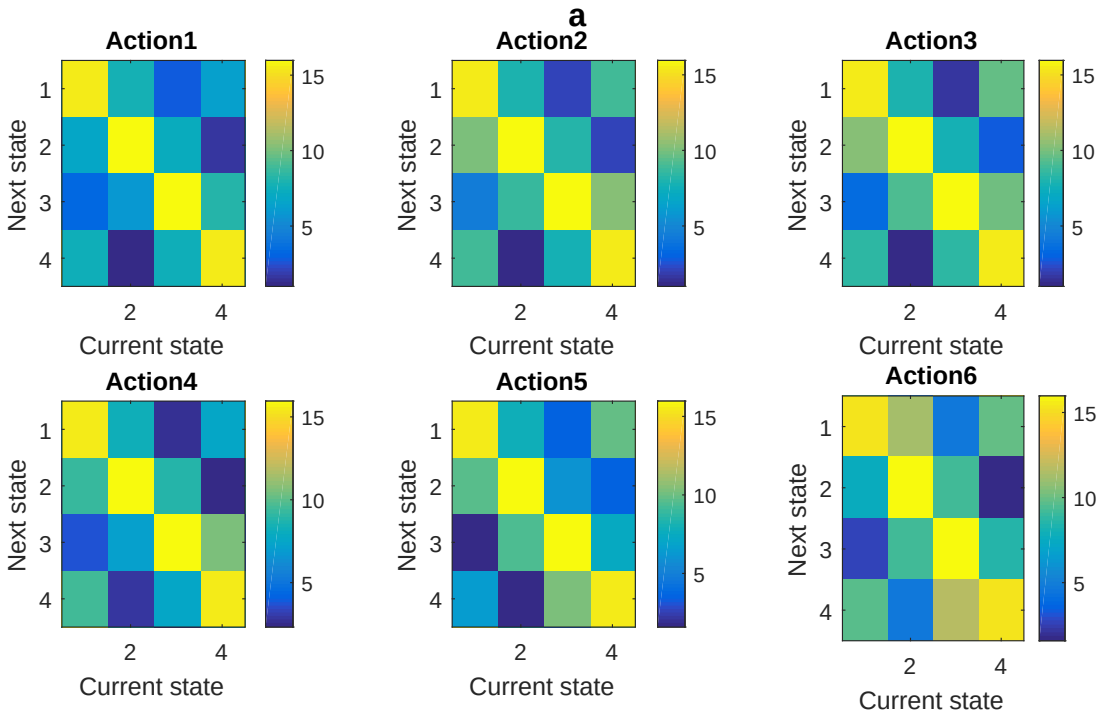

Figure 3.12: Log2 scaled transition count for *a* state component.

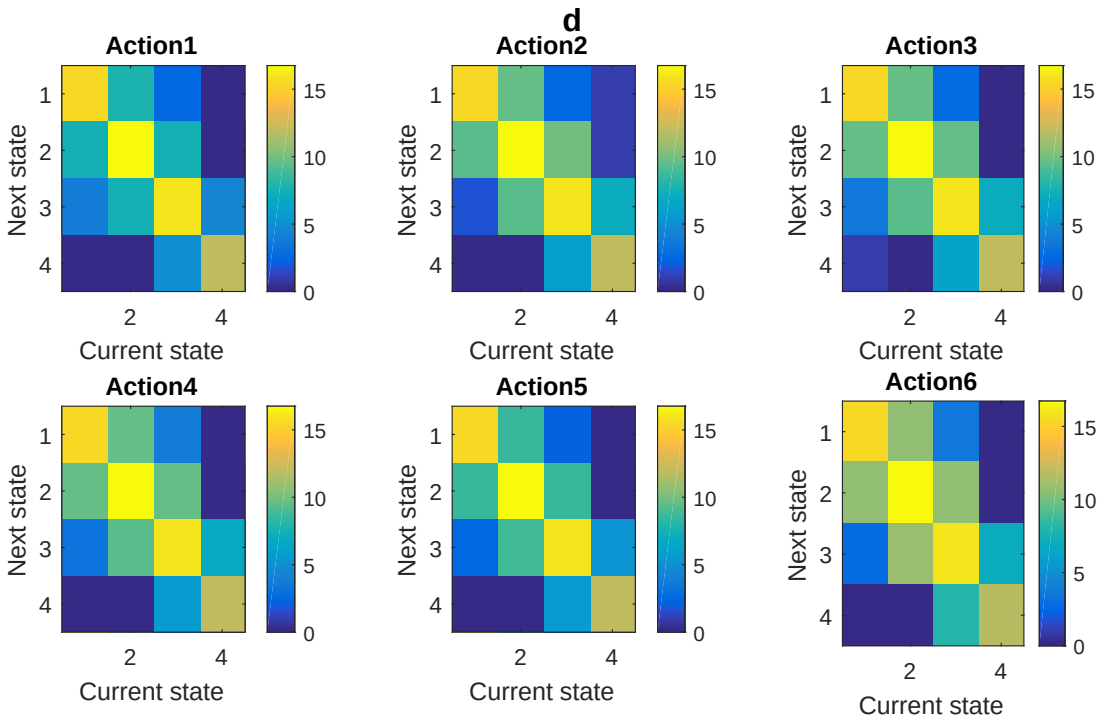

Figure 3.13: Log2 scaled transition count for *d* state component.

From the decomposed view of the state components transitions (Figs. 3.12 - 3.17) we can reach the conclusion that all states were visited and the robot tried every possible action from each one of them enough times.

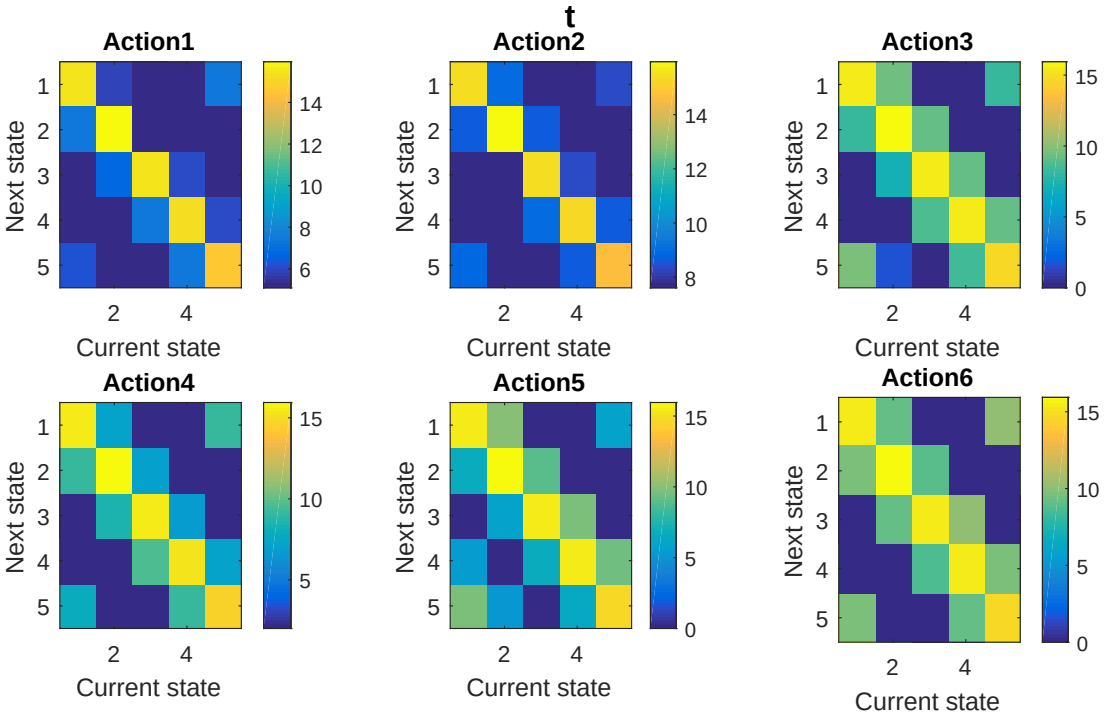

Figure 3.14: Log2 scaled transition count for *t* state component.

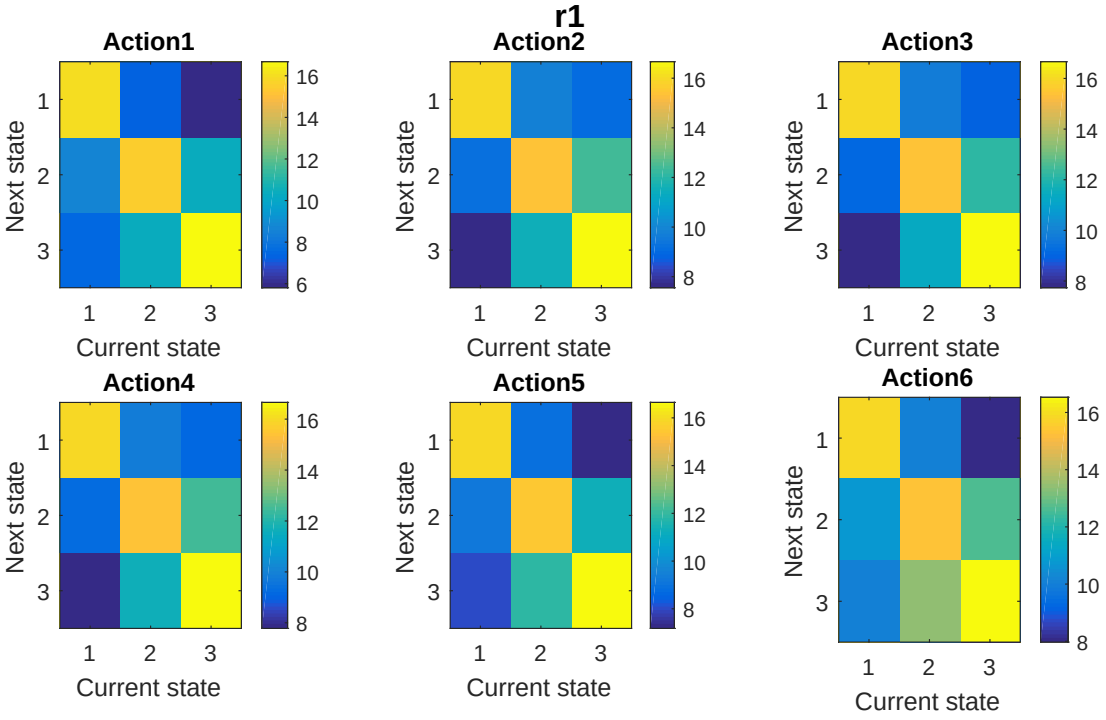

Figure 3.15: Log2 scaled transition count for *r(1)* state component.

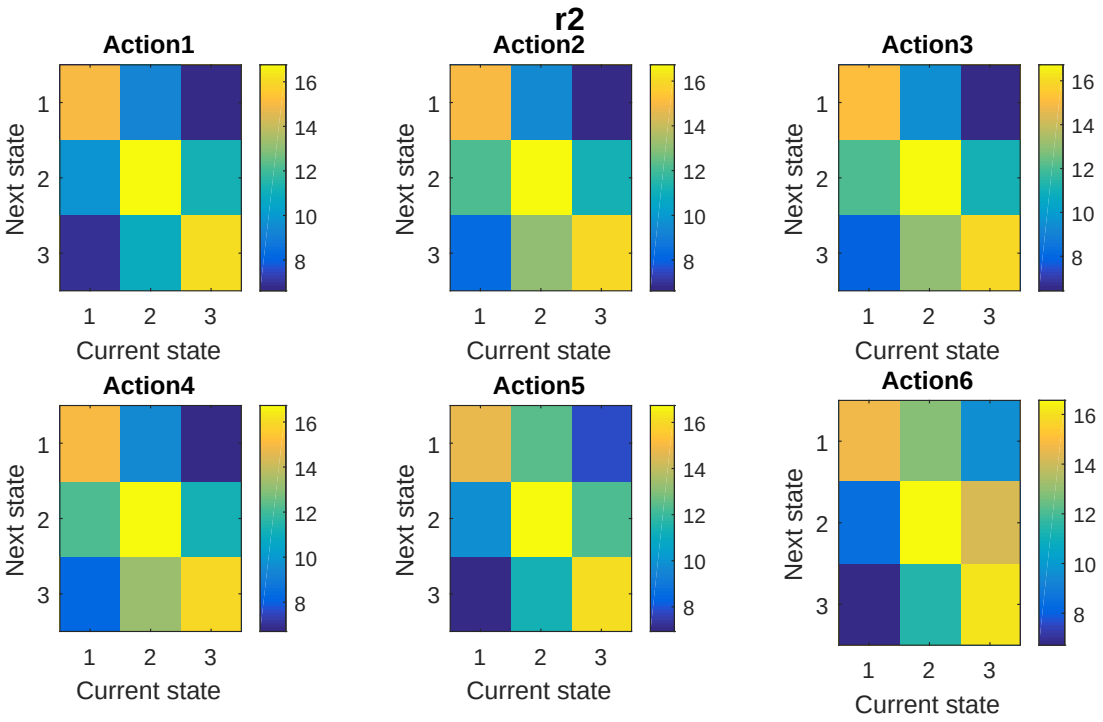

Figure 3.16: Log2 scaled transition count for *r(2)* state component.

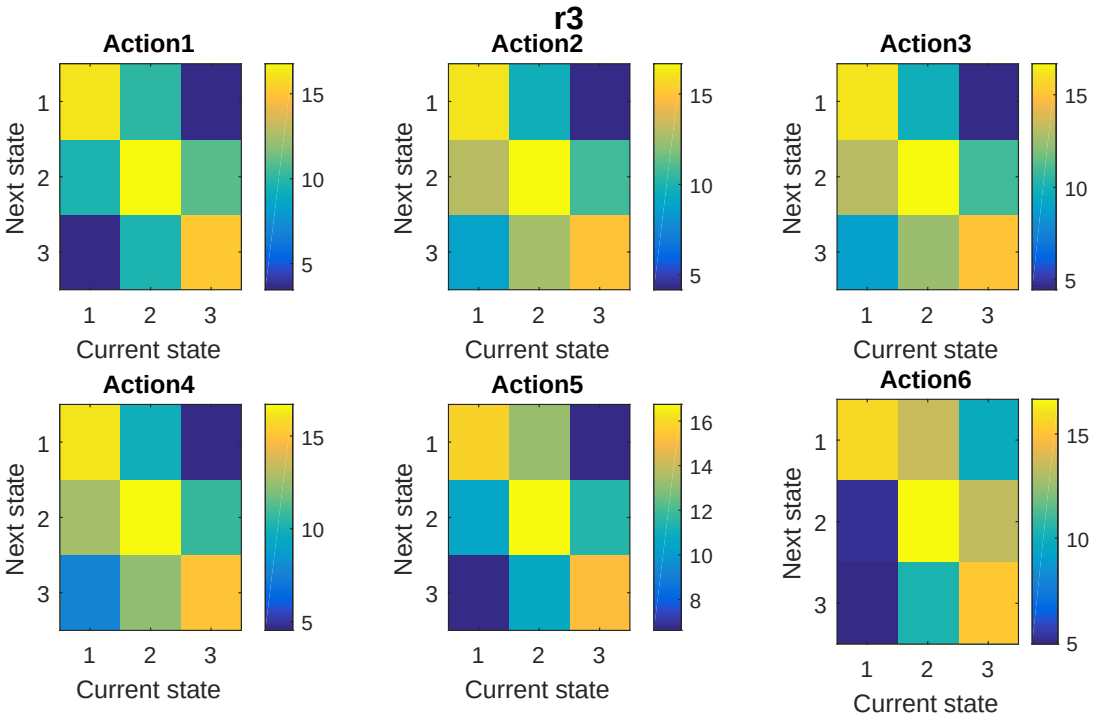

Figure 3.17: Log2 scaled transition count for *r(3)* state component.

3.3.2. Requirement 2: the MDP model must comply with the Markov property

It is important to know whether the simulation provides independent states or not mainly for the applicability of MDP algorithms. In a MDP the probability that the process moves in a new state s' is supposed to be influenced only by the current state s and the chosen action; in other words, the transition must be conditionally independent on the previous states and actions. We have thought of a way to approximately tell the "markovianity" of our system by measuring the effect/influence of the previous action on the current action.

Initially we had decided to simulate our experiment using actions that last for 0.5 seconds. We started to worry about the fulfilment of the Markov property only when we noticed an odd distribution of the transitions that are not on the diagonal (the current state and next state are one and the same) for action 1 (stay still). At first we thought that this behaviour could be explained by sensor's noise. So we decided to take a closer look at the number of transitions for this case to make sure that there is not an abnormally large number of transitions that could not be justified by noise alone. Once we realised that our choice for the action duration was not appropriate for an MDP we had to discard almost three days worth of simulation data and start afresh with actions like in the option three described in the following.

We measure how well our model complies with the Markov property by computing the probability of a state to end up in the same state after taking the action that does nothing. The higher the probability, the more markovian the system is considered to be. Equation 3.1 shows how we compute the markovianity (M) of our MDP model. The left term of the numerator, S_x, is the initial state, while S_y is the following state after taking action 1 (stay still) - A_1 in the equation.

$$M = \frac{\#(S_x - A_1 -> S_x)}{\#(S_x - A_1 -> S_y)}, \forall x, y \epsilon [1 : NS], M \epsilon [0 : 1] \tag{3.1}$$

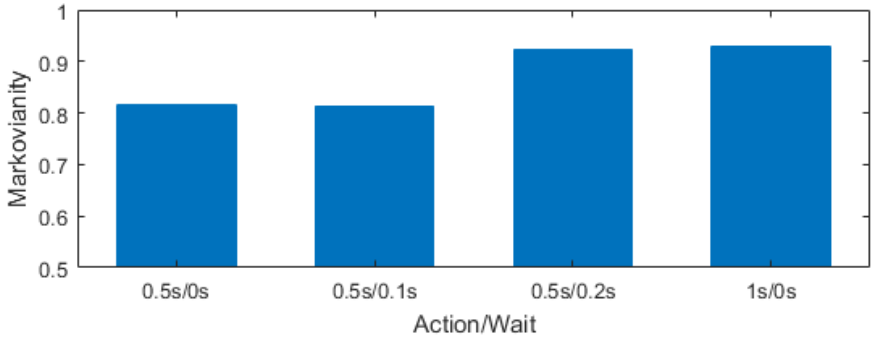

Figure 3.18: Markoviaity measure for different action/wait duration.

In our experiments we observed that the previous action influences somewhat the current action effect, the first bar from the chart), therefore the transition is not fully independent of the previous states and action. This happens because the robot keeps its dynamics from the previous action and that influences the current action. This influence would be neglectable if every action would last longer, but we cannot afford making the action last much longer because the simulation would require much more time. A workaround for this problem is to introduce a small gap between two consecutive actions, when the robot does nothing, in order to attenuate this undesired effect. In figure 3.18 are presented the four options that we have considered, along with their effects on the markovianity of the system:

- Actions that last for 0.5 seconds with no pause between consecutive actions.
- Actions that last for 0.5 seconds with a 0.1s pause between consecutive actions.
- Actions that last for 0.5 seconds with a 0.2s pause between consecutive actions.
- Actions that last for 1 second with no pause between consecutive actions.

We notice that for the first and second options the robot remains in the same state after taking the action to stay still only for about 81% of the times. Our system has uncertainties induced by the sensor measurements and noise so, it is expected to have some error. The problem is that 19% is too much error to be justified by imperfect sensor measurements and noise alone. Our hypothesis is that the extent of the error is induced by the influence of the previous action on the current one through the dynamics of the robot. In other words, even though the robot is given the command to stay still, it is has not stopped completely and keeps moving for a while due to the previous action (whenever the previous action is a movement action). This hypothesis is supported by the third and fourth options that we have experimented with: by increasing the gap between consecutive actions to 0.2s we decrease the error to less that 8%, which we can work with. We obtain a similar effect by duplicating the duration of the action from 0.5s to 1s and including no gap between consecutive actions. We chose to use the third option, 0.5s action and 0.2s gap, for the final experiment because it is shorter and involves less execution time overall.

CHAPTER 4
Implementations of the Value Iteration algorithm

In this section we will cover five topics:

- The definition of the data structures used for the state and transition probability matrix.
- A description of the sequential version of the Value Iteration algorithm.
- Optimizations using SPMD parallelism on a multi-core with OpenMP and TBB.
- Optimizations using SIMT parallelism on a GPU with OpenCL.
- Load balancing strategy for the heterogeneous implementation.
- Comparing the performance of the presented implementations of VI algorithm.
- A discussion about the correctness of the Value Iteration implementations.

4.1 Data representation

In order to decide how to represent and store our data we have to answer the following questions: *What kind of data do we have?*, and *Do we have any limitations of storage capacity?*

Answering the first question, the type of data that our algorithm manages is as follows. There is the state that we have to represent by its components: θ, d, r , and a. Each component will have discrete values determined by the chosen discretization $[0..NT-1]$ for θ, $[0..ND-1]$ for d, etc.

The rewards matrix R contains $NS \times NX$ real values, where NS is the total number of states and NX is the number of actions. It is a full matrix, because the algorithm requires the reward for taking every possible action from all reachable states. Q has too $NS \times NX$ real values and it is a full matrix. The policy, π, and the policy values, V, are one dimensional arrays of NS values. The first one contains integers in the interval $[0..NX-1]$ and the second one contains real values. Both of them are full arrays. The transition matrix T contains $NS \times NX \times NS$ real values. It is a sparse matrix because when parting from any initial state with any possible action it is impossible to reach every other state. There will be approximately one to ten reachable states from any state-action combination while the remaining states are impossible to reach (for being too far).

Do we have limitations of storage capacity? The answer is yes; our Heterogeneous Computing Environment (HCE) is bound to both memory and computation limitations.

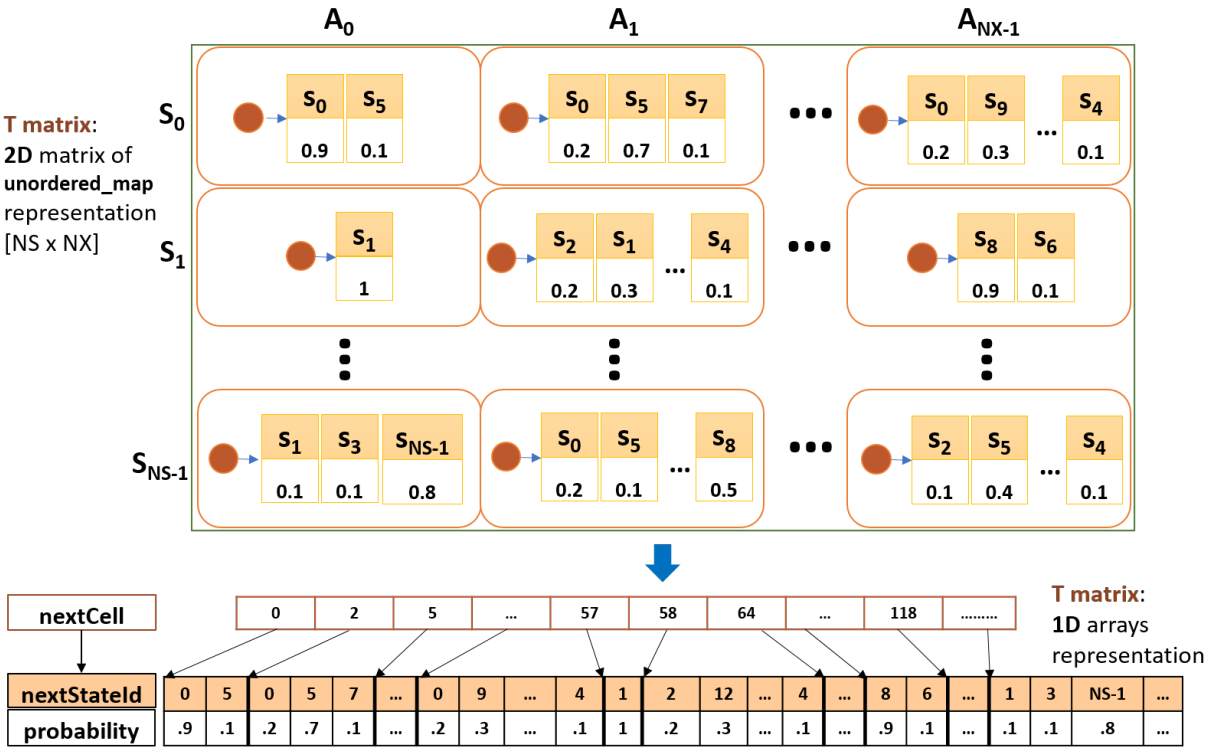

Figure 4.1: T matrix sparse representations.

Having answered these two question, it is clear that we need to decide carefully how to represent and store the T matrix, that is potentially large (depending on the granularity of the state discretization). Using a 3D (three dimensional) matrix (States x Actions x States) would be suboptimal because T is sparse (i.e., the great majority the values are zeros). For the sequential and OpenMP versions we could use a $NS \times NX$ matrix, each cell containing a map (`unordered_map` in C++). Every map contains a list with pairs of <*(next)stateId, probability*> for the reachable states from the state-action corresponding to the map cell, as in figure 4.1, upper part. All pairs corresponding to a probability of zero are not included in the map.

The previously described representation is not appropriate for OpenCL, though. This is because the OpenCL (v1.1) kernel allows only built-in scalar data types, 1D arrays (pointers to the built-in datatypes), and well aligned structures as arguments. As a result, the T 3D sparse matrix will have to be represented using 1D arrays, as in figure 4.1, lower part. This 1D representation is composed of three arrays:

- *probabilty*: contains the transition probabilities from the T matrix, ordered row-wise (first all the transition probabilities for S_0, then for S_1, etc.) The transition probabilities for a state-action pair always sum up 1. This can be seen both in the previous representation (2D matrix of *unordered_map*) and in the 1D representation. Its size is given by the total number of non-zero transition probabilities for all state-action combinations.

- *nextStateId*: holds in every cell the Id of the "next state" that has associated the *probability* corresponding to the *probability* cell occupying the same position. It matches the size of *probabilty* array.

- *nextCell*: indicates the starting position of groups of consecutive elements form the *probability nextStateId* arrays that correspond to a given state-action pair. Its size is $NS \times NX$ (total number of states times total number of actions).

The rest of the parameters remain the same.

4.2 Sequential implementation

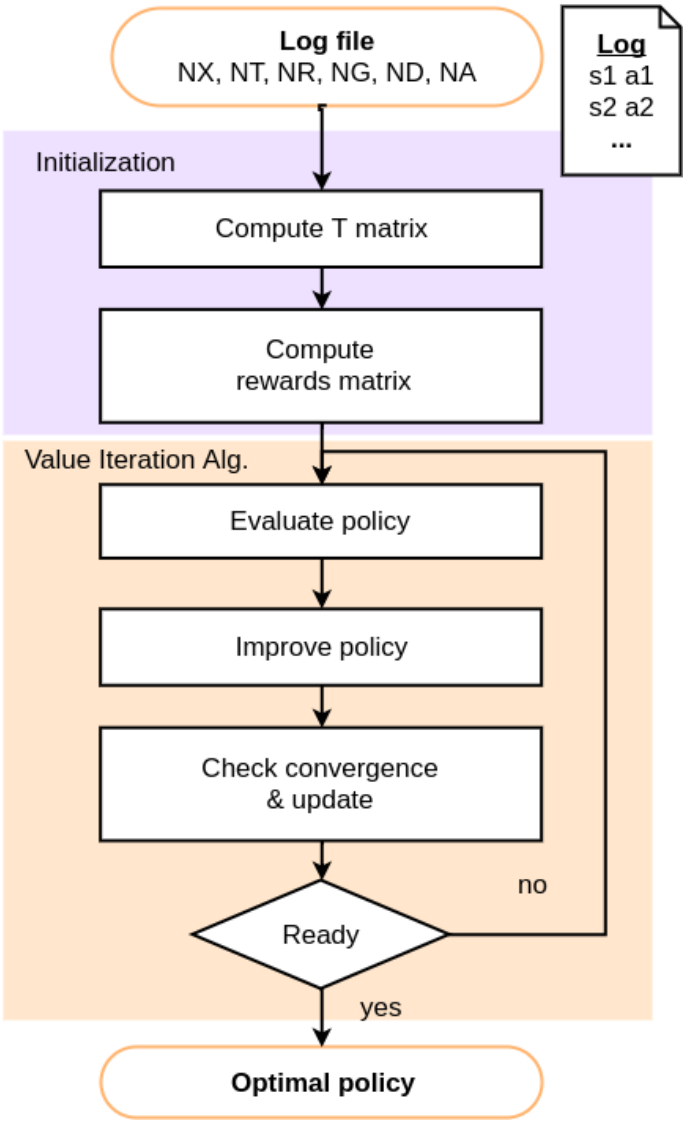

Figure 4.2: Sequential Value Iteration implementation diagram.

The sequential version corresponds to a plain C++ implementation of the algorithm (Fig. 4.2) and serves as a reference for the correctness and speed of the optimized versions of the algorithm.

In figure 2.2 is represented the pseudocode of this sequential implementation. The program receives as input the log file resulting from the simulation, the number of actions (NX), and the discretization parameters (NT, NR, NG, ND and, NA). The output of the algorithm is the (quasi) optimal policy. The *log* is a TSV file containing a list of state-action tuples gathered during a previous simulation. By looking at the block diagram (Fig.4.2) we may see that the implementation has two parts: the initialization (purple) and the algorithm core (orange). In the first part we compute the transition probabilities matrix using the data from the log file, the rewards matrix as described in the *problem definition* section [Eq.2.2] and initialize all the needed variables. The second part is the core of the implementation and it corresponds to the Value Iteration algorithm (Fig.2.2).

In this sequential implementation we identified as the most computationally intensive elements of the algorithm the "Evaluate policy" and the "Improve policy" kernels (the first

two kernels from the lower part of Fig.4.2). Therefore, they will receive our full attention from now on for optimization purposes.

4.3 OpenMP implementation

From this point on we will focus only on the lower part of the sequential implementation for optimization purposes, the one that corresponds to the Value Iteration algorithm implementation.

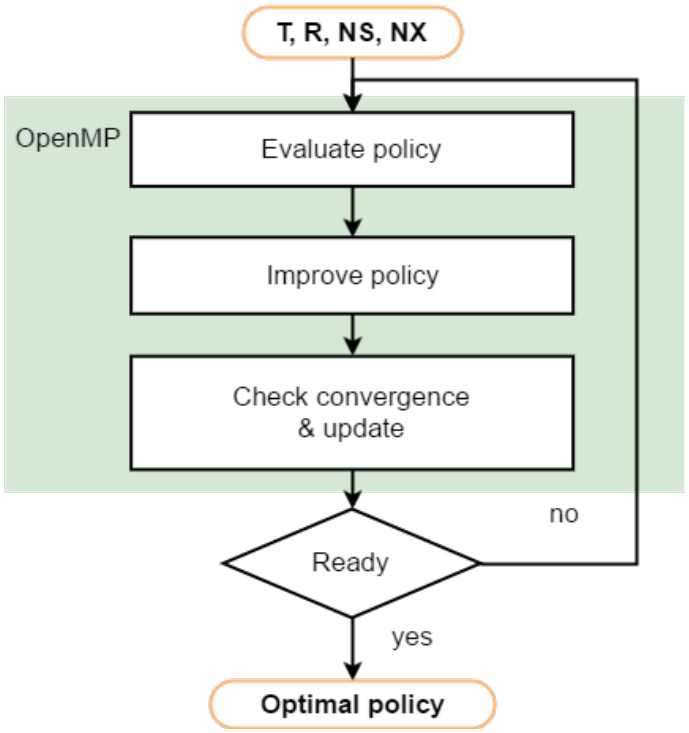

Figure 4.3: OpenMP implementation.

The first optimization that we introduce takes better advantage of the CPU resources by using OpenMP to program parallel threads in the multicore processor for the Evaluate policy and Improve policy kernels (4.3).

The data accessed in the most inner loop of Evaluate policy is irregular (T matrix). Also, our CPU has a big.LITTLE architecture, so we have to use a dynamic scheduling technique to handle the unbalance between the threads. OpenMP provides *dynamic* and *guided* scheduling for this purpose [18, 11]. They both use an internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. The difference between dynamic and guided scheduling is that the first one uses a fixed chunk size, while guided scheduling starts with a large chunk size that is decreased to handle better the load unbalance between the threads. This kind of scheduling involves additional overhead, thus it may not be advantageous for the policy improvement kernel, where the loops are regular.

In Figs. 4.4, 4.5 and 4.6 the OpenMP implementations for the Evaluate policy and the Improve policy kernels using guided scheduling are summarized. All variables have the same meaning as in the definition of the algorithm from section 2.6. There is a new variable, Q, that is a linearised $NS \times NA$ matrix which contains that value of taking an

```
#pragma omp for schedule(guided) collapse(2) \
private(s,a,s2,idx) reduction (+:w)
for (s = 0; s < NS; s++) { // current state
    for (a = 0; a < NX; a++) { // action
        w = 0;
        idx = s * NX + a;
        for (s2 = 0; s2 < NS; s2++) { // next state
            w += T.getProbabilitySAS(s, a, s2) * V[s2];
        }
        Q[idx] = R[idx] + gamma * w;
    }
}
```

Figure 4.4: OpenMP optimization for Policy evaluation using the unordered-map matrix representation for the T matrix.

```
#pragma omp for schedule(guided) firstprivate(gamma) \
private(s,a,s2,idx,cellPos) reduction (+:w)
for (s = 0; s < NS; s++) { // current state
    for (a = 0; a < NX; a++) { // action
        w = 0;
        idx = s * NX + a;
        for (s2 = nextCellPos[idx]; s2 < nextCellPos[idx + 1]; s2++) {
            w += probability[s2] * V[nextStateId[s2]];
        }
        Q[idx] = R[idx] + gamma * w;
    }
}
```

Figure 4.5: OpenMP optimization for Policy evaluation using the 1D array representation for the T matrix.

```
#pragma omp for schedule(guided) private(s,a,idx)
//for every state select the action with the best value
for (s = 0; s < NS; s++) { // states
    for (a = 0; a < NX; a++) { // actions
        idx = s * NX + a;
        if (Q[idx] - Z[s] > EPSILON) {
            Z[s] = Q[idx];
            P[s] = a;
        }
    }
}
```

Figure 4.6: OpenMP optimization for Policy improvement. It is the same for both data structure representations.

action a from a state s. R is also linearised. Z contains the value of the policy from the previous iteration and it. Thus, V in the implementation corresponds to V_k and Z to V_{k-1}. P is the computed policy, π.

4.4 TBB implementation

The Intel Threading Building Blocks (TBB) is a C++ library for task parallelism that provides functions and templates to build applications that run on multicore CPUs. The API is not processor dependent so it should work with both intel and non intel processors (eg. ARM based processors or AMD processors). The resulting applications can be executed on Windows, Linux and Mac OS platforms provided that they include an ISO C++ compiler such as GCC, ICC or Clang.

Among others, this library provides a *tbb::parallel_reduce* and a *tbb::parallel_for* function template. Both perform a parallel run of a loop over a range of iterations. The default partitioner of these function templates recursively splits the range of iterations into chunks, until a minimum threshold size is reached. Each chunk is then run as an independent task and the internal TBB runtime scheduler employs a workstealing technique in order to balance the load of task across all CPU cores. The TBB scheduler uses all available cores by default, unless we specify the number of threads to be used by the task scheduler when calling the *tbb::task_scheduler_init* function.

The OpenMP *#pragma omp for*, used in the previous implementation, and the TBB *tbb::parallel_for* have the same functionality. They differ though in the scheduler implementation. The same applies to *#pragma omp for* with the *reduce* clause and the *tbb::parallel_reduce* function.

In figure 4.7 is resumed the implementation of VI using TBB library. Intel TBB has split the declaration of its classes and functions in different headers. We only need the *tbb::task_scheduler_init*, *tbb::blocked_range*, *tbb::parallel_reduce* and *tbb::parallel_for* in the code so we only need to include their corresponding headers (lines 1 to 5). The functor is an object of an anonymous single-method class that provides an implementation for operator(). We use two functor objects:

- `f` (line 27) - implements "Evaluate policy" and "Improve policy" kernels of the VI algorithm, and it will be called inside *tbb::parallel_for*.

- `n2sum` (line 47) - is an instance of the single method class Norm2 (line 22) that computes the squared norm2 of two arrays of float and it will be called inside *tbb::parallel_reduce* (line 52).

We call the *tbb::parallel_for* loop (line 29) and give it as parameters a range object and the functor. The range object indicates the portion of the total work that will be processed, in our case from 0 to NS, which is the total work. We can see that the functor also receives a *blocked_range* object as parameter. By calling *tbb::parallel_for* the code in the functor is executed (on several threads, each with its own subrange) so that after the call, the policy will have been computed for all the elements in the range [0, NS). The *tbb::parallel_reduce* function call behaves in a similar way to *tbb::parallel_for*, only that the individual results from each subrange are cumulated or reduced in one variable, `norm2.my_sum`. From line 51 to 56 is executed the "Check convergence & update" kernel (Fig. 4.2).

```
#include <tbb/task_scheduler_init.h>
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#include <tbb/parallel_reduce.h>
using namespace tbb;
...
class Norm2 {
    float* my_a;
    float* my_b;
public:
    float my_sum;
    void operator()( const blocked_range<size_t>& r ) {
        float *a = my_a;
        float *b = my_b;
        float sum = 0;
        size_t end = r.end();
        for( size_t i=r.begin(); i!=end; ++i )  {
            sum += pow((a[i]-b[i]),2);
        }
        my_sum = sum;
    }
    Norm2( Norm2& x, split ) : my_a(x.my_a), my_b(x.my_b), my_sum(0) {}
    void join( const Norm2& y ) {my_sum+=y.my_sum;}
    Norm2(float *a, float *b) :  my_a(a), my_b(b), my_sum(0) {}
};
...
auto f = [&](const blocked_range<size_t>& r){
  for (ulong s = r.begin(); s != r.end(); ++s) {
      for (int a = 0; a < NX; a++) {
          float w = 0;
          size_t idx = s * NX + a; // go to the next cell, T/Q/R matrix
          // Policy Evaluation
          for (size_t s2 = nextCellPos[idx]; s2 < nextCellPos[idx + 1]; s2++) {
              w += probability[s2] * V[nextStatePos[s2]];
          }
          Q[idx] = R[idx] + gamma * w;
          // Policy Improvement
          if (Q[idx] - Z[s] > EPSILON) {
              Z[s] = Q[idx];
              P[s] = a;
          }
      }
  }
};
...
//Value  iteration algorithm
Norm2 n2sum(Z,V);
while(notReady) {
    // Evaluate & Improve policy
    parallel_for( blocked_range<size_t>(0,NS), f);
    // Check convergence condition and update
    parallel_reduce( blocked_range<size_t>(0,NS), n2sum);
    norm = sqrt(n2sum.my_sum);
    notReady = norm > g && nrIter < MAX_ITER;
    memcpy(Z, V, sizeof(float) * NS);
    nrIter++;
}
```

Figure 4.7: TBB optimization for VI algorithm.

4.5 OpenCL implementation

The algorithm does not allow parallel execution of the Evaluate policy and the Improve policy kernels because the second one depends on the results of the first one. Therefore, it is necessary to execute the two kernels sequentially. Knowing this, there are two possible approaches:

1. Use the GPU to compute only the Evaluate policy kernel, which is the most computationally expensive part of the algorithm, while leaving the rest to the CPU, as in Fig.4.8.
2. Use the GPU to compute both kernels, using a synchronization barrier between the two kernels that are executed sequentially.

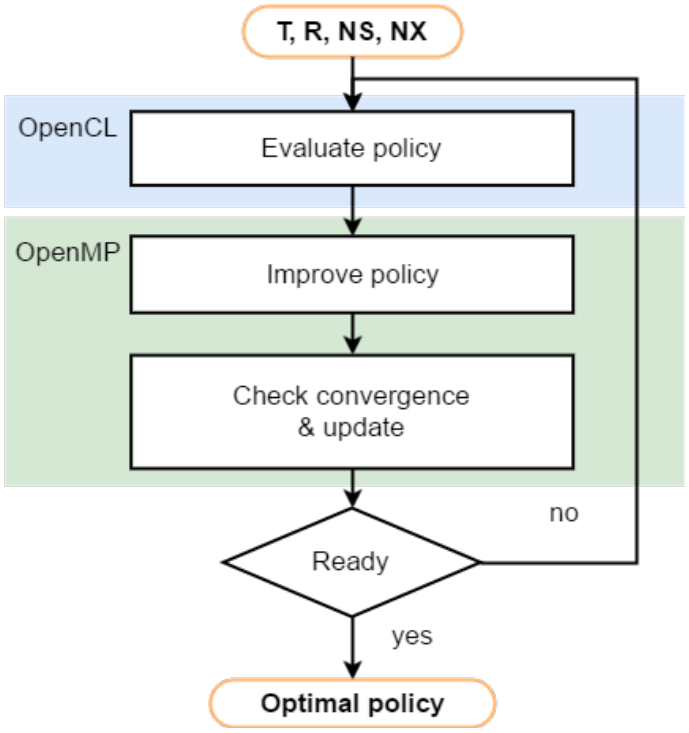

Figure 4.8: OpenCL - OpenMP implementation.

Fig. 4.9 shows our OpenCL kernel implementation for the Evaluate policy kernel. Its input arguments are the T matrix linearised components (probability, nextCell, and nextStateId), the rewards matrix R, the value of the current policy, V, and the number of states and actions (NS and NX). The kernel returns the computed value of taking an action from a given state. The result is stored in Q, in the cell corresponding to the global position given by the work-item executing the kernel code.

It would be ideal to divide the work in identical chunks of work-items for the GPU. Unfortunately, this is not possible with the kind of input data we have (sparse data grouped contiguously by state-action pairs and arranged action-wise). By orderly grouping the T matrix data by the contents of the state-action cells we use far less memory but at the same time induce a major difficulty in dividing the input data (probabilities, R, Q,...) into equally sized work chunks. As we explained previously, this happens because when the robot takes an action from a given state it can end up in a variable number of states. Because of this, the GPU cores will unavoidably have unbalanced workloads to process.

```
__kernel void policyEvaluation(
                __global const  float* probability,
                __global float* V,
                __global float* Q,
                __global const  unsigned long*  nextCellPos,
                __global const  unsigned long*  nextStatePos,
                __global const  float* R,
                unsigned long   NS,
                unsigned long   NX
)
{
  float w = 0.0;
  float gamma = 0.1;
  ulong s2;
  unsigned int myCellPos = get_global_id(0);
  if ( myCellPos < NX * NS) {
    for (s2 = nextCellPos[myCellPos]; s2 < nextCellPos[myCellPos + 1]; s2++) {
      w += probability[s2] * V[nextStatePos[s2]]; //  w += T(s,a,s2) * V[s2];
    }
    Q[myCellPos] = R[myCellPos] + gamma * w ;
  }
  return;
}
```

Figure 4.9: OpenCL kernel for Policy evaluation using the 1D representation of the T matrix.

4.6 Load balancing and scheduling

In this section we present a VI implementation that allows applications static and dynamic load balancing, and to run on CPU and GPU concurrently. For this purpose we use a library that extends the functionality of the Intel TBB library *tbb::parallel_for* by including the possibility the send work to the GPU too. This library offers a number of scheduling policies which implement a *heterogeneous_parallel_for* function [19].

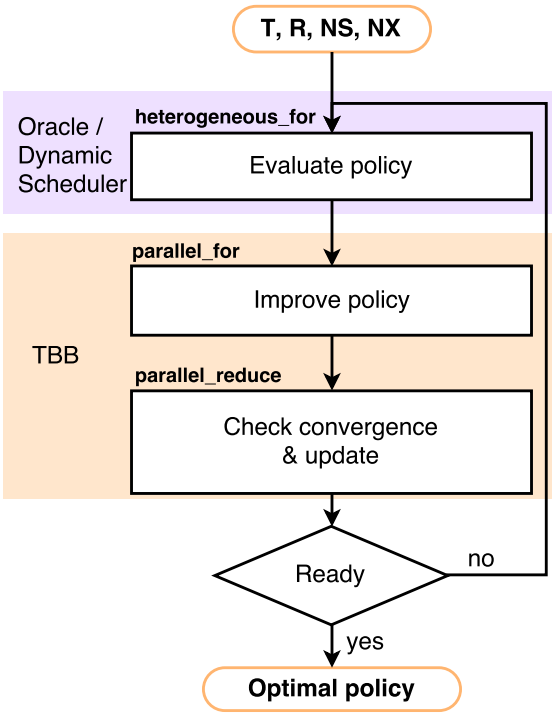

Figure 4.10: VI algorithm implementation using Oracle/Dynamic schedulers and TBB library.

We use only two scheduling policies from this library: the Oracle and the Dynamic schedulers. With diagram 4.10 we illustrate the structure of this VI implementation that uses one of the two Schedulers, either Oracle or Dynamic. The Oracle scheduler gives the possibility to manually divide the workload between the CPU and the GPU by setting a parameter to a number between zero and one to indicate how much of the work goes to the GPU. The remaining part goes to the GPU. In Order to obtain the optimum work balance between the two devices one should try every possibility. For a good approximation of the optimal work division one could try all ratios between zero and one with an increment of 0.1 for instance.

The Dynamic scheduler on the other hand attempts to get to a fair approximation of the optimal work load share between the devices without having to exhaustively try every possibility. For this purpose a chunk size is defined for the GPU and the devices steal work until the execution is done.

The the *heterogeneous_parallel_for* function implemented by these two schedulers works in a similar way to TBB *tbb::parallel_for* function (see section 4.4). It also receives as input the range that will be executed (described by two integer values, begin, and end) and an object instance of a functor class. In our implementation, this object is an instance of the *ValueIterationPlanifier* class (see Fig. 4.11) which implements *OperatorGPU* and *OperatorCPU* methods for executing the "Evaluate policy" kernel on the GPU device and on the CPU respectively. Also, it implements methods for allocating memory, sending work and receiving the results from the GPU device. The *ValueIterationPlanifier* class uses OpenCL (GPU) and TBB (multicore CPU) to execute code on a heterogeneous platform.

```
1  class ValueIterationPlanifier {
2  ...
3  public:
4      /*Allocate memory objects on GPU memory address space*/
5      void AllocateMemoryObjects() {...}
6      /*Send the data to be computed in the GPU*/
7      void sendObjectToGPU( cl_ulong begin, cl_ulong end, cl_event * event) {...}
8      /*Launch the kernel*/
9      void OperatorGPU(cl_ulong begin, cl_ulong end, cl_event * event) {...}
10     /*Receive the data from GPU memory*/
11     void getBackObjectFromGPU(cl_ulong begin, cl_ulong end, cl_event *event)
       {...}
12     /*Serial version of the code */
13     void OperatorCPU(cl_ulong begin, cl_ulong end) {...}
14 };
```

Figure 4.11: ValueIterationPlanifier class.

In figure 4.12 is shown a code sample that illustrates the use of the Oracle and Dynamic schedulers. Here, `f_impPol` and `f_n2sum` are functor objects that implement the Improve policy kernel and the function for computing $|V_k - V_{k-1}|_2$ respectively (norm2 of Z minus V in our C++ implementation). The later is used in the last kernel to check the convergence in the current iteration.

For performance evaluation, we launched our VI algorithm using both schedulers for the input combinations listed next, and measured the execution time and energy consumption by calling startTimeAndEnergy() method before the execution of the VI while loop, and endTimeAndEnergy() and saveResultsForBench() metods afterwards (lines 9, 21, and 22, Fig.4.12).

```
int main(int argc, char **argv) {
    ValueIterationPlanifier vip;
    Params p;
    // Define and init VI and scheduler params
    ...
    Oracle|Dynamic * hs = Oracle|Dynamic::getInstance(&p);
    vip.AllocateMemoryObjects();
    Norm2 f_n2sum(Z,V); // TBB functor object
    hs->startTimeAndEnergy();
    while(notReady) {
        // Evaluate policy (Heterogeneous scheduling: CPU + GPU)
        hs->heterogeneous_parallel_for(0, NS, &vip);
        // Improve policy (TBB - CPU)
        parallel_for( blocked_range<size_t>(0,NS), f_impPol);
        // Check convergence and update   (TBB - CPU)
        parallel_reduce( blocked_range<size_t>(0,NS), f_n2sum);
        norm = sqrt(n2sum.my_sum);
        notReady = norm > g && nrIter < MAX_ITER;
        ...
    } // end of VI
    hs->endTimeAndEnergy();
    hs->saveResultsForBench();
}
```

Figure 4.12: Heterogeneous implementations using Oracle/Dynamic Scheduler for Policy evaluation kernel and TBB for Improve policy, and Check convergence & update.

4.7 Performance comparison of the algorithm implementations

In this section we present the performance of several implementations in terms of execution time and energy consumption on two low power platforms:

1. An Odroid-XU3 board [20] that features a Samsung Exynos-5422 CPU (Cortex-A15 and Cortex-A7 big.LITTLE processor with 2GByte LPDDR3 RAM); an ARM Mali-T628 GPU (600 MHz, OpenCL 1.1), and an energy monitor that contains four separated current sensors to measure the power consumption of Big CPU, Little CPU, GPU and DRAM in real time. Results are discussed in subsection 4.7.1.

2. A 3.30GHz Intel(R) Broadwell quad core CPU, i7-5775C, featuring an Iris Pro integrated GPU with a base frequency of 300 MHz. We rely on the *I*ntel Performance Counter Monitor V2.10 (PCM) library [21, 22] to access the hardware counters. In particular they allow us to measure at runtime the energy consumption (in Joules) on the CPU, GPU and Uncore components for a given application. In Intel terminology, the Uncore is the part of the processor that contains the integrated memory controller and the Intel QuickPath Interconnect to the other processors and the I/O hub. Overall, the following metrics are supported by PCM:

    - Core metrics: instructions retired, elapsed core clock ticks, core frequency including Intel Turbo boost technology, L2 cache hits and misses, L3 cache misses and hits.
    - Uncore metrics: read bytes from memory controller(s), bytes written to memory controller(s), data traffic transferred by the Intel QuickPath Interconnect links.

To be more precise, we will use only the second metric that is equivalent to the memory energy measurement available for the Odroid platform. Our results are shown in subsection 4.7.2).

The implementations were evaluated for four different input sizes: `NZ40445_SD243`, `NZ61012_SD576`, `NZ84565_SD1080`, and `NZ143771_SD1920`, each defined by the number of non-zero values in the T matrix (NZ) and the total number of states (SD). We obtain the input size by adjusting the state discretization granularity (i.e., set the value for NT, ND, NR, NG and NA) so that it increases in exponents of two (approximately). The number of NZ values of the T matrix is derived from the state discretization. We have observed that the NZ varies almost linearly with the SD number.

4.7.1. Odroid platform results

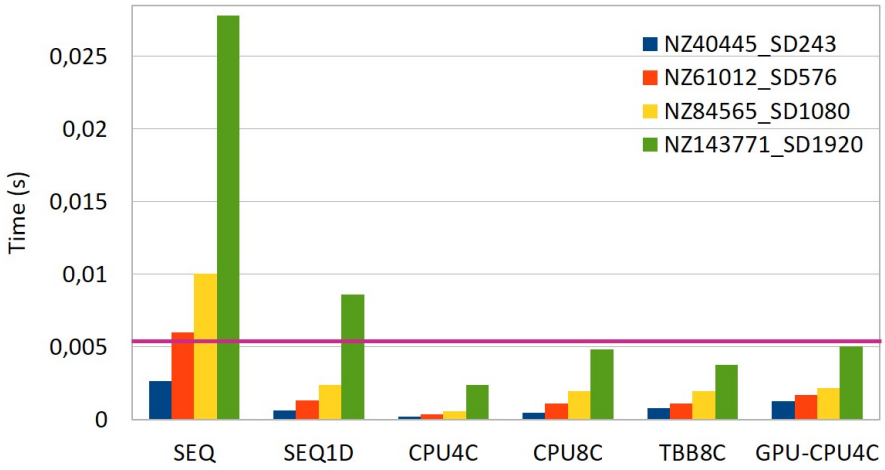

Figure 4.13: Execution time per iteration in Odroid.

Fig. 4.13 shows the average execution time per iteration of the different implementations evaluated in the Odroid platform. The first two implementations are named SEQ and SEQ1D, and correspond to sequential implementations of the VI algorithm, which are used as baseline. They are different from each other because of the T matrix representation: `unordered_map` 2D matrix for SEQ, and one 1D array for SEQ1D. The third one is a OpenMP implementation that uses the same T matrix representation as SEQ1D. This one is tested for two configurations: CPU4C - which is executed only on the Big quad core processor, and CPU8C (big.LITTLE) - that runs on the two quad core processors. We named TBB the fourth implementation which uses the Intel TBB library to distribute the VI algorithm execution among the two available cluster of processors. The TBB implementation is comparable to the CPU8C because they both use the full CPU capacity (four cores from the Cortex-A15 CPU and four cores from the Cortex-A7 CPU). Finally, we have the GPU-CPU4C implementation, that processes the Evaluate policy algorithm kernel on the GPU while the Improve policy and Convergence check & update kernels are computed using the corresponding CPU4C implementation. This one also uses the one dimensional input representation.

From the figure we may observe that the execution time increments abruptly when the input is increased. This behaviour is explained by the fact that the VI algorithm has a quadratic complexity. The magenta line (above 0.005 s) from the same figure marks the maximum admissible execution time per iteration in order to still have real time (RT)

execution. This time is computed assuming that the algorithm would converge in approximately 40 iterations or less and that an acceptable action-perception loop in a typical mobile robot is of about 200 milliseconds. We chose 40 because it is more than the double of the number of iterations needed to converge in tests that we made. With this restriction, all implementations except for the sequential ones (SEQ and SEQ1D) are appropriate for RT. In the next figure we can appreciate the SEQ, CPU4C, CPU8C, TBB8C and GPU-CPU4C relative speed-up with respect to the SEQ1D implementation.

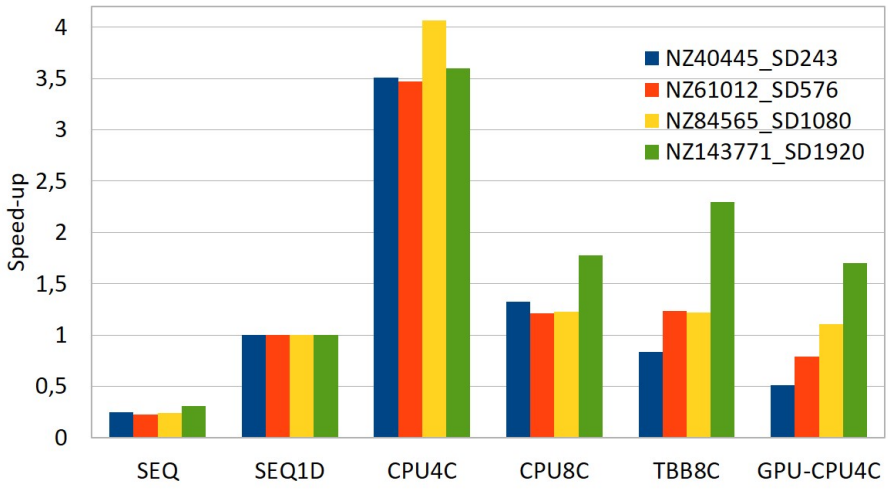

Figure 4.14: Speed-up relative to the SEQ1 implementation in Odroid.

We have analysed the average energy consumption per iteration of the SEQ1D, CPU4C, CPU8C, TBB8C and GPU-CPU4C executions, measured on the CPUs, GPU, and memory (Mem) (Fig. 4.15) components. Our experiments indicate that the main energy consumption comes from the CPUs, while the memory and the GPU have very little impact on the total. As we expected, the implementation that uses less the CPU, that is, the GPU-CPU4C implementation, is also the one that uses the least energy, while the sequential implementations tend to consume most because they also take longer to converge. As seen in the figure, The CPU8C implementation is not as energy efficient as CPU4C even though CPU8C uses less energy per time unit than CPU4C; the reason is the extended execution time of the CPU8C, induced by a limitation of the OMP guided scheduling technique in load balancing between threads that are executed on asymmetric processors (A15 and A7 cores in this case). Similarly, the degradation of performance of the GPU-CPU4C executions when compared to the CPU4C ones is due to GPU/CPU load imbalance in the current GPU-CPU4C implementation. Interestingly, for the bigger inputs (`NZ84565_SD1080` and `NZ143771_SD1920`) we see the heterogeneous GPU-CPU4C implementation is the most energy efficient, because now there is enough computation offloaded to the most energetically efficient GPU, computation that is not performed in the more power hungry A15 cores (see the reduction in the CPU energy consumption bar in the figure).

We have used ANalysis Of VAriance (ANOVA) framework to determine whether there are any statistically significant differences both in the execution time and energy consumption measurements of the different VI implementations, including a Tukey's post-hoc test to decide the ordering of elements when differences that are significant [23] are detected. We assume that the measurements are independent, something that in our experiments in Odroid the platform must be taken carefully [1].

[1]Our implementations execute too fast for the energy measurement sensor sampling capability in the Odroid. Our solution was to take one measure of the energy and execution time when repeating each VI

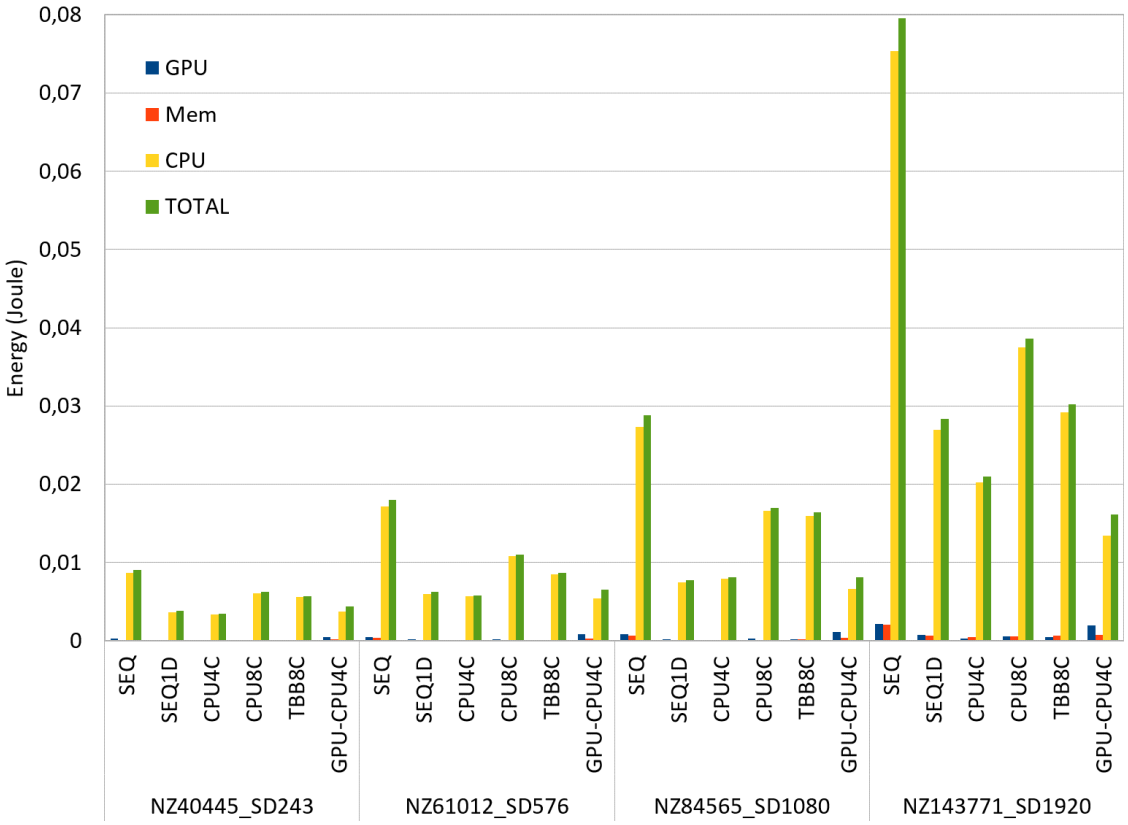

Figure 4.15: Energy consumption overview in Odroid.

The ANOVA test corresponding to the averages shown in Fig. 4.13 indicates that for the two smallest input-sizes, the computation time is increasing statistically with significance for the algorithms in this order: CPU4C, CPU8C, TBB8C, SEQ1D, GPU-CPU4, and SEQ. For the two largest input-sizes, the order is: CPU4C, TBB8C, CPU8C, GPU-CPU4, SEQ1D, and SEQ. Additionally, the computation time required to execute any of the implementations increases in the same way with the input size (the smallest computation time corresponds to the smallest size and so on), with statistically meaningful differences between the test groups. Thus ANOVA confirms the statistical differences observed in the graphs. The ANOVA test results for the total energy confirm that for the smallest input size the algorithm of lowest consumption is CPU4C, then SEQ1D, GPU-CPU4C and CPU8C, which are indistinguishable from each other, being the one of greatest consumption SEQ. On the other hand, the total energy for the two largest input sizes increases with statistically significant differences in this order: GPU-CPU4C, CPU4C, SEQ1D, TBB8C, CPU8C, and SEQ. From the point of view of the algorithm, for all of implementations except GPU-CPU4C it is satisfied that, the smaller the input sizes the lower the consumption, with statistically significant differences in all cases. In the case of memory energy for all implementations, it is true that the smaller the input size, the lower the consumption, with statistically significant differences in all cases.

From the presented experiments we have learnt that the CPU4C implementation is always faster than the heterogeneous GPU-CPU4C one, whereas the latter is equal or more energy efficient (only for the larger input sizes) than the former. Therefore, we recommend the GPU-CPU4C implementation for navigation problems with large input sizes for which RT constraint could be guaranteed, or even for problems with a very low consumption restriction and a permissive enough RT constraint.

algorithm implementation 100 times. The graphs showed here use the average execution time and energy per iteration of twenty such measurements.

4.7.2. Broadwell platform results

Fig. 4.16 shows the average execution time per iteration of the different implementations for the Intel Broadwell platform. The sequential SEQ and SEQ1D versions are measured again and used as baseline. We also measure a OpenMP implementation that exploits the four cores available in this system (CPU4C). Next, we measure the TBB implementation (TBB4C) and finally, we have the heterogeneous GPU-CPU4C implementation, that processes the Evaluate policy algorithm kernel on the GPU while the Improve policy and Convergence check & update kernels are computed using the corresponding CPU4C version. Similarly to the previous section, we also evaluate the speed-up (Fig. 4.17) and energy consumption (Fig. 4.18) of our implementations.

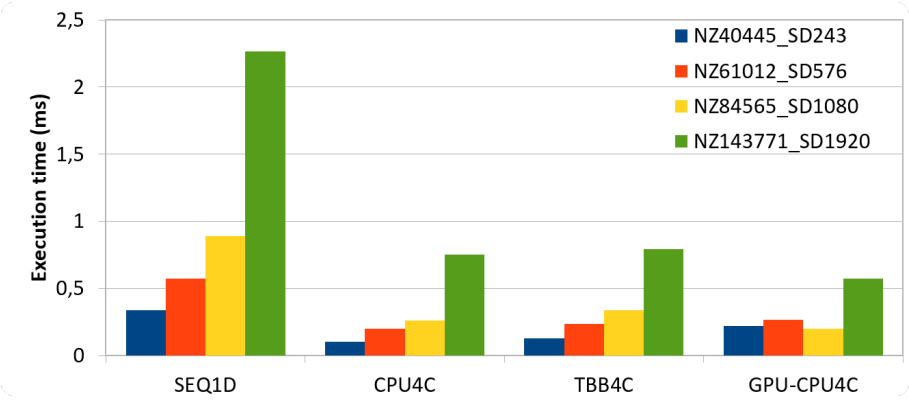

Figure 4.16: Execution time per iteration in Broadwell.

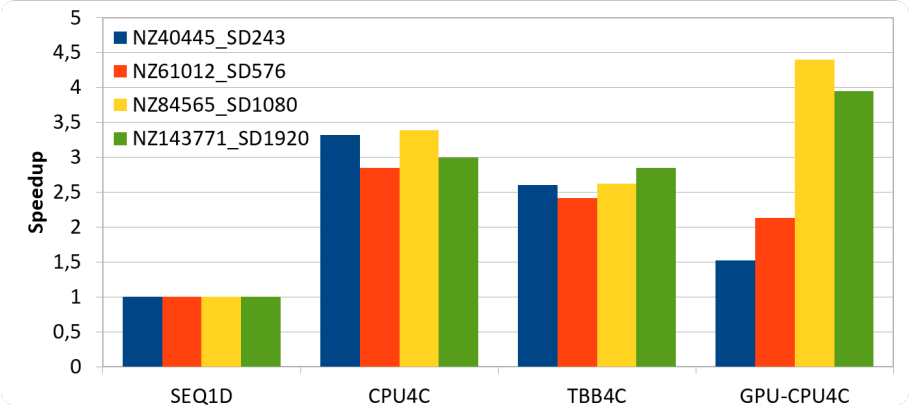

Figure 4.17: Speed-up relative to the SEQ1 implementation in Broadwell.

In Fig. 4.16 is represented the average execution time per iteration of the already mentioned implementations. We notice that the computation time increases with the input size for all implementations except for CPU-4C. The CPU-4C time measurements indicate that the smallest execution time per iteration corresponds to the second largest input size (NZ84565_SD1080), although the rest of the measurements do follow the rule: the fastest result corresponds to the smallest input, the second fastest to the second smallest input, etc.

The ANOVA results confirm that the two smallest inputs (Z40445-SD243 and NZ61012-SD576) lead to increasing computation times according to the used algorithm, following this order: CPU4C, TBB4C, GPU-CPU4C,and SEQ1D, while the two largest inputs (NZ84565_SD1080 and NZ143771_SD1920) lead to increasing computation times according to the algorithm used, following this order: GPU-CPU4C, CPU4C, TBB, SEQ1D. The same conclusion can be confirmed with ease by inspecting the speed-up graph (Fig. 4.17).

As expected, the implementations that use multi-core parallelism optimizations (i.e, CPU4C and TBB4C) perform better for smaller problem sizes, while the use of the GPU accelerated implementation (GPU-CPU4C) makes sense only for inputs that are large enough to fully occupy the GPU resources.

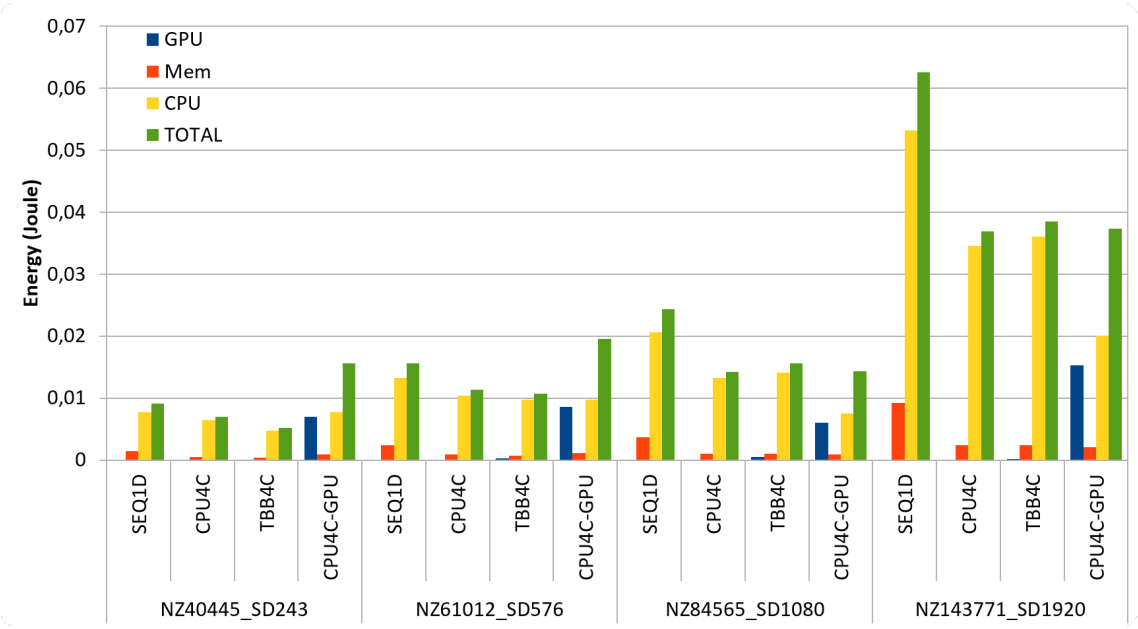

Figure 4.18: Energy consumption overview in Broadwell.

Next we will discuss the energy performance results of our four implementations on the Intel platform. Fig. 4.18 shows the average energy consumption per iteration of our VI implementations. In green we indicate the total energy that is composed by the GPU energy (blue), memory energy (red), and CPU energy (yellow). For total energy the two smallest input sizes give increasing energy per iteration in the following increasing order: TBB4C, CPU4C, GPU-CPU4C, SEQ1D. The increasing consumption order for the third and fourth input sizes (NZ84565_SD1080) is: GPU-CPU4C, CPU4C, TBB4C, SEQ1D.

We note that the GPU-CPU4C is the only one that leaves records of GPU energy consumption, which is totally normal, as it is the only implementation from the four that actually uses the GPU. The rest give zero and near-zero values for GPU energy consumption measurements.

Finally, we include a couple of figures that illustrate the time (Fig. 4.19 and Fig. 4.20) and total energy performance on the Intel Broadwell versus the Odroid platforms. From the first graph it is obvious that the Intel platform is far better suited than Odroid for real time execution (about 3x faster for the slowest implementation tested - SEQ1D). Basically all the parallel tested implementations respect the real time constraint indicated by the magenta line. On the other hand, we see that Odroid consumes less than half of than Intel for SEQ1D and the rule is approximately maintained for all the equivalent implementations (CPU4C, and GPU-CPU4C). Finally, Fig. 4.21 shows the energy efficiency comparison on both platforms: we report the ratio of the most energy efficient Broadwell implementation vs. the most energy efficient Odroid implementation. The energy improvement ranges from 51% to 128%.

Exploring heterogeneous schedulers

As we have seen in the previous section, it could be worthy to explore heterogeneous solutions in the Intel Broadwell platform to try to reduce energy consumption. For it, we try

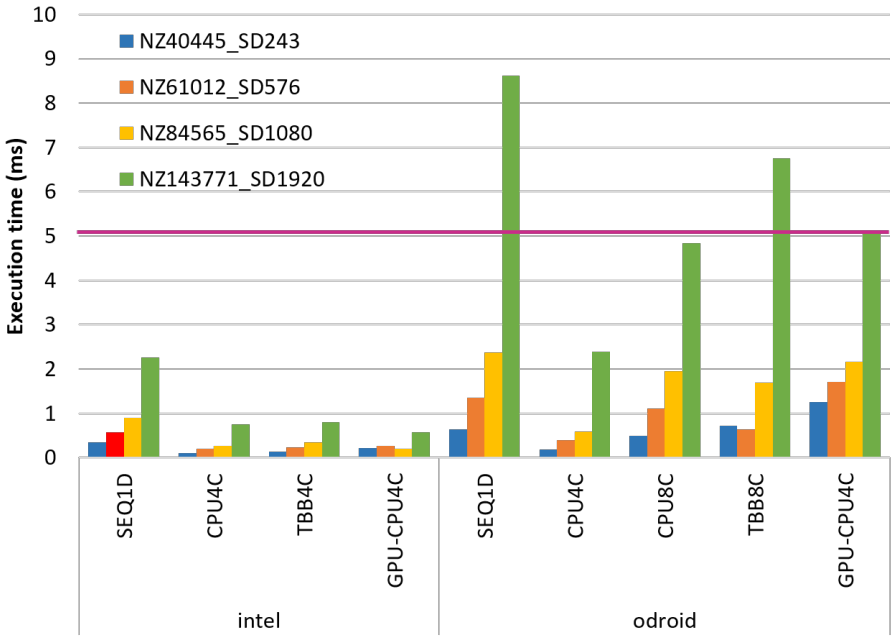

Figure 4.19: Execution time overview: Broadwell versus Odroid.

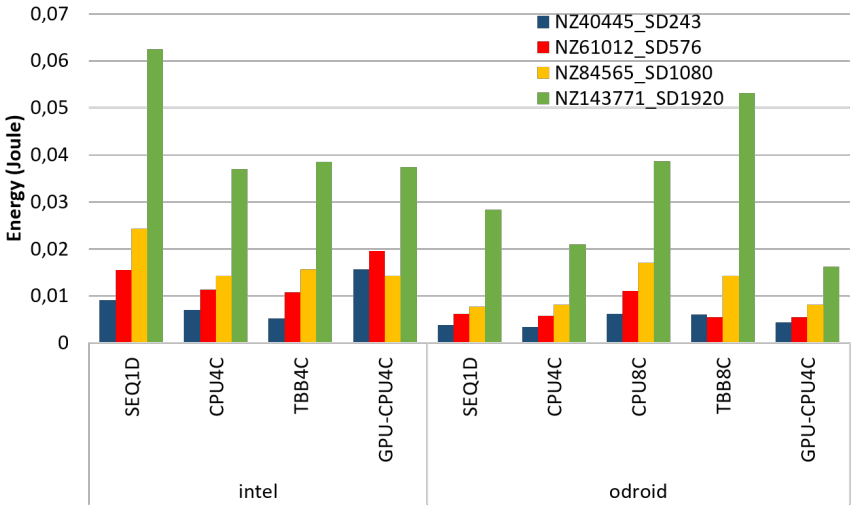

Figure 4.20: Total energy consumption overview: Broadwell versus Odroid.

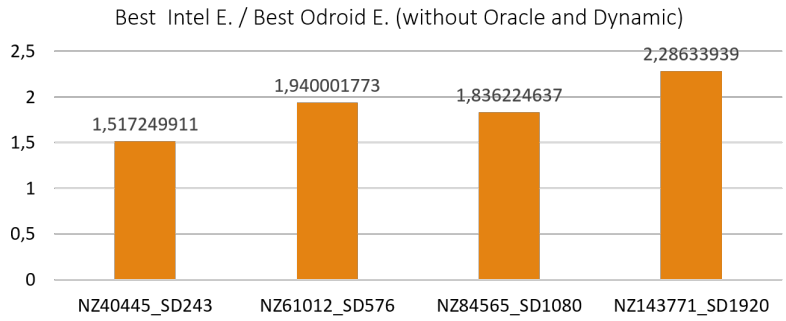

Figure 4.21: Total energy consumption overview: reporting the energy of the most energy efficient Broadwell implementation to the most energy efficient Odroid implementation.

two heterogenous schedulers that are invoked through a parallel_for template. Through the template we invoke two different scheduling strategies: Oracle and Dynamic. Both of them take care of the distribution of the parallel iterations among the CPU cores and the GPU. In this study we focus on the distribution of the iterations of the policy evaluation parallel_for kernel. Let's recall that in the previous study we had either the CPU or the GPU devices doing the computation of this kernel, but not both simultaneously, as in this case. Our goal is to try to load balance the workload between both devices and avoid to keep one of them sitting idle while wasting energy without performing useful work.

The goal of our analysis is to find the workload distribution between devices (the CPU multicore and the GPU) that gets the smallest energy consumption while fulfilling a given RT constraint. In any case we evaluate two new implementations that are identified by the name of their corresponding invoked schedulers, Oracle and Dynamic [19]. The Oracle (Static) scheduler receives as a parameter a number in range [0.0, 1.0] representing the ratio of iterations that should be computed on the GPU. Then the scheduler splits the iteration space just once at the beginning of the execution in two blocks: a block with the number of iterations corresponding to that range is offloaded to the GPU and another block with the remaining iterations will be sent to the multicore CPU. On the other hand, the Dynamic scheduler gests a positive integer that sets the size of the blocks of iterations that will be offloaded dynamically to the GPU, while simultaneously smaller blocks of iterations are sent to the multicore CPU, in a similar way to the OMP dynamic scheduling strategy.

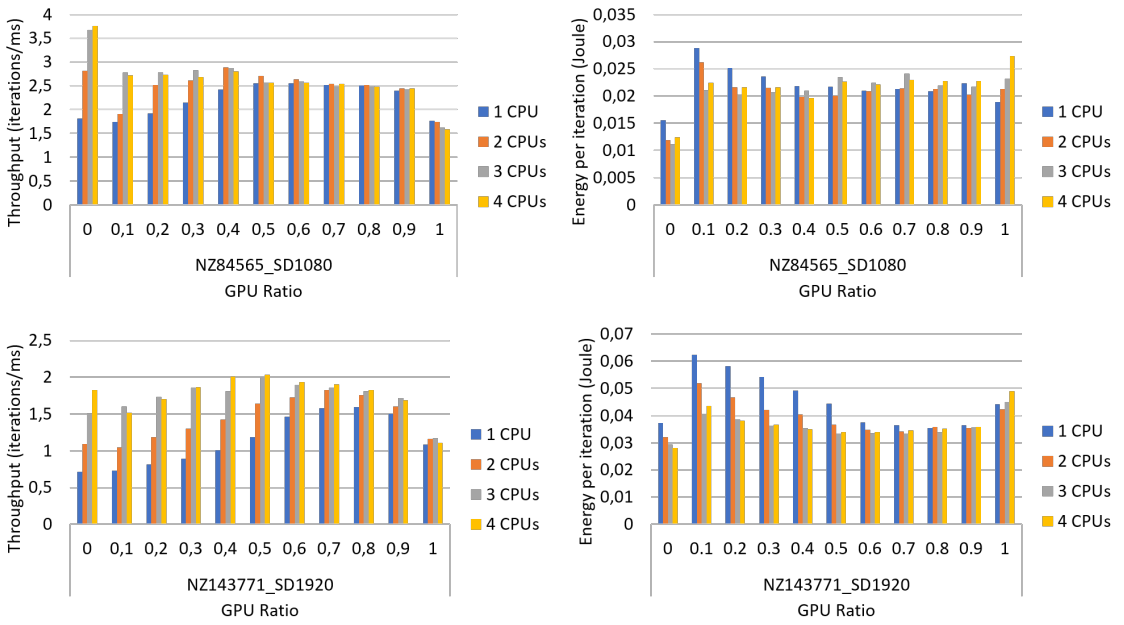

Figure 4.22: Oracle scheduler implementation performance evaluation for throughput (left column graphs) and energy (right column graphs). X-ases represent the ratio of iterations offloaded to the GPU.

In order to evaluate and compare the performance of these last two implementations we have drawn four types of graphs from the measurements that were taken on the Intel platform:

- The first one represents the throughput (iterations/ms), while the second represents the energy required per iteration when we vary the GPU load (for Oracle) or the GPU block size (for Dynamic) as we can see in Figs. 4.22 and 4.23. More specifically, the left side graphs of these figures represent the average throughput and

the right hand graphs show the energy consumption per iteration. For the Oracle VI implementation we vary the GPU load from 0% to 100% with an increment of 10% (Fig. 4.22). However, for the Dynamic implementation we use the GPU block size instead of the GPU ratio. In the mentioned graphs the block size increases from 128 to 65536 in exponents of two (Fig. 4.23). We note that in Fig. 4.23 the largest block size for `NZ84565_SD1080` input is 8192 due to input size constraints. Every graph represents the evolution of throughput or energy for one thread (only-GPU), two threads (GPU+1CPU), three threads (GPU+2CPUs) and four threads (GPU+3CPUs). We drew the throughput and energy graphs for two largest input sizes: `NZ84565_SD1080` and `NZ143771_SD1920`.

- The third type of graph illustrates the maximum throughput on X axis with the corresponding energy consumption per iteration on the Y axis for three different input sizes as in Figs. 4.24 and 4.25. For every input size we drew a different colour line. Every point on the line marks the maximum throughput (X axis), corresponding energy (Y axis), and how many threads were used for the measurement and the GPU ratio (or block size). The number of threads (NT) and the GPU ratio or block (NG) are indicated next to each point marker as NT/NG.

- The last kind of graph (Fig. 4.26) is similar to the third one, only that it shows together the results of the Oracle and Dynamic scheduler VI implementations for easier comparison. Here we depict only the two largest input measurements for better clearness.

By inspecting Fig. 4.22 we notice that, for each input size and number of threads, the Oracle scheduler finds the optimal throughput on a different workload distribution between the GPU and the CPUs. However, the minimum energy per iteration is always obtained for the 0% results (only CPU execution). Similarly, by studying Fig. 4.23 we see that for each input size and number of threads, the Dynamic scheduler finds the optimal throughput on a different block size. Moreover, the minimum energy consumption also depends on the number of threads, and it is not correlated with the block size for which we obtain the maximum throughput.

By analyzing Fig. 4.24 we draw the conclusion that there is a clear advantage in distributing the computation between the CPU cores and the GPU for parallel execution when we have a large problem size. On the contrary, for smaller problem sizes we obtain better throughput and lower energy consumption when using only the CPU cores. For instance, see Oracle results in Fig. 4.24 for the NZ61012_SD576 input. In this case, we obtain the best throughput and minimum energy when employing 3 threads and 0% of iterations are offloaded to the GPU (blue dotted line, mark 3/0). See also that for the largest problem size we obtain the best throughput when employing 4 threads and share the load equally between the CPU and the GPU (the orange dotted line, 4/0.5 mark), and the lowest energy for 3 threads and 0.5 load for the GPU (3/0.5 mark). Fig. 4.25 shows us that for a large input, we can improve performance and reduce energy consumption when we increase the number of threads and carefully select the optimal block size (see yellow solid line). However for smaller inputs (see blue and orange lines) we can improve throughput but not energy consumption when we increase the number of threads.

By looking at Fig. 4.26 we notice that the Oracle and the Dynamic VI implementations obtain closed results for throughput - energy consumption for the largest problem sizes (NZ143771_SD1920) while the Dynamic VI implementation clearly outperforms the Oracle VI implementation for the second largest input (NZ84665_1080). In any case, it finds a solution with a 14.24% higher throughput for the largest input size and and a 10.1% lower energy than its Oracle counterpart (3/4096 - Dynamic versus 4/0 - Oracle). Although the

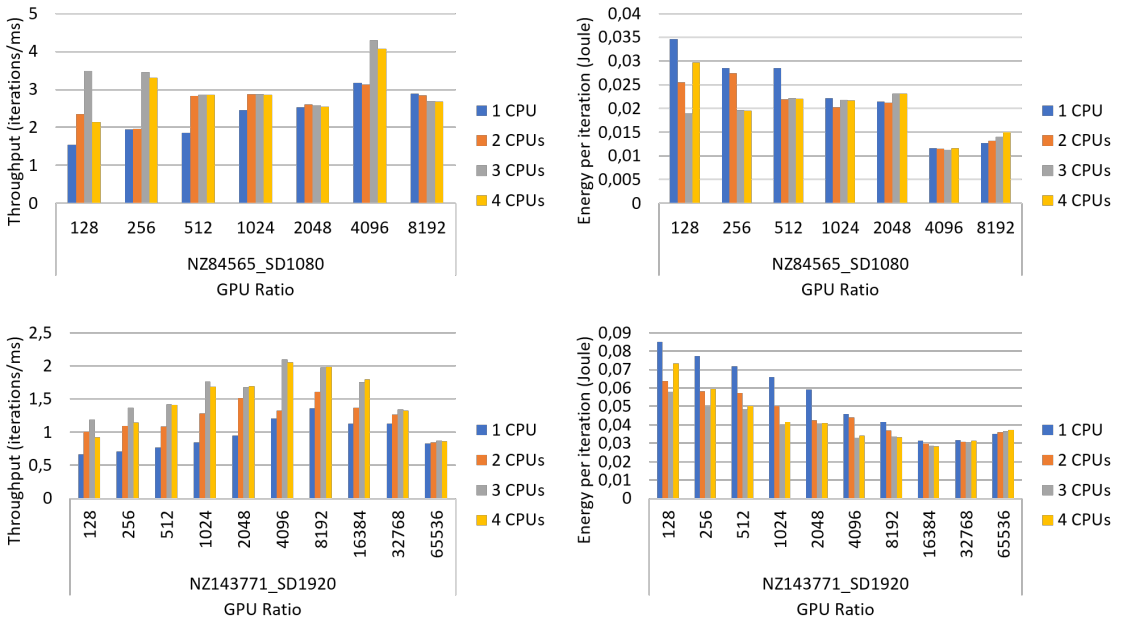

Figure 4.23: Dynamic scheduler implementation performance evaluation for throughput (left column graphs) and energy (right column graphs). X-ases represent the size of the blocks of iterations dynamically offloaded to the GPU.

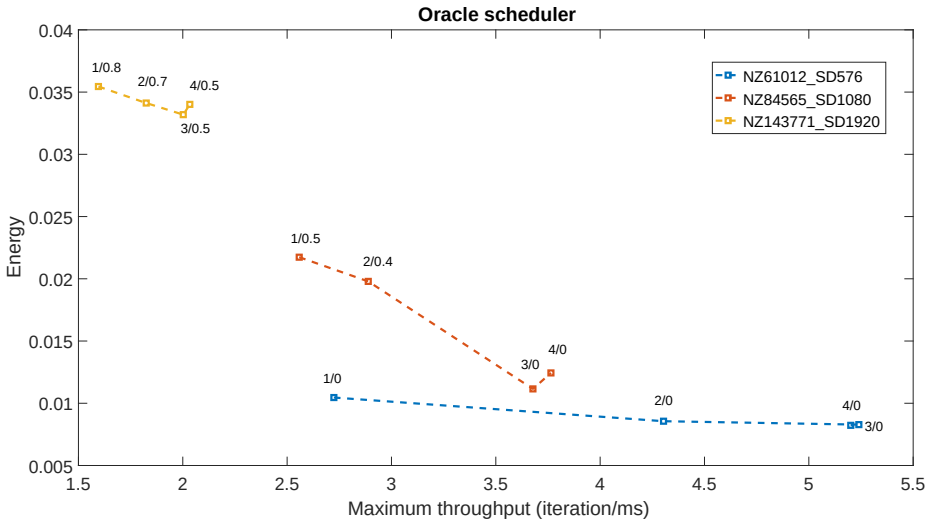

Figure 4.24: Maximum throughput scenarios for Oracle scheduler.

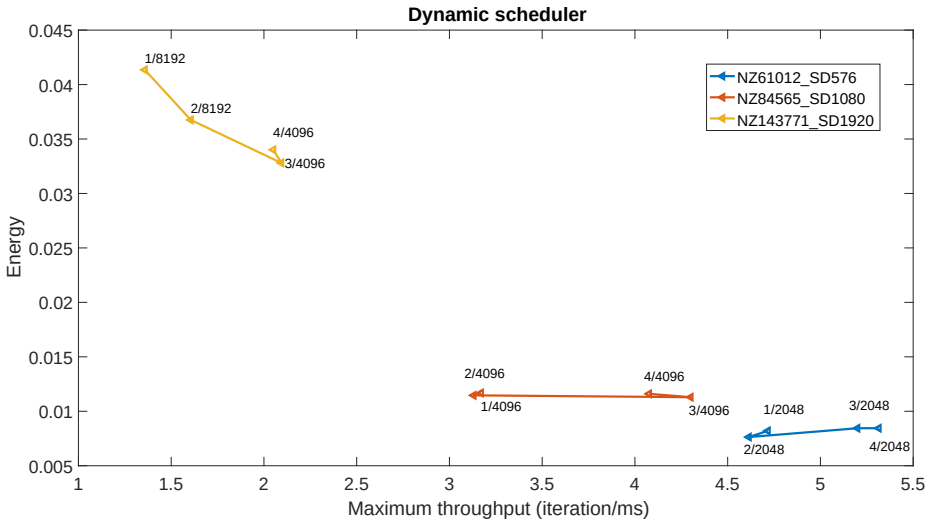

Figure 4.25: Maximum throughput scenarios for Dynamic scheduler.

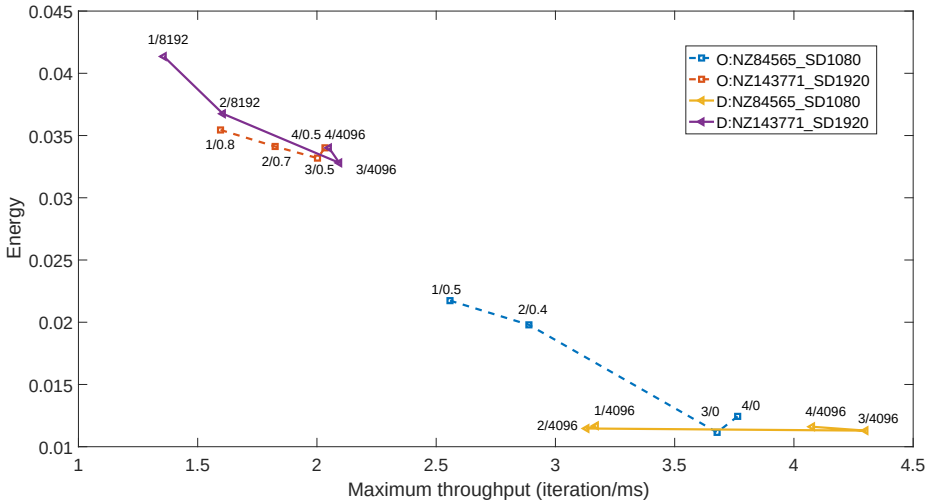

Figure 4.26: Maximum throughput scenarios for Oracle (dashed lines) and Dynamic (solid lines) scheduler.

Oracle implementation obtains approximately the same energy performance as the best solution (3/4096 - Dynamic) for the 3/0 configuration, but at the cost of a 16.91% less throughput.

Finally, now Fig. 4.27 shows the energy efficiency comparison on both platforms considering the new heterogeneous schedulers previously studied. The figure report the ratio of the most energy efficient Broadwell implementation vs. the most energy efficient Odroid implementation. We should note that the heterogenous scheduler implementations bring a slight improvement in the energy performance on the Intel platform side of the equation, because now the energy improvement ranges from 9% to 103%.

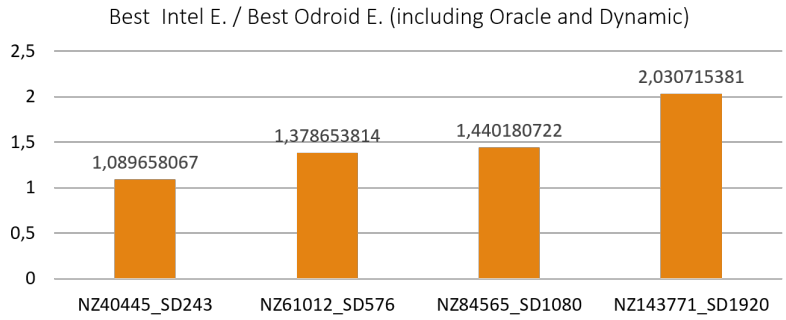

Figure 4.27: Total energy consumption overview: reporting the energy of the most energy efficient Broadwell implementation to the most energy efficient Odroid implementation.

In conclusion, by using either implementation we obtain a reasonable approximation of the optimal load balancing point between the CPU cores and the GPU. Both have the disadvantage that one has to try different load balancing configurations by hand in order to find the better ones in terms of speed and energy consumption. Even so, this is not a major drawback because we assume that the environment will always be the same, our application is the only one using the available resources and the problem is always the same - to (re)compute the optimal policy. Therefore it is reasonable to compute the optimal scheduling solution offline as we only have to do it once.

4.8 Correctness of the Value Iteration implementations

We have observed that the sequentially obtained policy and the policies obtained from optimized implementations occasionally differ in a few actions. This has led us to consider that maybe the OpenMP/OpenCL implementations might induce some inaccuracies. After carefully investigating the problem we realized that this behaviour is due to the lack of commutativity and associativity of sum operation with float numbers in C++. This affects the result of the Q matrix (representing the value expected for taking a certain action from a given state) which in its turn affects the resulting policy, P. The differences between the values of Q computed sequentially and those computed with OpenMP threads or OpenCL kernels are in the order of 1e-10 to 1e-32.

We have taken two measures in order to diminish this unwanted effect:

- Use a precision error when comparing floats, EPSILON = 1e-16,
- Set rewards for the actions that are much bigger (e.g.:1000), so that the resulting policy actions are not easily discriminated by very small differences in the values of Q.

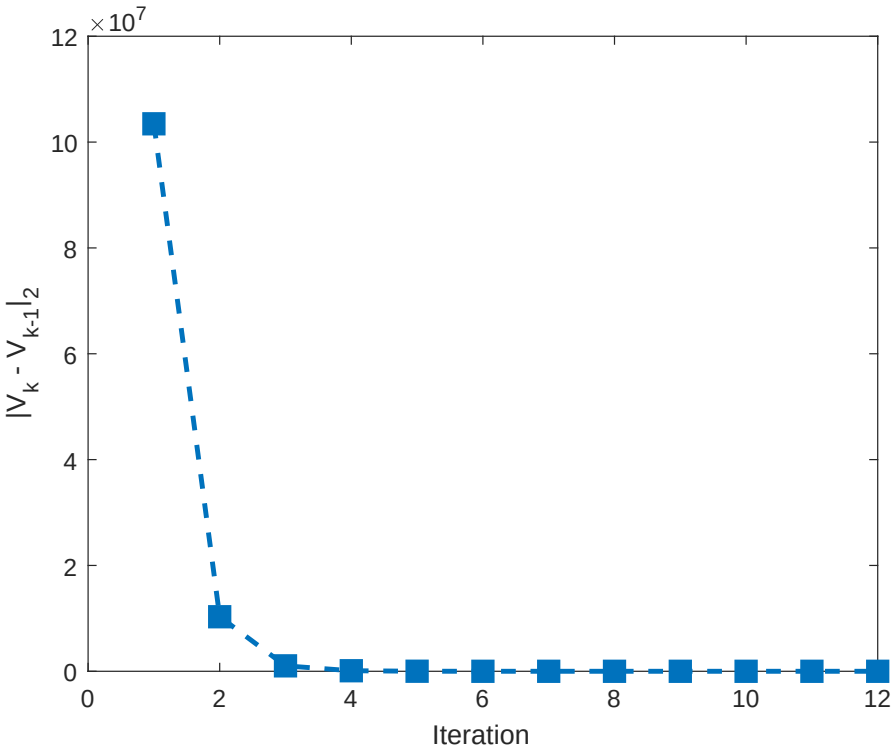

Figure 4.28: VI algorithm convergence.

Still another solution is to use higher precision for Q (double instead of float values), but this is not possible for the OpenCL implementation because the current drivers does not allow double precision.

Having taken these measures, the Q matrix values still differ slightly; for different executions of the algorithm, the resulting policies may differ only in very few values, and the number of iterations needed to converge can differ in a maximum of one iteration.

Another aspect that should be looked into with respect to the correctness of the implementation is whether the algorithm converges or not. By plotting the norm two of the difference between the policy value from the current iteration and the policy value from the previous iteration (Fig. 4.28) we can notice that the algorithm indeed converges. In other words, it reaches a point when the policy can no longer be improved in a finite number of iterations, i.e., $V_k - V_{k-1} \simeq 0$.

CHAPTER 5
Conclusions

As we stated in the introduction, one of our main objectives has been to propose one or more implementations for the VI algorithm that make a good compromise between computing the optimal navigation decisions for CRUMB in real time and reducing the energy consumption. In the previous two sections [Sec. 4.2 and 4.7] we have proposed four implementations of the Value Iteration algorithm and measured their performance on two heterogeneous platforms. Our experimental results for the Odroid platform show that the CPU4C (4 CPU big processors) implementation is the best in terms of execution time, while the heterogeneous, GPU-CPU4C (GPU+4 CPU big cores) implementation has the smallest energy consumption, being pure sequential programming the only version unsuitable for real-time. On the other hand, on the Intel platform all four implementations are suitable for real time, while in terms of energy efficiency, and for large inputs again GPU-CPU4C (GPU+4 CPU cores) tends to require the minimum energy.

We define the most appropriate implementation for our use-case scenario as the one that consumes the least energy and respects the real time restriction. From a practical point of view, the GPU-CPU4C implementation is the recommended solution because it is the cheapest in terms of energy use and it computes the optimal policy faster than the real time constraint for Odroid and Intel Broadwell when considering real-word large inputs.

We manage to reduce the total energy consumption on the Intel platform by enhancing the GPU-CPU4C implementation by invoking heterogeneous schedulers that consider load balancing between the devices (GPU and CPU cores). It is important to mention that even though all the implementations perfectly comply with the real time constraint when they are executed on the Intel Broadwell platform and they always execute faster on Intel than on Odroid for all algorithms and input sizes, we obtain better energy efficiency on Odroid. The most energy efficient implementation for Odroid consumes approximately two times less energy than the best implementation for Intel platform in all cases (see figures 4.21 and 4.21).

The other key objective of this project has been to characterize the robot model and scenarios that can be solved in a reasonable amount of time given the extent of resources available writing this thesis. In Chapter 3 are presented in detail the robot model and how we defined the simulation scenarios for our use-case. The resulting MDP model, given our time restrictions, is not as accurate as we would have liked in order to be a good enough approximation for the real robot. A possible direction of improvement is to dedicate more time and computing resources to simulating the robot. This would allow us to use a finer discretization of the state, thus a more accurate model.

In this work we have proposed a number optimizations for improved execution time and power use that take advantage of the Odroid-XU3 and Intel Broadwell heteroge-

neous platform resources, and which can be easily ported on different heterogeneous platforms. We should stress that the CRUMB robot indoor navigation use-case is just an example of how one can model a robotic system using MDPs and optimize a decision making algorithm for real time and energy consumption restrictions. In principle, we could extrapolate and apply the same process to model any other robot navigation scenario by defining its actions, continuous and discrete states, and then computing the transition matrix using simulation or maybe a real robot. The proposed VI algorithm implementations for decision making should be suitable to compute the optimal policy for any such robot (that has been modelled as described in the previous chapters) by simply substituting the transition matrix, actions, and the discretization parameters with the ones corresponding to the new problem.

APPENDIX A
Development Environment

In this Annex we briefly present the software tools used through the development of this thesis.

A.1 CLion C++ IDE

We start with the development environment, CLion, a JetBrains cross-platform IDE for C and C++ that supports GCC/Clang, MinGW Cygwing toolchains, CMake, git, smart code auto-completion for STD libraries (this function was really helpful for me as newcomer to the C++ 14 Standard), and many other features. We chose it because it is a great IDE for managing large programming C++ projects, it has an amazing interface, and it is available free of charge for students.

In order to make the project compiler-independent and operating system-independent we used CMake 2.6 for managing the building process because we had to work with four different platforms through the development and testing process for the VI algorithm that had installed Ubuntu 14.04 or Windows 7/10.

We used gcc 6.2 (MinGW gcc for Windows) both on the development and testing platforms.

The heterogeneous programming of the different VI algorithm implementations was made using OpenMP 3.1., the 2017 release of TBB, and OpenCL 1.1.

For version control and coordination we used a Bitbucket repository.

A.2 Matlab

The development, testing and exploitation of the robot controller was made in Matlab 2016a and 2016b. We also used it for data acquisition, post-simulation processing and analysis.

A.3 V-REP

We introduced the educational version of V-REP (V-REP PRO EDU, version 3.3.2.)[14] robot simulator in chapter 3. Here we leave a comment about the use we gave it:

It as almost impossible to make V-REP run faster than real-time. A work-around this limitation is to execute several headless instances of V-REP (no GUI) on the same

computer or remotely in order to get more simulation data in less time. The number of instances is limited by the computer resources.

Option 1: Using a "ssh -X" remote connection to a server

```
>> ssh -X username@abc.com
>> cd path_to_vrep_install_directory
>> xvfb-run --auto-servernum --server-num=1 -s "-screen 0 640x480x24"
./vrep.sh -h -s -gREMOTEAPISERVERSERVICE_40000_FALSE_FALSE
/path_to_your_scenario/my_scenario.ttt
```

Option 2: Using your local PC or a VNC/Teamviewer/etc. remote connection

```
>> cd path_to_vrep_install_directory
>> ./vrep.sh -h -s -gREMOTEAPISERVERSERVICE_40000_FALSE_FALSE
/path_to_your_scenario/my_scenario.ttt
```

Note1: in `-gREMOTEAPISERVERSERVICE_40000_FALSE_FALSE`, 40000 is the port number. You can use any available port number on your machine. Each one of the instances must have associated a different port number!

Note2: for Option 1 you need to have installed *xvfb* (X Virtual Frame Buffer) previously. There are plenty of resources about it on the internet.

Note3: this example works for linux systems; you should change ./vrep.sh for vrep.exe if you are using a windows operating system.

Note4: For a more detailed explanation of command line v-rep execution check the following reference: http://www.coppeliarobotics.com/helpFiles/en/commandLine.htm (accessed in June 2017).

Bibliography

[1] "Crumb: Systems engineering and automation dpt. (2017)," *Official web page of the CRUMB robot. http://babel.isa.uma.es/crumb*, Consulted 10th of May, 2017.

[2] A. Munir, A. Gordon-Ross, and S. Ranka, *Modeling and optimization of parallel and distributed embedded systems*. John Wiley & Sons, 2015.

[3] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Scientific Programming*, vol. 18, no. 1, pp. 1–33, 2010.

[4] S. Ruiz and B. Hernández, "A parallel solver for markov decision process in crowd simulations," in *Artificial Intelligence (MICAI), 2015 Fourteenth Mexican International Conference on*, pp. 107–116, IEEE, 2015.

[5] D. Noer, *Parallelization of the Value-Iteration algorithm for Partially Observable Markov Decision Processes*. PhD thesis, B. Sc. Thesis. Department of DTU Compute, Technical University of Denmark, 2013.

[6] P. Chen and L. Lu, "Markov decision process parallel value iteration algorithm on gpu," in *2013 International Conference on Information Science and Computer Applications (ISCA 2013)*, Atlantis Press, 2013.

[7] B. Lacerda, D. Parker, and N. Hawes, "Multi-objective policy generation for mobile robots under probabilistic time-bounded guarantees," 2017.

[8] S. Zhang, P. Khandelwal, and P. Stone, "Dynamically constructed (po)mdps for adaptive robot planning," in *AAAI Conference on Artificial Intelligence (AAAI), San Francisco, USA*, 2017.

[9] A. OpenMP, "The openmp api specification for parallel programming," *Official web site of OpenMP. http://openmp.org*, 2010.

[10] Intel, "Threading building blocks (intel(r) tbb), 2017," *Official web site of Threading Building Blocks. https://www.threadingbuildingblocks.org/*, Consulted 22th of May, 2017.

[11] K. O. W. Group *et al.*, "The opencl specification version 1.1. khronos group, 2011."

[12] F. Corbera, A. Rodríguez, R. Asenjo, A. Navarro, A. Vilches, and M. J. Garzarán, "Reducing overheads of dynamic scheduling on heterogeneous chips," *arXiv preprint arXiv:1501.03336*, 2015.

[13] O. Sigaud and O. Buffet, *Markov decision processes in artificial intelligence*. John Wiley & Sons, 2013.

[14] C. Robotics, "V-rep, virtual robot experimentation platform, 2017," *Official web site of Copelia Robotics. http://www.coppeliarobotics.com*, Consulted 12th of May, 2017.

[15] "Gazebo: Open source robotics foundation, inc. (2017)," *Official web site of Gazebo. http://gazebosim.org/*, Consulted 12th of May, 2017.

[16] "Vrep: Coppelia robotics. (2017)," *Official web site of V-REP. http://www.coppeliarobotics.com/*, Consulted 12th of May, 2017.

[17] I. F. Vega, *Development od a programming envitonment for a simulated TurtrleBot-2 robot with a WindowsX manipulator arm through the connection of V-REP and MATLAB*. PhD thesis, B. Sc. Thesis. University of Málaga, 2016.

[18] Intel, "Openmp loop scheduling," *Official web site of Intel. https://software.intel.com/en-us/articles/openmp-loop-scheduling*, Consulted 12th of May, 2017.

[19] A. Vilches, R. Asenjo, A. Navarro, F. Corbera, R. Gran, and M. J. Garzaran, "Adaptive partitioning for irregular applications on heterogeneous cpu-gpu chips," *Procedia Computater Science*, vol. 51, pp. 140–149, 2013.

[20] H. ODROID, "Odroid-xu3," *Official web site of Hardkernel. http://www.hardkernel.com*, Consulted 12th of May, 2017.

[21] R. Dementiev, T. Willhalm, O. Bruggeman, P. Fay, P. Ungerer, A. Ott, P. Lu, J. Harris, P. Kerly, and P. Konsor, "Intel performance counter monitor - a better way to measure cpu utilization,," tech. rep., Intel, 2012.

[22] T. Willhalm (Intel), R. Dementiev (Intel), and P. Fay (Intel), "Intel performance counter monitor - a better way to measure cpu utilization, 2017," *Official web site of Intel. https://software.intel.com/en-us/articles/intel-performance-counter-monitor*, Consulted 20th of May, 2017.

[23] S. E. Maxwell and H. D. Delaney, *Designing experiments and analyzing data. A model comparison perspective.* Lawrence Erlbaum Associates Pulbishers, ISBN 0-8058-3718-3, 2004.