UNIVERSIDAD
DE MÁLAGA

Escuela de Ingenierías Industriales
Programa de Doctorado en Ingeniería Mecatrónica
Departamento de
Arquitectura de Computadores

TESIS DOCTORAL

Towards energy efficiency and productivity for decision making in mobile robot navigation

Denisa-Andreea Constantinescu

Julio 2022

Dirigida por:
Dr. Mª Ángeles González Navarro
Dr. Rafael Asenjo Plaza

UNIVERSIDAD DE MÁLAGA

ANDALUCÍA TECH
Campus de Excelencia Internacional

Escuela de Doctorado

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña DENISA-ANDREEA CONSTANTINESCU
Estudiante del programa de doctorado INGENIERÍA MECATRÓNICA de la Universidad de Málaga, autor/a de la tesis, presentada para la obtención del título de doctor por la Universidad de Málaga, titulada: TOWARDS ENERGY EFFICIENCY AND PRODUCTIVITY FOR DECISION MAKING IN MOBILE ROBOT NAVIGATION

Realizada bajo la tutorización de RAFAEL ASENJO PLAZA y dirección de Mª ÁNGELES GONZÁLEZ NAVARRO Y DE RAFAEL ASENJO PLAZA (si tuviera varios directores deberá hacer constar el nombre de todos)

DECLARO QUE:

La tesis presentada es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, conforme al ordenamiento jurídico vigente (Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), modificado por la Ley 2/2019, de 1 de marzo.

Igualmente asumo, ante a la Universidad de Málaga y ante cualquier otra instancia, la responsabilidad que pudiera derivarse en caso de plagio de contenidos en la tesis presentada, conforme al ordenamiento jurídico vigente.

En Málaga, a 23 de Junio de 2022

Firmado por CONSTANTINESCU DENISA-ANDREEA - ****9471* el día 23/06/2022 con un certificado emitido por AC FNMT Usuarios Fdo.: DENISA-ANDREEA CONSTANTINESCU Doctorando/a	ASENJO PLAZA RAFAEL - 24250704X Digitally signed by ASENJO PLAZA RAFAEL - 24250704X Date: 2022.06.23 13:44:45 +02'00' Fdo.: RAFAEL ASENJO PLAZA Tutor/a
GONZALEZ NAVARRO MARIA ANGELES - 33364027C Digitally signed by GONZALEZ NAVARRO MARIA ANGELES - 33364027C Date: 2022.06.23 13:47:05 +02'00' Fdo.: Mª ÁNGELES GONZÁLEZ NAVARRO	

RAFAEL ASENJO PLAZA Director/es de tesis	ASENJO PLAZA RAFAEL - 24250704X	Digitally signed by ASENJO PLAZA RAFAEL - 24250704X Date: 2022.06.23 13:45:19 +02'00'

Dra. Dña. Mª Ángeles González Navarro.
Catedrática del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

Dr. D. Rafael Asenjo Plaza.
Catedrático del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada "Towards energy efficiency and productivity for decisions making in mobile robot navigation", 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 Tesis que presenta para optar al grado de Doctor en Ingenieria Mecatrónica.

Málaga, 23 de Junio de 2022

GONZALEZ NAVARRO MARIA ANGELES - 33364027C
Digitally signed by GONZALEZ NAVARRO MARIA ANGELES - 33364027C
Date: 2022.06.23 13:47:50 +02'00'

Dra. Dña. Mª Ángeles González Navarro.
Codirectora de la tesis.

ASENJO PLAZA RAFAEL - 24250704X
Digitally signed by ASENJO PLAZA RAFAEL - 24250704X
Date: 2022.06.23 13:45:44 +02'00'

Dr. D. Rafael Asenjo Plaza
Codirector de la tesis.

UNIVERSIDAD
DE MÁLAGA

Autorización para la lectura de la Tesis e Informe de la utilización de las publicaciones que la avalan

Los abajo firmantes declaran, bajo su responsabilidad, que autorizan la lectura de la tesis de la doctoranda Dña. Denisa-Andreea Constantinescu con NIE Y4339471Y titulada “Towards energy efficiency and productivity for decisions making in mobile robot navigation” y que ninguna de las publicaciones que avalan dicha tesis han sido utilizadas en tesis anteriores.

Málaga, 26 de Junio de 2022

GONZALEZ NAVARRO MARIA ANGELES - 33364027C

Digitally signed by GONZALEZ NAVARRO MARIA ANGELES - 33364027C
Date: 2022.06.23 13:48:27 +02'00'

Dra. Dña. Mª Ángeles González Navarro.
Directora de la tesis.

ASENJO PLAZA RAFAEL - 24250704X

Digitally signed by ASENJO PLAZA RAFAEL - 24250704X
Date: 2022.06.23 13:46:24 +02'00'

Dr. D. Rafael Asenjo Plaza
Tutor y Director de la tesis.

To my family

Dedic această lucrare familiei mele

Acknowledgments

I stand on the shoulders of giants—Sir Isaac Newton. I am fortunate to have been born in a place and time that has allowed me to travel, meet, and learn from many wonderful people.

MA, Rafa, and JA, I could not have asked for more supportive, pragmatic, and visionary advisors. True role models for the career path I am only starting to walk on, partly because of you. When I see you working, I think, *now, this is the kind of commitment and passion for lifelong learning and teaching that I want to have in 30 and even 60 years from now!* JA, many thanks for your insights on (PO)MDPs and Bayesian thinking. David Kaeli, thank you for taking me under your wing during my stay at NUCAR. You've been a great mentor and inspiration.

Ana+, Sonia, Oscar, Francisco, Inma, Julio, Gloria, Ricardo, Chema, Eligius, Andrés, Julian, Javi,... I am lucky to have had you as my teachers, colleagues, and friends. From each of you, I carry a little lesson about life and the workings of academia. I hope we'll work together on many other projects to come!

My dear colleagues, with whom I have shared the experience of pursuing a Ph.D., Alejandro, José Carlos, JM Herruzo, Fran, Iván, Manuel, Elmira, Shi, Xiangyu, Leming, Nico, Andrés,... I thank you all for your friendship and insightful conversations. It has been a pleasure and an honor.

I am grateful to the outstanding administrative and IT staff at DAC: Carmen, Paco, Juanjo, you make everyone's life at work so much easier.

Special thanks to my partner and family: I dedicate this work to you, mum, dad, Alina, Milu, and Pedro. You guys are always by my side, cheering me up when I am down and tempting me to chill and have fun when I most need it. And of course, Anastasia and Silvia, thanks for being my best friends at all times.

Thank You, dear reader, for taking the time to go through my PhD story.

Finally, I thank the projects that have supported this research. This thesis has been mainly supported by the Spanish MECD under grant TIN2016-80920-R, and partly by PID2019-105396RB-I00, P20-00395-R, UMA18-FEDERJA-108, UMA18-FEDERJA-113, B4-2021-12, Intel, and IMFAHE.

Abstract

Our goal in this work is to make it easy and feasible to implement solutions for autonomous decision-making and planning under uncertainty on low-power mobile platforms. We focus on practical applications, such as autonomous driving and service robotics, that must run on mobile SoC platforms. These applications often have real-time execution constraints. The main challenge is to keep the runtime and energy performance in check while enabling the users (programmers) to code efficient solvers for decision-making problems. Our proposal uses low-power heterogeneous computing strategies, sparse data structures to fit real-world size decision-making problems on SoCs with scarce memory and computing resources, and oneAPI with DPC++ programming.

In the first part of this thesis, we compare three heterogeneous scheduling strategies to run parallel code on CPU+iGPU SoCs. We evaluate their performance on a set of benchmarks for planning sequences of actions for mobile robot navigation. The benchmarks compute an optimal navigation plan through Value Iteration algorithm—a fundamental method to find optimal policies in decision making under uncertainty, allowing an intelligent agent modeled as Markov Decision Processes to act autonomously. Our experimental results show that the implementations based on oneAPI programming model are up to 5× easier to program than those based on OpenCL while incurring only 3 to 8% overhead.

In the second part, we take the lessons learned from optimizing Value Iteration for low-power execution and apply them to a more complex autonomous decision-making framework that accounts for all sources of uncertainty in the agent interaction with the environment—Partially Observable Markov Decision Processes. We propose a new method for online planning under uncertainty for POMDPs, *Recall-Planner*, that outperforms the state-of-the-art online planners for a known set of real-time navigation benchmarks.

This research demonstrates that it is feasible to solve large-scale (Partially Observable) Markov Decision Processes in real-time using low-power heterogeneous CPU+iGPU platforms. We can achieve both performance and productivity if we carefully select the scheduling strategy and programming model. In particular, we remark that the oneAPI programming model creates new opportunities to improve productivity, performance, and efficiency in low-power systems.

Contents

Acknowledgments **i**

Abstract **iii**

Contents **viii**

List of Figures **xi**

List of Tables **xiii**

Glossary **xv**

1.- Introduction **1**

1.1 Motivation 2

1.2 Objectives 3

1.3 Document structure 3

2.- Background **5**

2.1 MDPs and the Value Iteration Method 5

2.1.1 The Value Iteration Method 7

2.2 POMDPs and Point-Based Methods 8

2.2.1 The Point-Based Value Iteration Method 10

2.2.2 Particle Filters for Belief State Estimation 11
2.2.3 Sequential Importance Sampling 12
2.3 Programming heterogeneous MPSoCs 14

3.- Solving Large MDPs Optimally on Mobile Platforms 17
3.1 Introduction 17
3.2 Research Methodology 20
3.2.1 Modeling, Simulation and Validation of the Use-Case 21
3.2.2 Solving Large MDPs Optimally on Low-Power SoCs 24
3.2.3 Performance and Productivity Evaluation 25
3.2.4 Mobile Robot Navigation MDP Use-Case 26
3.3 Related Work 29
3.4 Solving Large-Scale MDPs Optimally on Mobile Platforms 31
3.4.1 Data Structures – 3D-lite-CSR 31
3.4.2 Taxonomy of VI Implementations 34
3.4.3 Sequential Implementation 36
3.4.4 CPU-Only Implementations 37
3.4.5 CPU+GPU Heterogeneous Implementations 39
3.4.5.1 Hybrid-1: Exploring Functional CPU-GPU execution 40
3.4.5.2 Hybrid-2: Exploring Heterogeneous Scheduling . . 40
3.4.5.3 Programming Interface 43
3.5 Evaluation and Experimental Results 47
3.5.1 Experimental Setup 47
3.5.2 Exploring the Limits of Mobile Platforms for Different MDP Sizes 49
3.5.3 Productivity Evaluation: OpenCL vs. oneAPI 51
3.5.4 Looking for the Sweet Spot: Productivity vs. Efficiency vs. Performance 53

3.5.4.1 Impact of the Programming Model 53
3.5.4.2 Impact of the Scheduling Strategy 54
3.5.4.3 Study of Offline-Tuned Schedulers: HO and HD . 58
3.5.4.4 Study of the Adaptive Scheduler: HL 59
3.5.5 Time and Energy Consumption Evaluation 61
3.5.5.1 Scalability of Implementations 62
3.5.5.2 Time Efficiency vs Energy Efficiency 66
3.5.6 Summary of Results and Lessons Learned 70
3.5.7 Rules of Thumb 71
3.6 Conclusions 71

4.- Online Planning on Mobile Platforms for POMDP Agents 74
4.1 Introduction 74
4.2 Related Work 77
4.3 Efficient Memory Mechanisms for Storing and Retrieving Online Planning Experience 80
4.3.1 Similarity Search in Belief Space 81
4.3.2 Considerations in Designing the Experience Memory 85
4.4 Online Planning with Bloom Filter Memory 87
4.4.1 Data Structures – Bloom Filter Memory 87
4.4.2 The Recall-Planner Algorithm 89
4.5 Implementation Strategies and Guidelines 94
4.6 Parallel Implementation of Recall-Planner 95
4.7 Evaluation and Experimental Results 98
4.7.1 Methodology 98
4.7.2 Experimental Setting 99
4.7.3 Preliminary Exploration and Tuning 101
4.7.4 BF Memory Overhead Evaluation 105

4.7.5 Exploring the Limits of Recall-Planner 110
4.7.6 Summary of Performance Results 116
4.8 Conclusions 119

5.- Conclusions 121

5.1 Summary of Contributions 122
5.1.1 Previous Work 122
5.1.2 Stage 1: Solving Large MDPs Optimally on SoCs 123
5.1.3 Stage 2: Online Planning for POMDPs on SoCs 124
5.1.4 Answer to Research Question 126

6.- Future Work: Decision Making for Real-World Scenarios 127

6.1 Application Architecture Proposal 127
6.2 Modeling People Intentions as a POMDP for Social-Aware Navigation 131

Appendices 133

A.- Resumen en español 133

Bibliography 135

List of Figures

2.1 VI algorithm. 8
2.2 PBVI value update pseudo-code. 11
2.3 Example of heterogeneous MPSoC. 15
3.1 Model-based vs model-free decision making and planning strategies. 18
3.2 Outline of the work done in this chapter. 20
3.3 Learning the underlying model for the MDP 22
3.4 The CRUMB robot: physical (a) and simulated (b). 22
3.5 Robot simulation setup for MDP model 23
3.6 Sparse representations of the transition probability matrix, T. . . . 32
3.7 T matrix sparsity. 34
3.8 VI control flow graph. 36
3.9 CPU-only: OpenMP and TBB VI implementations. 37
3.10 OpenMP implementation of VI 38
3.11 TBB implementation of VI. 38
3.12 Hybrid-1 implementation of VI. 40
3.13 Control flow graph for heterogeneous implementation of VI. 41
3.14 ValueIteration class. 43
3.15 Structure of heterogeneous implementations 44
3.16 Pseudo-C++ code of `ViBodyUSM` functor 45
3.17 Pseudo-C++ code of `ViBodyBUFF` functor 45

3.18 Cyclomatic complexity and and programming effort 52
3.19 GPU-only vs heterogeneous 53
3.20 Energy and speedup: heterogeneous vs CPU/GPU-only 55
3.21 Time and energy ratios for heterogeneous vs CPU-only 56
3.22 oneAPI vs OpenCL 57
3.23 HO and HD Chunk Size exploration 58
3.24 Adaptive vs offline-tuned scheduling 59
3.25 HL GPU Chunk Size exploration on Kaby-Lake 60
3.26 HL GPU Chunk Size exploration on Broadwell-Desktop 60
3.27 Time and energy consumption evaluation 63
3.28 Speedup and energy improvement 64
3.29 Normalized energy vs execution time 66
3.30 Normalized execution time for medium and large problems 68
3.31 Normalized energy for medium and large problems 69
4.1 Online planning for a POMDP agent. 75
4.2 POMDP Taxonomy. 77
4.3 Bloom filter memory diagram. 88
4.4 Recall kernel pseudo-code 91
4.5 Sequence diagram of Recall-Planner 93
4.6 Parallel implementation of BF Memory 97
4.7 Preliminary results - Tag benchmark 102
4.8 Experience recalls 103
4.9 Recall depth exploration 104
4.10 Basic planning 106
4.11 Recall planning - case 2 106
4.12 Recall planning - typical 107
4.13 Memoryless-search time distribution 107
4.14 Recall time distribution 108

4.15 Recall+search time distribution 109
4.16 Reward exploration 112
4.17 Planning time on Kaby Lake - Tag 113
4.18 Planning time on Kaby Lake - LaserTag 114
4.19 Planning time on Tiger Lake - Tag 115
4.20 Planning time on Tiger Lake - LaserTag 116
4.21 Planning time of parallel implementation 117
6.1 ROS navigation stack proposal. 128
6.2 AgileX AI and mobile robot research kit. 131

List of Tables

3.1 A qualitative review of related works. 30
3.2 Testbeds description 48
3.3 MDP problem sizes 50
3.4 SEQ1 mean time and energy 55
3.5 Optimal workload distribution 56
3.6 Energy and time task grain for SEQ1. 63
3.7 Mean time and energy of TBB implementation 65
3.8 Mean time and energy of heterogeneous implem. 65
3.9 Best performance and implementations 67
3.10 Energy bounds 68
4.1 Representative POMFP methods. 80
4.2 POMDP benchmarks 100
4.3 POMDP testbeds 101
4.4 Precomputed Bloom filter parameters 111
4.5 Execution time evaluation 118

Acronyms

AEMS Anytime Error Minimization Search. 79

ANOVA ANalysis Of VAriance. 26, 61, 62, 67, 124

BF Bloom Filter. x, 82, 83, 85–90, 94, 95, 97–99, 101, 102, 105, 108, 110, 117

CPU Central Processing Unit. 2, 3, 18, 19, 25, 30, 31, 34, 35, 37–46, 53, 54, 58, 62, 64, 72, 123

CSR Compressed Sparse Row. 27, 29

DESPOT DEterminized Sparse Partially Observable Tree. 79, 94

DPC++ Data Parallel C++. 35, 44, 72, 97, 117, 121

FPGA Field-Programmable Gate Array. 40, 46, 72, 101

GPU Graphics Processing Unit. x, 2, 18, 19, 24–26, 30, 31, 33–35, 37, 39–44, 46, 49, 52–54, 56, 58–61, 64, 71, 72, 101, 117, 123

HCP Heterogeneous Computing Platform. 2, 3, 19, 20, 30–32, 47–49, 62, 69, 71, 101

HSVI Heuristic Search Value Iteration. 79

IMU Inertial Measurement Unit. 130, 132

IS Importance Sampling. 12, 13

MCTS Monte Carlo Tree Search. 80

MDP Markov Decision Process. ix, 3, 5, 7, 17–22, 24–29, 31, 36, 44, 47, 49–51, 54, 59, 61, 62, 64, 65, 67–69, 78, 124

PBVI Point Based Value Iteration. 10, 11, 75, 79

PF Particle Filter. 10, 14

PI Policy Iteration. 29

POMCPOW Partially Observable Monte Carlo Planning With Observation Widening. 79

POMDP Partially Observable Markov Decision Process. x, 8–10, 13, 74–80, 83, 89–92, 98–100, 110, 111, 119, 128, 131

RL Reinforcement Learning. 28

ROS Robotic Operating System. 21–23, 130, 132

SIMT Single Instruction, Multiple Threads. 24

SIR Sequential Importance Resampling. 13

SLAM Simultaneous Localization And Mapping. 129

SoC System on a Chip. iii, 1, 56, 72, 119

SPMD Single Program, Multiple Data. 24

UCB Upper Confidence Bound. 91

USM Unified Shared Memory. 35, 45, 46, 51–54, 56, 57, 71–73

VI Value Iteration. ix, 7, 19, 20, 22, 24–26, 28–31, 35–39, 43, 47, 49, 50, 54, 60–62, 69–71, 79, 123

1 Introduction

(Partially Observable) Markov Decision Processes, or (PO)MDPs, are powerful frameworks to model how intelligent agents operate in the physical world. In most real-world problems, sensory perception is imperfect, and the agent-world dynamics are rarely entirely known. Thus, the system is not fully observable, and any action's outcome is not fully controllable either. When planning, the ability to account for uncertainty comes at a high computational and memory cost. In this work, we research and propose answers to the question: *is it possible to solve large-scale decision-making problems on mobile consumer platforms, despite their inherent limited computing, memory, and energy resources?*

Given the current rate at which consumer embedded and mobile systems have been evolving in the last decades, we are inclined to assume that mobile SoCs are, or will soon be, capable of running complex decision-making applications. We know for a fact that the development of mobile SoC hardware—driven by the need to increase the performance and energy efficiency and simultaneously reduce the cost and size of chips—shifts towards heterogeneity and specialization. Unfortunately, running code on heterogeneous SoCs is fundamentally more complex than on single-core processors, posing a major challenge for programmers to use these computing platforms efficiently.

In summary, in this research, we make a thorough evaluation of the energy, performance, and productivity of modern heterogeneous computing techniques to solve computationally complex decision-making problems. We use a selection of representative low to medium power CPU+iGPU SoCs as testbeds, all consumer platforms likely to be found onboard a mobile phone, laptop, and even a mobile robot. We propose a series of strategies to optimize the memory, energy efficiency, runtime, and productivity of the state-of-the-art methods for online planning in (PO)MDPs. Finally, we illustrate our study with mobile robot navigation tasks and provide a list of guidelines and lessons learned through this project to facilitate the transfer of best practices to other use case scenarios and platforms.

We focus our efforts on mobile robot navigation because the specific problems that mobile robots can perform are heavily influenced by the degree of energy and decision-making autonomy. Mobile robots are helpful and even indispensable in access and operations in dangerous or inaccessible environments, such as nuclear and chemical, but also in missions of civil rescue, protection, and even exploration of outer space.

In this work, we use V-REP [92] (rebranded as CoppeliaSim) robot simulation for fast prototyping with broad support for learning robots that make use of heavy use of differentiable physics. V-REP can accurately simulate RGBD+LiDAR sensors and random external forces using different options for multiple physics engines [25]. According to [17], this simulator can reproduce the behavior of the real robotic system very accurately. For a test with an e-PuCK robot and camera SLAM localization, the simulation had a maximum error of 2cm.

1.1 Motivation

We formulate the motivation for this research in the context of the overarching challenge of our research group, i.e., to hide complexity without compromising performance, applied to decision-making under uncertainty for mobile robot navigation. This work builds upon previous research in our group on techniques that enable the simultaneous execution of the workload of a given software by using both the CPU cores and the integrated GPU on high-performance HCPs to low-power SoCs [29, 80].

In the literature study, we have noticed the knowledge gap between the state-of-the-art implementations of decision-making research for physical systems and state-of-the-art methods in parallel and heterogeneous computing. Proving the feasibility of planning for large (PO)MDPs is insufficient, so we attempt to facilitate and simplify the use of cross-domain know-how in practice. In particular, we target use case scenarios where energy consumption awareness is mandatory and close to real-time response restrictions may be required too.

In particular, we aim to provide a generalizable approach to help professionals and researchers implement energy-efficient decision-making applications on heterogeneous mobile SoCs by using a variety of well-known parallel programming models, namely OpenMP, TBB, OpenCL, and oneAPI.

1.2 Objectives

Our main objective is to investigate and improve the feasibility of solving large (PO)MDPs in near real-time while minimizing the overall energy consumption. We quantify feasibility in this context by the ease of programming efficient decision-making applications for mobile and physical intelligent agents.

As a means to it, our first action is exploring the capabilities and limitations of low-power Heterogeneous Computing Platforms (HCPs). In particular, we study the pervasive mobile SoCs integrating a CPU and GPU, commonly used in the smartphone and laptop industry. From a hardware point of view, we are interested in how well they can handle performance and power efficiency, while from a software point of view, our focus point is on productivity—how easy it is to program these SoCs.

Following, we list the main steps taken towards reaching our objective:

1. Study the state-of-art for online planning in (PO)MDPs.
2. Propose data structures and methods to reduce memory and computation constraints for solving large (PO)MDPs on mobile platforms.
3. Evaluate and compare the energy and runtime performance of (1) existing heterogeneous programming models and (2) scheduling techniques for partitioning and load balancing the execution on low-power CPU+GPU SoCs.
4. Propose directions for implementing (PO)MDP planning solutions suitable for (near) real-time use onboard of a mobile platform, like a mobile robot, tablet, or smartphone.

1.3 Document structure

The rest of this work is organized as follows. In Chapter 2, we introduce two core decision-making frameworks in the literature: Markov Decision Processes and Partially observable Markov Decision Processes, which are the foundation of the research in this thesis. Then, in Chapter 3, we study how to efficiently solve MDP problems on low-power CPU+iGPU SoCs by leveraging modern heterogeneous programming models. Next, in Chapter 4, we use the lessons learned from Chapter 3 to propose a novel solution for online planning in POMDP. Finally, in Chapter 5, we present the conclusions of this work, followed by a plan to use the proposed methods in this thesis in a real-world use-case for social-aware mobile robot navigation, in Chapter 6.

2 Background

The research in this thesis builds upon two core frameworks in sequential decision making under uncertainty: Markov Decision Processes (MDPs), used in Chapter 3, and Partially Observable Markov Decision Processes (POMDPs), used in chapter 4. We dedicate this chapter to introducing how MDPs and POMDPs are used to model intelligent agents and the methods we use as a foundation for planning in these frameworks. We end the chapter by presenting an introduction to heterogeneous multiprocessing SoCs and the supported open parallel programming models and tools 2.3.

2.1 MDPs and the Value Iteration Method

A discrete MDP is defined as a tuple $< S, A, T, r >$, where $< S, A >$ is a directed graph, T the so-called *transition* function, and r a *reward* function. More concretely, S is a finite set of states where the agent and its environment may be at a given time. A is a finite set of actions that the agent may take. Each action produces a stochastic change of the current state; this gets the decision-making agent a certain reward. The state transition encodes the uncertain dynamics in the system, governed by $T = P(s_{t+1}|s_t, a_t)$. This equation also reflects the *Markovianity* of the process, i.e., the next state, s_{t+1}, only depends on the current state, s_t, and the selected action, a_t. The reward, in turn, is defined by $r : (S \times A) \to \mathbb{R}$ and associates state-action pairs to real numbers.

The reward establishes the problem to solve (the goal the agent should attain) in an indirect and intuitive form: optimizing that reward optimizes the decision-making policy. Rewards can also have uncertainty, but we are particularly interested in the expected reward of being in state s and taking action a, disregarding the next state, which can be defined as:

$$R(s,a) = E_{succ(s)}[r(s,a)] = \sum_{s' \in succ(s)} T(s'|s,a) r(s,a) \tag{2.1}$$

In Eq. 2.1, *succ*(s) is the set of reachable states when taking action a while being in state s.

A *policy* $\pi : S \to A$ provides the action that should be taken at any state. Usually, policies are *stationary*, that is, they always indicate either the same action (deterministic policies) or a probability distribution over actions (stochastic policies), given the current state. Here we focus on stationary, deterministic policies. Each such policy π can be assigned a *value* V_π (a vector of real numbers that map to S, each one indicating the utility of the policy when started at that state) that allows us to find the best one for some criteria of optimality.

The most common criterion of optimization is the *expected total discounted reward*, which would be accumulated after an infinite number of actions are taken starting at each state s and using π to decide the action at every step. See Eq. 2.2 for details: $\gamma \in (0,1)$ is the *discount factor* (defines how myopic the algorithm is) and *seq*(s) is the set of all possible sequences of states followed from s according to π. The discount factor represents the difference of importance between future and present rewards. When γ equals one, all rewards are equally important in the decision making, while γ equal to zero indicates that only the immediate reward matters.

$$V_\pi(s) = E_{seq(s)}[r(s_1, \pi(s)) + \gamma r(s_2, \pi(s_1)) + \gamma^2 r(s_3, \pi(s_2)) + \dots] \tag{2.2}$$

The value function has a recursive form if we use the linearity of expectation:

$$V_\pi(s_k) = R(s_k, \pi(s_k)) + \gamma E_{seq(succ(s_k))}[V_\pi(s_{k+1})] \tag{2.3}$$

or, in more detail:

$$V_\pi(s_k) = R(s_k, \pi(s_k)) + \gamma \sum_{s' \in succ(s_k)} T(s'|s, \pi(s)) V_\pi(s_{k+1}) \tag{2.4}$$

This is the so-called *Bellman equation.* In the following, we need a more relaxed definition of value function, when the first action is not forced by the policy:

$$Q_\pi(s_k, a_k) = R(s_k, a_k) + \gamma \sum_{s' \in succ(s_k)} T(s'|s, \pi(s)) V_\pi(s_{k+1}) \tag{2.5}$$

Actually, $Q_\pi(s_k, \pi(s_k)) = V_\pi(s_k)$, thus, [2.6], which is recursive again.

$$Q_\pi(s_k, a_k) = R(s_k, a_k) + \gamma \sum_{s' \in succ(s_k)} T(s'|s, \pi(s)) Q_\pi(s_{k+1}, \pi(s_{k+1})) \tag{2.6}$$

2.1.1 The Value Iteration Method

Solving an MDP consists in finding a policy that is optimal under some definition of value, such as the expected total discounted reward explained before. For this case, it can be demonstrated that at least one policy π^* exists that has a maximum value (it is not required to be unique).

T-based methods for solving MDPs through dynamic programming use the Bellman equation in these forms, where the (*) superscript means "optimal":

$$\forall s : V_{\pi^*}(s) = \max_{a \in A} Q_{\pi^*}(s, a), \; \pi^*(s) = \arg\max_{a \in A} Q_{\pi^*}(s, a)$$

These methods serve to set up iterative optimization procedures. In particular, Value Iteration (VI) iterates on the value of Q until close-enough convergence to the optimal Q^*, and consequently to V^* and π^*, is achieved. It calculates at each iteration step i a new $Q^i(s, a) = R(s, a) + \gamma \sum_{s' \in succ(s)} T(s'|s, a) V^{i-1}(s')$, then gets V^i and π^i from that Q^i, then repeats everything until, for instance, a suitable proximity between consecutive V occurs.

VI method, whose pseudo-code is shown in Fig. 2.1, has a superlinear complexity on the number of states and actions. This makes it impractical when the state or action space is large, which is common practice. It iterates on the execution of two kernels, "Evaluate Policy", with a complexity of $O(|A||S|^2))$, and "Improve Policy" with a complexity of $O(|A||S|)$, until the expected total discounted reward does not improve over a given threshold θ.

The VI procedure of Fig. 2.1, as in [99], needs as inputs the set S of states, the set A of actions, the transition distribution matrix T specifying *T(s'|s,a)*, the expected reward function R for every state-action pair, a threshold that specifies whether we consider that the algorithm has converged to the optimal policy, $\theta > 0$, and the discount factor $\gamma \in [0, 1]$. The outputs are the last V and π.

```
procedure ValueIteration($S, A, T, R, \theta, \gamma$)
    assign $V^0(S)$ arbitrarily, $i \leftarrow 0$
    repeat
        $i \leftarrow i + 1$
        // Evaluate policy
        for each state s
            for each action a
                $V^i(s) = \sum_{s'}$T(s'|s,a)$(R(s,a) + \gamma V^{i-1}(s'))$
            end
        end
        // Improve policy
        for each state s
            $\pi^i(s) = argmax_a \sum_{s'}$T(s'|s,a)$(R(s,a) + \gamma V^i(s')$
        end
    until $\forall s|V^i(s) - V^{i-1}(s)| < \theta$
    // Return optimal policy and its value
    return $\pi^i, V^i$
end
```

Figure 2.1: VI algorithm.

For a more detailed theoretical treatment of MDPs and related methods, there are several excellent texts on the matter [91, 99, 117].

2.2 POMDPs and Point-Based Methods

The POMDP is a formalism used in sequential decision making for an agent with sensory perception uncertainty (sensor readings are imprecise or incomplete; therefore, the agent state is uncertain too, unlike in MDPs). As the agent cannot fully observe its underlying state, the concept of *belief*, *belief state* or *information state* is introduced. The belief is often represented as a probability distribution over the state space. A POMDP is thus a tuple $< S, A, Z, T, O, R, \gamma >$, where:

- S is the state space. It can be either continuous or discrete. Here we study the discrete case.
- A is the action space, usually a small set of discrete actions, even though they may represent a continuous quantity such as a velocity (it could be discretized in "accelerate, decelerate, stop", or through incremental updates over the current velocity value).
- O is the observation space. An observation offers partial information about the state of the system. It can be either continuous or discrete.
- $Z(s', a, o) = p(o|s', a)$ is the observation model and models the probability that the agent observes o when reaching state s' after executing action a.

In other words, it provides indirect, noisy access to the state. We use here a finite set of possible observations.

- T is the state transition function, $T(s,a,s') = p(s'|s,a)$, i.e., the probability of transitioning to state s' from state s by performing action a. The probability of transition from the current state to any other state adds to 1. The same principle applies to Z.
- $R(s,a)$ is the reward obtained when taking action a from state s.
- $\gamma \in (0,1)$ is the discount factor that ponderates the current reward relative to the past ones, as in MDPs. Its effect is that immediate rewards weigh more than future rewards.

The primary source of uncertainty in a POMDP is modeled by maintaining an information state called belief, $b \in B$, representing a probability distribution over the state space S. In practice, a particle-based representation of the belief is commonly used—an approximation that reduces the belief to a number of sampled points, intended to be condensed in the vicinity of the actual state of the agent. This is called "belief space sampling", and it goes hand in hand with the "point-based methods", the state-of-the-art in online POMDP solvers that can handle very large POMDPs. Everything we know about MDPs applies to POMDPs, only that instead of having a tabular format, the policy maps over information states.

At any time t, the agent is in one of the possible states, s (recall that it only knows its belief b), and must take an action a among a finite set of possible actions. Taking action a results in an immediate reward $r(s,a) \in \mathbb{R}$ and in a transition to state s' with probability $p(s'|s,a)$. After each transition, the agent makes an observation o with probability $p(o|s',a)$ (or $p(o|s')$, a common simplification in the literature). Based on the observed information, the belief state b is updated to b'. This can be done with the following formula in the case of continuous belief representations [1]:

$$b' = b(s') = \tau(b,a,o) = \frac{p(o|s',a)}{p(o|b,a)} \int_s p(s'|s,a)b(s)ds \tag{2.7}$$

For discrete belief representations, we have:

$$b' = b(s') = \frac{p(o|s',a) \sum_{s \in S} p(s'|s,a)b(s)}{p(o|b,a)} \tag{2.8}$$

---

[1] This is a Recursive Bayesian Filter in the literature on the estimation of the state of dynamical systems [97].

where

$$p(o|b, a) = \sum_{s' \in S} p(o|s', a) \sum_{s' \in S} p(s'|s, a)b(s) \tag{2.9}$$

is a normalization constant that can be dropped out if Eq. 2.8 is turned into a proportionality.

This discrete representation is similar to a particle filter, where $1/p(o|b, a)$ is equivalent to the normalization factor. PFs have been successfully used for decades for state approximation, with practical application in robot localization, in particular, for Simultaneous Localization and Mapping (SLAM) [33].

Solving a POMDP consists in finding a policy $\pi : B \to A$ that maps a belief $b \in B$ to action $a \in A$, so that it maximizes some definition of value, such as the expected total discounted reward. The discount rate, which controls how much future rewards count compared to present ones, γ, indirectly defines how greedy the decision-making is.

A key concept when solving POMDPs is the belief tree that emerges from exploring all possible actions and likely observations (edges) starting at a given belief (node). An *optimal policy* results from picking the branch of the belief tree that begins at the current belief and has the maximum reward or value expected over the entire horizon. This tree is often called a decision tree.

Offline solvers are not bound by real-time restrictions and can explore the whole space to find the optimal policy. This is not feasible in some applications, thus online solvers that provide a good enough policy in real time are needed. Online solvers can scale to very large POMDPs but usually use previous knowledge less efficiently since they do forward search at every step.

2.2.1 The Point-Based Value Iteration Method

Most online methods for POMDP planning are at their core a variation of Point Based Value Iteration (PBVI), so it is worthwhile presenting the original algorithm. PBVI addresses the curse of history—the planning complexity grows exponentially with the planning horizon [87], as a function of possible histories, or sequences of actions and observations, from the first iteration to planning timestep t, and can be approximated by $O(|A|^{|O|^{t-1}})$—by limiting the planning to a reduced set of likely beliefs. What made it so successful is the idea of selecting the smallest set of reachable beliefs and planning for those beliefs only. The planning involves learning their value and gradient.

The PBVI algorithm alternates between 1) growing the set B of belief points or *belief expansion* (e.g, the set doubles in size every time) and 2) planning for those belief points or *value update*. The algorithm iterates over these two steps until it **runs out of time** or has found a **good enough policy**.

PBVI is an *anytime* algorithm, meaning that the procedure returns a valid solution (policy) even if the search is interrupted before it converges to an optimal solution. The algorithm is expected to find better solutions the longer it runs, and, given sufficient time, it approximates the optimal policy with a bounded error [86], as stated in the following theorem.

Theorem

For any set of belief points B and planning horizon n, the error of the PBVI algorithms is bounded by $||V_n^B - V_n^*||_\infty \leq \frac{R_{max}-R_{min}}{(1-\gamma)^2} max_{b' \in \Delta} max_{b \in B} ||b - b'||_1$, where Δ is the set of reachable.

The general outline of the *value update* function (polynomial complexity), is the following:

```
function ValueUpdate($S, A, Z, T, O, R, B, \gamma$)
    // Initialize value functions
    ...
    for each belief $b \in B$
        for each action-observation pair, $< a, o >$
            // Project one step forward the belief -> new belief
            $b \rightarrow b^{a,o}$
            Find the best value, $\beta_b^{a,o}(s) = V_n(b^{a,o})$, for the new belief:
            // Sum over observations
            $\beta_b^{a,o}(s) = R(s,a) + \gamma \sum_{a \in A, s' \in S} T(s,a,s')O(s,a,o)\beta_b^{a,o}(s')$
        end
        // Maximize over the action space (backup)
        $V_{n+1}(b) = argmax_a \beta_b^a$
    end
end
```

Figure 2.2: PBVI value update pseudo-code.

2.2.2 Particle Filters for Belief State Estimation

Particle Filters (PF) are essential for the point-based methods used in planning for POMDPs. We are interested here in variations of particle filters used to represent uncertain beliefs states in the context of POMDPs. This representation is most commonly used in approximated online methods, making it possible to solve large POMDPs while meeting time constraints.

In a particle filter, beliefs are distributions over the state space represented as (approximated by) finite sets S_t of N weighted particles or *samples* $< s_t^i, w_t^i >$. For each $i = 1, ..., N$ and time-step t, particle i refers to a possible state, s_t^i, where the system may be at that time (also known as state hypothesis) and to a weight or importance $w_t^i \geq 0$ of that state. Weights must sum up to one, i.e., $\sum_{i=0}^{N} w^i = 1$.

Particle Filters implement this sample-based form into a Bayes filter, which recursively estimates the belief b (posterior density) on the state s_t of a dynamic system: $b(s_t) \propto P(o_t|s_t) \int P(s_t|s_{t-1}, a_{t-1}) b(s_{t-1}) ds_{t-1}$. Here, o_t is a sensor measurement and a_t is the action or control command; $P(o_t|s_t)$ models the observations (provided by sensors) while transition $P(s_t|s_{t-1}, a_{t-1})$ describes the dynamics (motion due to actuators) of the system.

The samples that approximate a belief represent the posterior of that belief after the motion/observation step: $p(s) = \sum_{i=1}^{N} w^i \delta_{s^i}$, δ_{s^i} is the Dirac distribution centered in that state. If the filter converges, it is expected that the more particles fall into a region, the higher is the probability of the agent actually being in that region.

A particle filter has two steps:

1. *Prediction*: draw samples from the proposal distribution (e.g., agent motion model).
2. *Correction*: weight the samples by a ratio of the target and proposal; the observation is used for correction.

We must decide how to generate random samples for the actual implementation of the belief. Closed-form sampling is only possible for a few distributions, such as Gaussian.

2.2.3 Sequential Importance Sampling

Importance Sampling (IS) is a simulation technique that can estimate a probability distribution with better accuracy than Monte Carlo methods. This method generates random weighted samples from an auxiliary distribution (also called the proposal distribution) instead of drawing them from the distribution of interest. The most delicate part of IS is the choice of an efficient proposal distribution for the probability density function (PDF) that can simulate (generate) the more rare random events of the target distribution. [76] proposes an approach to optimize the proposal distribution with non-parametric adaptive importance sampling (NAIS).

Importance sampling is based on the following key ideas:

- Use particle sets to represent arbitrary distributions.
- Model the target distribution through weights—the larger the set of particles and the closer their weights to the actual distribution, the better the approximation is. Particles are initially randomly distributed and therefore have equal weights.
- Use a different distribution g (proposal distribution) to generate samples for the target distribution f; account for the differences between g and f using a weight correction factor, so $w = f/g$, while f and g must meet the following precondition: $f(x) > 0 \rightarrow g(x) > 0$.

In the context of POMDPs, the task of IS is to sample from probability densities T (and sometimes O), which are not always available nor easy to obtain. A solution is to sample from a *proposal* density g and weight each particle s^i by $\frac{Target\_PDF(s^i)}{g(s^i)}$. In some cases, this expression has simple forms (not needing to know the actual target PDF).

A widely used and simple form of importance sampling is Sequential Importance Sampling with Resampling (SIR/SISR). This procedure has three steps: sampling, importance sampling, and resampling. This is an old method that has been around for nearly two decades, but only recently has it been adopted in POMDP solutions. We also use it in our solutions.

The three steps of the SIR algorithm to model the current belief of the POMDP agent, *b(s)*, are:

1) **Sampling**: sample the next state s' using the states s of the current belief and action a, according to the transition probability function $T(s'|s,a)$, which describes the dynamics of the agent. If T is not available, generate the particles using the proposal distribution.

2) **Importance sampling**: weight each s' in the sample generated in the previous step by using the actual observation gathered by the agent, o, and the observation model (e.g., camera, rangefinder, sonar; it can also encode a map of the environment), i.e., $w' = O(o|s,a,s')$.

3) **Resampling**: if needed, draw with replacement a random sample of N particles from the belief resulting from the previous step, using the weights as probabilities. Set the final weights as uniform. This sampling with replacement assures that only the samples with the highest weight survive, i.e., it is called “survival of the fittest”.

A well-known problem of SIR is the so-called *particle deprivation.* It occurs when there are no particles in the vicinity of the correct state. A possible solution is to add a small number of randomly generated particles when resampling. This method reduces particle deprivation and is simple to implement. It cuts out particles that are not consistent with past evidence, therefore it increases the chances of getting closer to the real state. However, its main disadvantage is that the posterior estimate is incorrect even in the limit of infinite particles.

Alternatively, we can choose dynamically the number of samples needed to ensure a bound on the error made on the belief approximation by using an adaptive PF as in [63]. The authors use the Kullback-Leiber (KL) divergence to measure how much a probability distribution is different from another (also called relative entropy). KL should not be confused with the distance between two distributions (Jensen-Shannon divergence, for instance, computes the distance between two distributions).

An essential parameter in any PF application is the number of particles. Larger samples result in a better estimation, but there is a memory and computational cost associated. The simplest way is to use a fixed number for each use-case (ranging from 500 to 1000 particles for use-cases of navigation in grid-worlds), which is not optimal, but is computationally efficient. We have explored adaptive strategies, but it turns out they bring little improvement in precision at a high computational cost:

- Typically, more particles are needed initially, as the uncertainty in highest. Similarly, a higher number of particles are needed if the agent gets lost (its whereabouts become uncertain). A sampling procedure taking these aspects into account is called *adaptive*, because it adapts the number of particles on the run.
- The probability of the sensors measurement, $P(z_t|z_1 : z_{t-1}, u_{1:t}, m)$, has to be constantly monitored. It can be approximated by $\frac{1}{N}\sum_{i=1}^{N} w_t^i$, and average over multiple time steps to compare typical values when having reasonably accurate state estimates. If the accuracy is low, inject random particles.

2.3 Programming heterogeneous MPSoCs

A heterogeneous multiprocessing System-on-Chip (MPSoC) is a platform that contains multiple types of computational devices optimized for performance, energy efficiency, and specialized hardware accelerators on the same chip. An example is the Samsung Exynos 5422, whose architecture is illustrated in Fig. 2.3. Its heterogeneous computing architecture (HSA) integrates two types of multicore processors, a GPU accelerator and a unified shared memory. The LITTLE processor is designed for power efficiency, while the "big" processor is for compute performance. Each multicore cluster has its dedicated shared L2 cache, the same applies to the GPU execute units, and they all share the main memory.

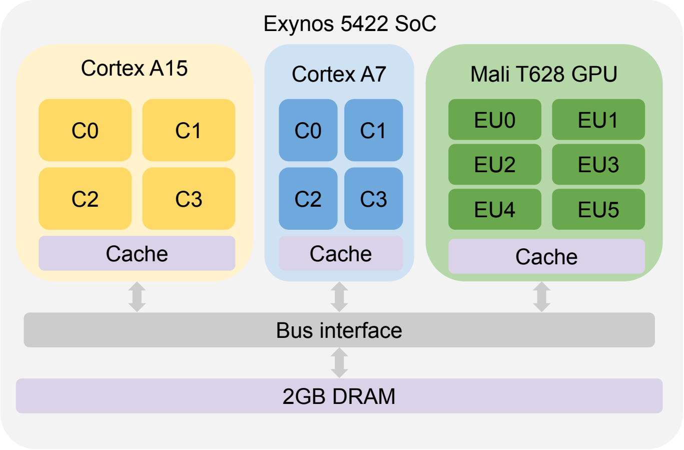


Figure 2.3: Example of heterogeneous MPSoC.

The heterogeneous computing platforms market is increasing since specialization has become an esstential means to improving energy efficiency and performance when miniaturization, increasing the computing frequency and adding more cores are no longer feasible. Unfortunately, specialization and heterogeneity make programmability and performance portability among platforms from different vendors and even from the same vendor extremely complex and unsustainable.

In the shared-memory programming model, some of the most used programming models to abstract how tasks are mapped to execute asynchronously or in parallel on the MPSoC include combinations of shared memory, message passing, data-parallel programming, and map-reduce.

The industry and academia communities have been laboriously contributing to standard and open vendor-agnostic parallel programming models in creating and improving directive-based programming models for C code and STL-based programming models for C++. Directive-based programming models are easy to use and debug, allowing the programmer to incrementally annotate their code using both industry standards as OpenMP and OpenACC and community-maintained efforts, such as OmpSs and XcalableMP.

Among the STL-based programming models, TBB (Threading Building Blocks) [2] and HPX (High Performance ParallelX) [3] are worth mentioning. TBB is a C++ template library that implements a set of parallel containers and algorithms for running code in parallel on multicore CPUs. HPX is C++ runtime system for asynchronous, parallel, and distributed computing, using CUDA, SYCL, and HCC as backend.

A plethora of vendor-specific programming models bind the users to the manufacturer, be it NVIDIA, AMD, or Intel, and we will not dive into them. Instead, we focus on OpenCL [4], the unified programming model for high-performance computing with the widest vendor support for many-core processors, GPUs, APUs, and FPGAs. It is highly portable but does not necessarily guarantee performance portability, nor is it user-friendly. Among the C++-based parallel programming models that attempt to fill the gaps, there is heavy development in SYCL cross-platform abstraction, DPC++/C++ [5] compiler and its oneAPI [6] imlementation. SYCL [7] inherits the main characteristics of OpenCL and uses it as a backend while improving on the programmability and performance portability aspects.

---

[2] https://github.com/oneapi-src/oneTBB

[3] https://stellar-group.org/libraries/hpx/

[4] https://www.khronos.org/opencl/

[5] https://intel.github.io/llvm-docs/

[6] https://www.oneapi.io/

[7] https://www.khronos.org/sycl/

3 Solving Large MDPs Optimally on Mobile Platforms

3.1 Introduction

A Markov Decision Process (MDP) is a standard framework for modeling stochastic planning and sequential decision-making under uncertainty in many disciplines, e.g., artificial intelligence, control systems, robotics, logistics, and maintenance, to name just a few [99, 11, 117, 91]. In this chapter, we focus mainly on how to efficiently *solve* a problem defined as an MDP and find a *policy* that determines the best sequence of actions for a decision-making agent. The policy that maximizes—under certain optimality criteria—the reward for the agent is called *optimal* and can only be obtained by solving the MDP *exactly*. This exactness relies on knowing the true Transition Probability Matrix (T) of the problem, i.e., the stochastic behavior or model of the decision-making agent while interacting with its environment.

The MDP model knowledge can be either given before executing the method or acquired online, during execution [99]. The most used methods for solving MDPs with explicit knowledge about T are known as *model-based* methods. They are based on *Dynamic programming* and include *Value iteration* (VI) and *Policy iteration* (PI). Other sort of methods that may converge asymptotically to the optimal policy (under suitable constraints), in spite of not using explicit knowledge about T, include *Temporal difference learning*, *Q-learning* and *SARSA*. They are known as *model-free* methods for learning and decision making, and are frequently applied in Reinforcement Learning [9, 91]. We visually illustrate some of the key differences and similarities between the model-free and model-based methods in Fig. 3.1.

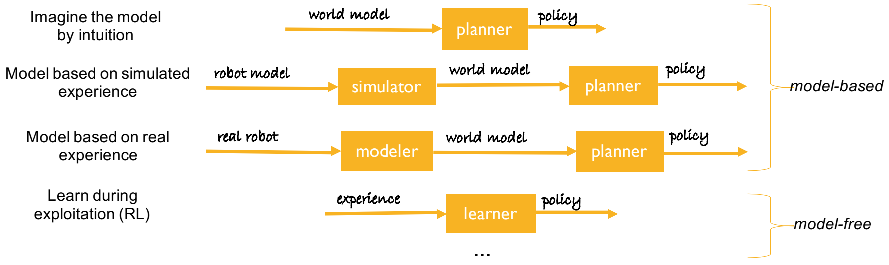


Figure 3.1: Model-based vs model-free decision making and planning strategies.

In this chapter, we deal with solving large-scale (with millions of states) tabular model-based MDPs efficiently on low-power computing platforms when using T explicitly. Notice that this can be impractical for large MDP sizes due to the "curse of dimensionality", i.e., the memory required to represent an MDP increase quadratically with the states, and the time to find the optimal solution, exponentially [8]. Model-free methods spread this computational cost over longer, smaller-grained sequences of experiences of the agent. However, this is not a clear dichotomy [9, 40, 90], since model-based methods are used in online scenarios as well, and in their approximate forms can employ at their core dynamic programming algorithms, such as VI, for progressively estimating the true T—the same as in T-based calculations. In the particular case of learning in physical environments (e.g., in robotics), model-free methods are recognized not to be as suitable as model-based ones (see Chapter 18 - Reinforcement Learning in Robotics: A Survey in [118]). This highlights the importance of dynamic programming and T-based methods (originally devised for offline and exact computation) in approximate, real-time, and physical applications.

Although the execution of decision-making algorithms in physical agents (e.g., mobile robotics, autonomous driving) reveals the important problem of having a limited energy supply, most previous works in solving large MDPs use high-performance platforms connected to an uninterruptible power source. They use one or a combination of the following three approaches to improve efficiency: exploiting parallelism on CPU (SPMD, implemented with OpenMP and vectorization) [96, 54, 122, 49], exploiting parallelism on GPU (SIMT, implemented with CUDA on discrete GPUs) [96, 46, 126, 77], and using approximate methods (parallelized for multicore and GPU execution) [109, 54]. Some of the reviewed works consider solving MDPs on low-power platforms [96], but do not evaluate the power dissipation [96, 54, 122, 46]. All these approaches have the common goal to reduce the time required to compute a policy while neglecting the energy footprint. Also, they do not exploit the full potential of simultaneous heterogeneous computing, i.e., they use one kind of device at a time —CPU or GPU, but not both simultaneously.

In contrast, we target low-power platforms that usually rely on battery power supply; in this scenario, energy consumption awareness is mandatory. Our approach to solving this kind of computationally complex problems is using low-power high-performance accelerator hardware along with multicore processors. The demand for high-performance and energy-efficient computing on consumer mobile devices such as tablets, smartphones, laptops, and gaming consoles has created the perfect conditions for the development of the ubiquitous low-power heterogeneous System on Chip (SoC) that integrates a GPU accelerator with a multicore. There is also a growing interest from the scientific community and the industry in low-power Heterogeneous Computing Platforms (HCPs) because they promise improved resource utilization, energy efficiency and an overall gain in performance cheaply. Their use in embedded and mobile systems is being extended to solve complex decision problems, e.g., in autonomous driving and service robotics [28, 110, 123].

We illustrate this study with a bare implementation of VI for mobile robot navigation, where a robot is intended to reach some metrical target from its current position while avoiding obstacles. This is not the best approach for learning this robotic task, since we simplify some parts and abstract away several details and components of a complete solution—those not related to the VI procedure itself. The implemented VI core should certainly be part of more complex methods. But it is a suitable scenario where the benefits of different approaches to reduce the computational and energy costs of decision making can be analyzed and compared.

The remainder of the chapter is structured as follows. In the next section, we present the research methodology used. Next, we explain the MDP formalism and the VI core method that we use as a benchmark, the approaches to be analyzed and studied for improving performance and programmer productivity (Section 3.2). Then, we present our experimental setup and results of the evaluation of the ease of programming, the computational time, and the energy consumption (Section 3.5). We end the chapter with a discussions and conclusions section (Section 3.6).

3.2 Research Methodology

Throughout this chapter we employ the *use-case methodology* to study and optimize the VI method for solving large decision-making making problems efficiently onboard of mobile platforms. We formally define our navigation use-case as an MDP in Subsection 3.2.4, where the main actor is CRUMB [34] —a mobile robot that has to safely navigate to a target. Our aim is to evaluate and propose techniques that improve the planning time and energy efficiency of VI in low-power computers, and we use navigation as an illustration of their use. We are not proposing a new navigation or learning method; there are a number of excellent robotic navigation algorithms that do not involve MDP decision making or learning, as [7].

This work has been developed in three main phases, illustrated in Fig. 3.2:

- *Phase 1* – Modeling, simulation and validation of the robot navigation problem as a Markov decision process.
- *Phase 2* – Implementation and optimization of VI algorithm for execution on low-power HCPs.
- *Phase 3* – Evaluation of productivity, performance and energy efficiency on the target platforms.

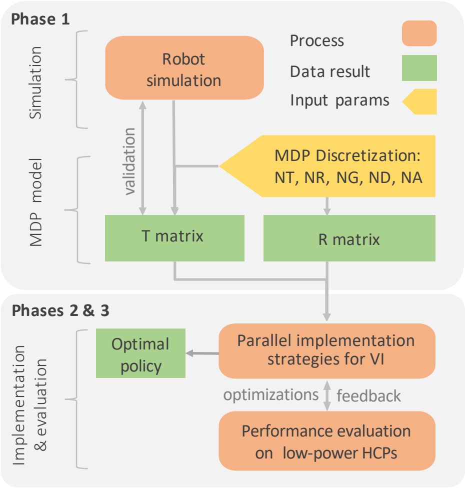

Phase 1
Simulation
MDP model
Robot simulation
Process
Data result
Input params
validation
MDP Discretization: NT, NR, NG, ND, NA
T matrix
R matrix
Phases 2 & 3
Implementation & evaluation
Optimal policy
Parallel implementation strategies for VI
optimizations
feedback
Performance evaluation on low-power HCPs

Figure 3.2: Outline of the work done in this chapter.

We use an iterative and incremental approach to improve the proposed VI implementations. This strategy is applied in all phases. First, in obtaining a valid MDP model for the robot navigation use-case (phase 1, upper part of the diagram), through repeated evaluation, improvement and validation of the interim model for the transition probability matrix (or transition function T). Second, as you may see in the lower part of the diagram (phases 2 and 3), the processes of implementation, optimization and evaluation feed each other through an optimization and feedback loop.

3.2.1 Modeling, Simulation and Validation of the Use-Case

Our use-case involves a single agent, the CRUMB robot [34], a non-holonomic mobile robot that has to navigate through a structured indoor environment to reach a given target while avoiding obstacles. The agent does not possess a map of its surroundings, but it is capable of perceiving them locally through its sensors. From an MDP perspective, the goal of this agent is to find an optimal policy to perform such navigation. The target can vary: it could be the recharging station or maybe a desk where mail has to be delivered. We define the environment as a five by five square meters space bounded by walls that can contain any number of obstacles.

To formally define the robot navigation problem as an MDP, we must describe the robot *states*, *actions*, *transition probabilities* of reaching a state from any other state by taking a particular action (i.e., T), and the *rewards matrix* (R) or utility of performing a certain action in the current state. Given the complexity of the problem, we need to set up a simulation to gather the necessary data to make a probabilistic model of the robot interaction with the environment—the method marked in blue in Fig. 3.3. A more precise outline of its implementation is depicted in Fig. 3.5, showing how robot simulation provides data to extract the underlying model of the MDP, i.e., the T component of the MDP model. The MDP for our use-case scenario is described in section 3.2.4.

For the simulation task we have used the educational version of V-REP robot simulator (V-REP PRO EDU, version 3.3.2.) [1], integrated with Matlab development environment and a CRUMB toolbox for V-REP [114, 34] to realistically simulate and control the robot. The communication between the V-REP and Matlab is made possible with a Robot Operating System (ROS) API that is available for Matlab. This API is a library of functions that allows the programmer to exchange data with ROS-enabled physical robots, or with robot simulators such as Gazebo [84] or V-REP[92]. We have chosen V-REP because it is an excellent platform for creating and simulating highly realistic robots and it is freely available for educational entities.

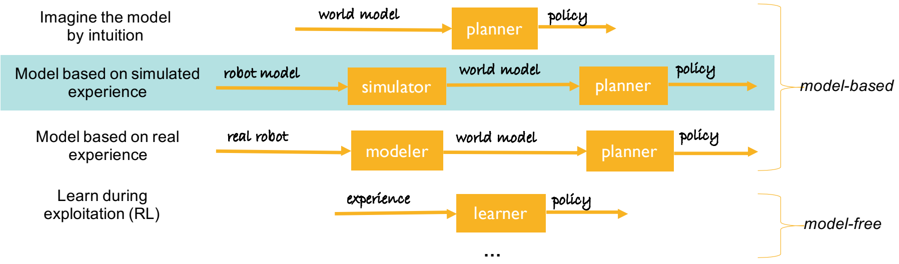
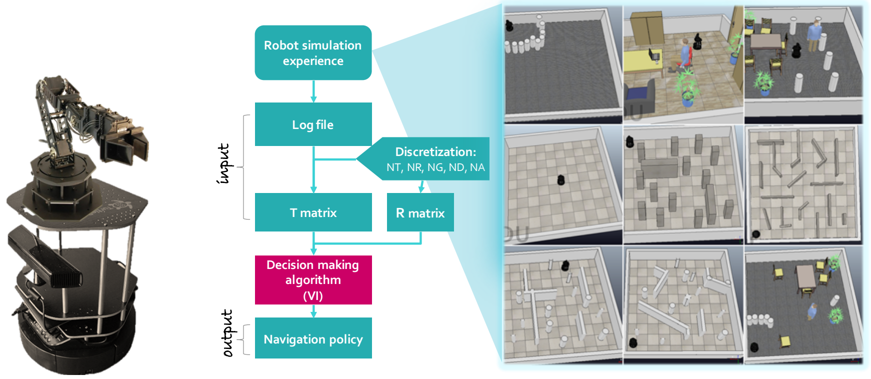


Figure 3.3: Learning the underlying model for the MDP based on simulated experience allows us to obtain the T matrix safely (for the physical robot) and timely. The model resulting from simulation is the input for the VI algorithm (i.e., the *planner*).



(a) CRUMB. (b) Navigation scenarios in V-REP simulator.

Figure 3.4: The CRUMB robot: physical (a) and simulated (b).

In our experiments, we use the CRUMB robot (Cognitive Robotics sUpporting Mobile Base), pictured in Fig. 3.4a [34]. 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.

A CRUMB robot model [114, 34] has been previously designed with V-REP to accurately match the physical description of the real robot. V-REP allows us to simulate the robot behavior in any environment setting when we give it specific commands. This CRUMB model also includes a Hokuyo URG-04LX scanning laser rangefinder, featuring a detectable range from 2 cm to 400 cm, 100 ms/scan and with a 240° scanning range with 0.36° angular resolution. We use the Hokuyo laser sensor for the rangefinder component of the state in the MDP. Both the physical and simulated CRUMB robot can be controlled via an embedded script, a plug-in, a ROS node, a remote API client, etc., and the controllers can be written in C/C++, Python, Java, Lua, Matlab or Octave [92].

To gather the data needed to build the model (T matrix), we have developed a controller in Matlab that works as a remote API client. This controller generates the commands for the simulated robot to run in different scenarios like the ones in Fig. 3.4b, then processes and logs the raw data from the sensors into state-action tuples, as shown in in Fig. 3.5. For the robot controller, we use a programming toolbox developed in collaboration with the CRUMB research team [114]. 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) both on the simulator and the actual robot.

Using this setup, we have recorded the navigation experience from the sim-

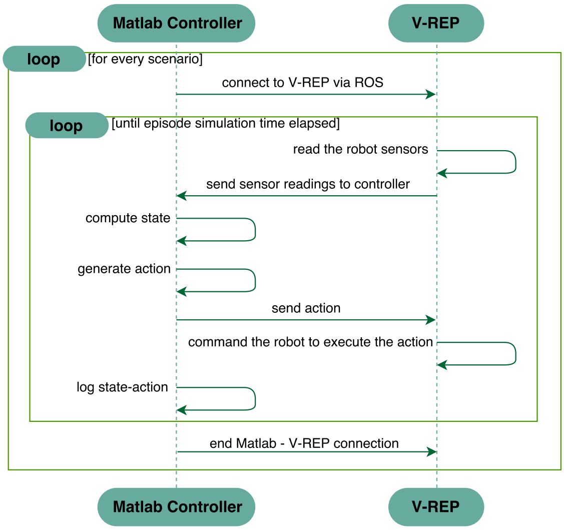


Figure 3.5: Simulation setup used to gather data from the robot interaction with different scenario configurations in the environment. Some of the scenarios used to learn T are shown in Fig. 3.4b.

ulation in a `log` file containing a list of state-action tuples. Based on a chosen discretization and the `log` file data, we build a parametric model of a continuous-space-time stochastic process. The result is a discrete MDP whose T represents the navigation experience from the CRUMB navigation simulation in V-REP. To generate smaller or larger MDP problems (benchmarks) we alter how fine-grained the state discretization is while keeping the number of actions constant. Since these details do not affect our analysis on performance and productivity of VI algorithm implementations, we will not dive in deeper. A detailed report of the simulation and validation of the MDP is available in [26], Sections 3.2 and 3.3.

Once we have the T functions for the navigation benchmarks, we can systematically evaluate different strategies to implement the VI method and solve increasingly large MDP problems. Keep in mind that the computation of T is not part of the VI method (see Fig. 3.4b), as we build it in a previous simulation stage carried out off-line entirely so it does not affect the results on the evaluation of the VI performance.

3.2.2 Solving Large MDPs Optimally on Low-Power SoCs

In the second phase, we first optimize the data structures for solving large-scale MDPs exactly with a sequential implementation of VI. Then we focus on accelerating our sequential VI implementation to compute the optimal policy while minimizing the runtime and power required for it. We made use of three programming models to optimize VI, which we identify as CPU-only, Hybrid-1, and Hybrid-2:

1. CPU-only: optimizations using SPMD parallelism on a multicore CPU, based on OpenMP and TBB [85, 52].
2. Hybrid-1: optimizations using both SPMD parallelism and SIMT parallelism (on a GPU), developed in OpenCL [58] and oneAPI [51]. We accelerate the *evaluate policy* kernel[1] (the more computationally expensive kernel) with the GPU and compute the other kernel (*improve policy*) on the multicore processor(s).
3. Hybrid-2: an extension of Hybrid-1, with the addition that the *evaluate policy* kernel is executed in parallel on both the GPU and the CPU, using two load balancing techniques for improved resources utilization.

---

[1] We have divided the VI algorithm in two kernels, *evaluate policy* kernel and the *improve policy*, as it can bee seen in Fig. 3.8

3.2.3 Performance and Productivity Evaluation

In this final phase, we: 1) explore the computational burden of different MDP problem sizes together with the platforms limits, 2) assess the obtained solutions for the available heterogeneous platforms and evaluate their efficiency and power consumption, and 3) measure the impact on productivity when increasing the abstraction level of the programming model. For the latter, we employ two well known metrics for complexity and ease of programming a code: the *cyclomatic complexity* and the *programming effort* [39, 73].

For the implementation and testing purpose we use four representative heterogeneous mobile platforms ranging from low to medium computing capacity, memory, and power requirements. We choose these platforms because they allow us to evaluate the implementability of small to very large MDPs for different power and computational capacity requirements. We identify them as **TP0**, **TP1**, **TP2**, and **TP3**, whereas TP0, TP1 and TP2 are clear examples of low power computing platforms; TP3 is an example of a higher performance system with medium power requirements and is included here for comparison:

1. **TP0** – An Odroid-XU3 board featuring a Samsung Exynos-5422 CPU (Cortex-A15 and Cortex-A7 big.LITTLE processor), an ARM Mali-T628 GPU (600 MHz, OpenCL 1.1), 2GB of LPDDR3 RAM, and a TDP of 4 to 10 Watts.
2. **TP1** – A 1.60GHz Intel(R) quad core CPU i5-8250U, featuring an UHD 620 integrated GPU with a base frequency of 300 MHz, 8GB of DDR4 and a TDP of 10 to 15 Watts.
3. **TP2** – A 3.10GHz Intel(R) dual core CPU i7-5557U, featuring an Iris integrated GPU (6100) with a base frequency of 300 MHz, 16GB of DDR3 RAM and a TDP of 23 to 28 Watts.
4. **TP3** – A 3.30GHz Intel(R) quad core CPU, i7-5775C, featuring an Iris Pro integrated GPU (6200) with a base frequency of 300 MHz, 32GB of DDR3L RAM, and a TDP of 37 to 65 Watts.

We used two tools to evaluate the performance of our VI implementations on the four platforms. For TP0 we have an in-house library to measure energy consumption [29] on **TP0**. The board features four on-board INA231 current and power monitors to monitor A15 cores, A7 cores, DRAM and GPU power dissipation in real time. The readouts from these power sensors are accessible through the `/sys` file system from user-space. The sample rate used is 10 Hz. This way, one sampled power value is obtained every 100 milliseconds. We have chosen this sample rate because the power sensors actualize their values every 260 milliseconds approximately, so a sample rate two times faster is good enough for getting accurate measurements. The in-house library allows starting/stopping a dedicated thread to sample power readings and integrate them through time using the real-time system clock. A previous study demonstrated that the overhead introduced by this sampling thread was negligible [29]. We rely on the *Processor Counter Monitor* (PCM) library [119] to access the hardware counters on **TP1, TP2** and **TP3**. They allow us to measure the energy consumption (in Joules) on the CPU, GPU and Uncore components for a given application at runtime. In the 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 and L3 cache hits and misses.
- Uncore metrics: read and written bytes from and to memory controllers, data traffic transferred by the Intel QuickPath Interconnect links. This metric is equivalent to the memory energy measurement available for the Odroid platform and is the one that we use as reference.

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 are detected. Previously, we have verified that the measurements are independent.

3.2.4 Mobile Robot Navigation MDP Use-Case

Our robotic problem has one agent, CRUMB, with a builtin low power processor. The robot has to navigate through a structured indoor environment realistically simulated to reach a given metrical target (in any orientation) while avoiding obstacles. From a practical view, the target can be the recharging station, a desk where mail has to be delivered, or maybe a moving person. We define an indoor environment as a five by five square meters space surrounded by walls. The inner space may contain any number of obstacles, placed in any position. From an MDP perspective, the goal of the robot is to find an optimal policy to perform such navigation.

When modeling our use-case as an MDP, as a requirement, the system must have *full observability*. In our case, the robot *state* has to be completely (deterministically) observable, derived directly from the values of the sensor readings.

This is only possible when the sensors are accurate enough. Otherwise, we would have to formalize our problem as a Partially Observable MDP or POMDP. In our case, the sensors of CRUMB can measure with sufficient accuracy the following parts of the state (we use *structured* formulation of a discrete MDP):

- Robot orientation in the universal frame of coordinates or θ angle.
- Measurements of distances to obstacles r, equally spaced in the frontal area of the robot (egocentrically), provided by the Hokuyo URG-04LX laser range finder, g being one such obstacle-distance.
- Distance d to the target.
- Angle a between the robot orientation and the relative location of the target.

All in all, the number of states of the resulting MDP is $|S| = NT \cdot NR \cdot NG \cdot ND \cdot NA$, where S represents the state space and NT, NG, ND, NA the cardinal of the discretized measurements corresponding to θ, g, d, and a, respectively.

As for the *actions*, CRUMB can take any of the following at any iteration of the sequential decision-making process: stay still (a_0), move forward (a_1), move along a curved trajectory to the left (a_2) or the right (a_3), turn around without displacement (a_4), and move backward (a_5). The number of actions is thus $|A| = 6$, where A represents the action space. The actions last for a given fixed time that is long enough to produce all their effects and avoid any non-markovianity caused by the dynamics of the physical system.

The T matrix models the robot-environment interaction and has the form of a 3D sparse matrix stored in the CSR (Compressed Sparse Row) format. Thus, it does not store the values for those state transitions that have a zero or close to zero probability to occur.

The robot receives a positive *reward* $R(s) = y$ when it reaches the target, or possibly when getting close enough to it (attracting state, Eq. 3.1), a negative large negative reward x if it collides with obstacles (repelling states), and usually a small negative or zero reward z in other cases (neutral states). The *reward matrix*, $R(s, a)$ (Eq. 3.2), quantifies the *expected* reward after taking action a while being in state s, reaching state s'. The reward function indirectly defines the task to accomplish or the behavior of the agent, so it must be prescribed carefully.

$$R(s) = \begin{cases} x : x < 0, & \text{if } r(i) = 0 \\ y : y > 0, & \text{if } d = 0 \quad, \forall i \in [1, NR], s \in S, a \in A \ , \\ z : x < z < y, & \text{otherwise} \end{cases} \tag{3.1}$$

$$R(s,a) = \sum_{s' \in S} P(s' \mid s,a) R(s') \tag{3.2}$$

T 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 behavior of the world when the robot acts in it. Since it holds probabilities, T must satisfy the condition defined by Eq. 3.3, where $next(s,a)$ is the set of states that can be reached from state s by executing action a.

$$\sum_{s' \in next(s,a)} T(s,a,s') = 1, \ \forall s,a,s \in S, a \in A \tag{3.3}$$

There are many ways to get the model of the agent interaction with the world for model-based decision making and planning scenarios, as you may see in Fig. 3.1. This include imagining it, using expert knowledge and creating the model based on real experience or simulation. We use the latter (marked in blue in the figure) to get the data to form this T. We have realistically simulated the robot using the physical simulator V-REP [92]. Our MDP is a discrete parametric model of a continuous-space-time stochastic process. To generate smaller or larger MDPs, which is necessary to evaluate our parallelizations of VI for different problem sizes, we alter the state quantization while keeping the number of actions constant. In this case, for MDPs with different levels of discretization, a way to find the optimal quantization is to compare the total estimated reward obtained when exploiting the optimal policy in the long-term, $R\pi$ (see Eq. 3.4). $R\pi$ is computed as the sum of the value of the policy in each state weighted by the marginal probability of reaching that state.

$$R\pi = \sum_{s'} V(s') \sum_{s,a} T(s'|s,a) P(s,a). \quad P(s,a) = \begin{cases} 1 : a = \pi(a) \\ 0 : \text{otherwise} \end{cases} \tag{3.4}$$

Note that this use-case study is not a practical nor complete solution to solve the robot navigation problem under the decision making perspective. Such a solution would involve more sophisticated methods, possibly based on reinforcement learning (RL), that can be built upon VI, but also add supplementary aspects to consider, which are out of the scope of this work. For instance, we provide the robot with a pre-built T matrix (learned in off-line simulation) instead of a progressively estimated one while executing the planning for navigation. RL approaches that make very efficient use of progressive knowledge about the system dynamics can be found, for instance, in asynchronous RL [75].

3.3 Related Work

Markov decision processes are a classic mathematical formalism for modeling sequential decision making under uncertainty in systems where the next state can only be determined stochastically from the current state and the action taken. They have been applied to a diversity of areas, such as operations research, civil engineering, communications engineering, ecology, finance & economics, transportation, and robotics [99, 11]. They can be extended to cope with additional sources of uncertainty, like in POMDPs [61], large and continuous spaces (the basic formulation uses discrete states and actions), and many other issues [118].

There are numerous methods to solve MDPs exactly, i.e., to plan an optimal course of action that may not be unique, known as a *policy*. When planning for MDPs, the norm is using VI (the method used in this study) or PI implementations which are only optimized for faster execution on platforms that have unlimited power supply. Although we have found some related works considering solving MDPs on low-power platforms [96], the power dissipation is not analyzed [110, 96, 55, 121, 54, 122, 46, 50]. Focusing then on the efficient use of computational resources, Zhou *et al.* [126] appear to be the first to develop a customized representation for sparse data to store and access the transition function in a (Bayesian) MDP implementation on a GPU. Their approach gives a 5× reduction in memory utilization (although this reduction is problem dependent) and a proportional decrease in the computations needed to solve the MDP.

The transition function T is a 3D matrix ($|States| \times |Actions| \times |States|$) that can be compressed using a CSR-like format only for the 3rd dimension. This representation makes sense in a navigation scenario because the agent can attempt any action from the current state (i.e., T is dense in width and height), though not all states are reachable by taking any action from the current state (i.e., T is sparse in depth). We do not use GPU-vendor-optimized Sparse BLAS libraries, such as cuSparse or clSparse [83, 41] because they do not support 3D sparse matrices (they support only sparse vectors and 2D matrices). We need a custom made solution that accounts for two facts: (1) VI does not require direct access to particular values in T, and (2) in model-based MDPs T is known apriori. In other words, VI does not require a lookup routine to access values in T, nor an update routine to add new nonzero values to it, as in [126]. We opt for a lightweight representation of T that employs only three array vectors. It allows O(1) data access for its use by VI by orderly grouping the access to the T values by the contents of the (current) state-action cell (see Fig. 3.6). We call it "3D-lite-CSR". Our representation improves memory utilization up to 9× for the evaluated benchmarks.

Besides choosing a suitable representation for the data, there are three main approaches to improve the computational efficiency of solving MDPs: exploiting parallelism on CPU (usually SPMD), exploiting parallelism on GPU (mainly SIMT), and using approximate methods. The first one can be based on multi-thread standard programming APIs such as OpenMP and vectorization strategies [96, 110, 54, 122, 50, 49]. The second one proposes discrete GPUs and, typically, CUDA to speed up both approximate and exact methods [96, 50, 55, 121, 46, 126, 77]; although these GPU-based solutions have been implemented on high-performance HCPs, they do not fully exploit the hardware resources for the VI implementation and are unconcerned about energy efficiency[2]. Some hybrid solutions use both CPUs and GPUs, but not simultaneously [96, 110, 122]. Finally, approximate methods for solving MDPs (our third category) have been proposed and parallelized for multicore and GPU execution in [109, 54].

Ref.	**Method**	**Multicore**	**GPU**	**Heterog.**	**Largest MDP** $\lvert States \rvert$ $\lvert Actions \rvert$	**Total TDP (Watt)**	**Testbeds & TDPs (Watt)**	**Sparse**	**Real Time**	**Energy Eval.**	**Application**
[96]	Value Iteration	✓	✓	✓	65,536 6	10 to 315W	K40c (245W) GTX Titan (250W) Jetson TK1 (10W) i7-x64 Desktop (65W)	✗	✓	✗	Crowd simulation
[110]	Value Iteration	✓	✗	✓	4,800 7	70W	GeForce GT 540M (35W) i7-2620M (35W)	✗	✗	✗	UVS trajectory planning
[55]	Value Iteration	✗	✓	✗	1,048,576 4	180W	GeForce 8800 GTX (155W) Intel Core 2 Duo (25W)	✗	✗	✗	Generic
[121]	Loxicogr. Value Iteration	✗	✓	✗	3608 10	137W	i7-4702HQ (37W) GTX 870M (100W)	✗	✗	✗	Semi autonomous driving
[122]	Value Iteration	✓	✗	✓	2,400,000 6	365 to 415W	Geforce Titan (250W) Kepler GK110 (250-300W) 2x Xeon E5-2670 (115W)	3D CSR	✗	✗	Model checking
[50]	Value Iteration Inifinite Stage	✓	✓	✗	N/A	272W	GTX680 (195W) i7-3770 (77W)	✗	✗	✗	Animat & mountain car probl.
[126]	Bayesian MDP	✗	✓	✗	512 2	334W	i5-4670 (84W) GTX 780 (250W)	2D duplex sparse storage (DSS)	✗	✗	Gene regulatory network control
[49]	Value Iteration	✓	✓	✗	2^{70} 70	680W	2x Xeon E5-2680v2 (115W) 2x Xeon Phi 5100 (225W)	✗	✗	✗	Intrusion Detection Systems
[21]	Policy Iteration	✗	✓	✗	1,000 N/A	N/A	N/A	✗	✗	✗	Generic
This work	Value Iteration	✓	✓	✓ Simultaneous Execution	82,800,000 6	4 to 65W	Odroid XU3 (4-10W) i5-8250U (10-15W) i7-5557U (23-28W) i7-5775C (65W)	3D lite CSR	✓	✓	Autonomous Navigation

Table 3.1: A qualitative review of related works.

Our research group has previously developed techniques that enable the simultaneous execution of the workload of a given software by using both the CPU cores and the integrated GPU on high-performance HCPs to low-power SoCs

---

[2] Some methods based on approximate solutions to MDPs have been applied to manage the energy consumption of CPUs when these CPUs are computing other problems [101], but up to our knowledge there has not been any study reported on the energy consumption of the parallelization of VI itself.

[29, 80]. Elsewhere, we have studied the use of VI for solving discrete MDPs in small educational robots equipped with an embedded microcontroller [70]. Here, we are interested in exploiting the parallelization capabilities of emerging low-power heterogeneous computing platforms to execute the VI method more efficiently.

In this proposal, we do not consider model-free methods, but focus on solving the MDP as optimally as the exactness of T allows it. Moreover, we enhance the previously mentioned parallelism approaches by including heterogeneous scheduling strategies that allow the simultaneous execution in all devices on a HCP (both GPU and CPU cores) while considering load balancing between them. We have picked this line of research because although existing strategies work well for small problems when applied to regular, dense data structures, like arrays and matrices, they do not perform so well with irregular computation and sparse data structures, which are common in real-life MDP applications. We also contribute with characterizing the ease of programming and energy efficiency of our parallel implementations.

It is difficult to make a fair quantitative comparison between this work and others because of the high variety of use-cases, problem dimensionality, and platforms used for evaluation, not to mention our focus on both good energy consumption and speed-up (not only on the latter). Nevertheless, we have compiled in Table 3.1 a qualitative comparison among the methods mentioned above aimed at providing a quick overview of the state-of-the-art in these aspects.

3.4 Solving Large-Scale MDPs Optimally on Mobile Platforms

We use a mobile robot navigation problem as a use-case to study the feasibility of solving exactly large decision making problems on low power SoCs. In this section introduce the data structures and over a dozen parallel alternatives of implementing the VI method to plan for our navigation usecase.

3.4.1 Data Structures – 3D-lite-CSR

The state in our MDP problem is defined and structured by its components, and each component of the state has values determined by the chosen discretization: $[0..NT-1]$ for θ, $[0..ND-1]$ for d, etc. If NS is the total number of states and NX is the number of actions, the matrix of expected rewards $R(s,a)$ contains

$NS \times NX$ real values; although individual rewards are highly sparse, this averaged matrix (through Eq. 3.2) becomes dense. The policy, π (or P), and the policy values, V, are one-dimensional arrays of NS values each (one per state). The former contains integer identifiers of actions in the interval $[0..NX-1]$, while the latter contains real values of discounted rewards. Both of them have to be implemented as dense arrays. Finally, T matrix contains $NS \times NX \times NS$ real values. T is a sparse matrix because when parting from any initial state with any possible action it is very common not to reach every other state. In the evaluated benchmarks, there are approximately one to ten reachable states from any state-action combination, the majority of the states being too far to be reached in one step.

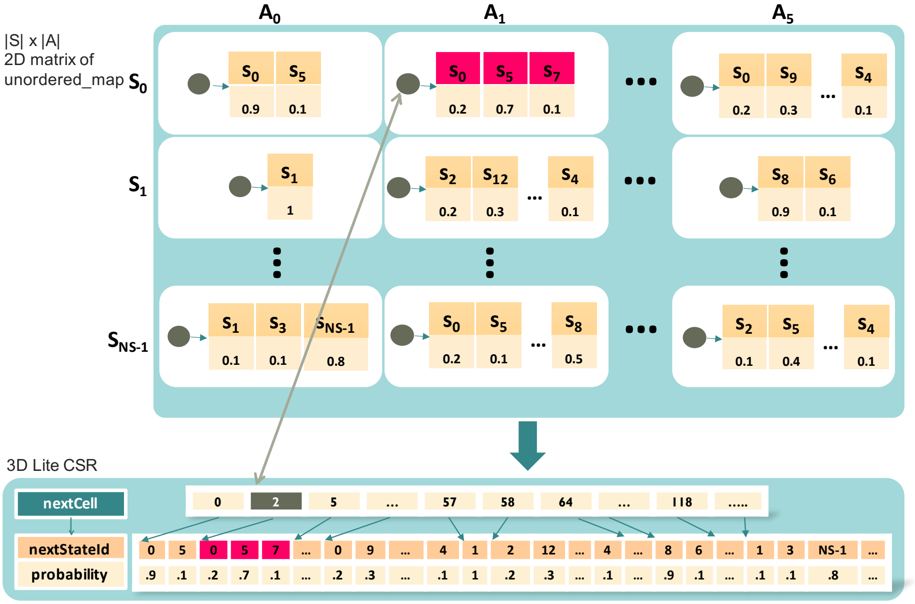


Figure 3.6: Sparse representations of the transition probability matrix, T.

Our HCP platforms are bound in memory and computing power, so 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 its sparsity, for the sequential and CPU-only versions we could use a $NS \times NX$ matrix, each cell containing a map (`unordered_map` in C++); every map would contain in turn a list with pairs `<nextStateId,probability>` for all reachable states from the state-action corresponding to the cell, as shown in the upper part of Fig. 3.6.

Pairs corresponding to a probability of zero are not stored in the map.

Unfortunately, the previously described representation is not appropriate for the OpenCL programming model that we use to code the kernels (and which are offloaded to the GPU in some of the parallelized versions). This is because the OpenCL (v1.1) kernel allows only built-in scalar data types, 1D arrays (pointers to the built-in data-types), and well-aligned structures as arguments. In these implementations, we use instead a representation of T with 1D arrays, as shown in the lower part of Fig. 3.6. This 1D representation is composed of three arrays:

- `probability`: containing the non-zero transition probabilities from the matrix, ordered row-wise (first, all the transition probabilities for S_0, then for S_1, etc.). Its size, S_z, is given by the total number of non-zero transition probabilities for all state-action combinations.
- `nextStateId`: each cell stores the Id of the "next state", with its associated `probability` being held in the corresponding cell (i.e., same position) of the `probability` vector. It also has size S_z.
- `nextCell`: it indicates the starting position of groups of consecutive elements from the `probability` and `nextStateId` arrays that correspond to a given state-action pair. Its size is $NS \times NX + 1$, so that `nextCell`$[NS \times NX] = S_z$.

In our experiments, we have noticed that the implementations using 3D-lite-CSR representation for T matrix outperform the ones based on the `unordered_map` and non-sparse representations in all cases, so in the following sections we review only the findings concerning the implementations using 3D-lite-CSR format. The main advantage of our sparse representation is that it reduces the memory and computation requirements of the algorithm by $\approx 90\%$ while conserving an O(1) data access by orderly grouping the T values by the contents of the state-action cells. This is only possible because of the sparse nature of T.

Studying the typical sparsity of the T matrix, we observe that nonzero transition probabilities occur almost exclusively for neighboring states and are concentrated on the diagonal. This pattern can be observed for small, medium, and large MDP models. For instance, in Figure 3 from [57] you can see the T matrix of a four states MDP for pediatric sepsis; in Figure 4 from [32] the T matrix of a typical small-medium MDP for analyzing human movement trajectories, and in Figure 8 from [31] we can observe a medium T MDP used in fabric-aware 3D capacitance extraction. Generally, the larger the state space, the higher the sparsity. Figure 3.7 gives an intuition on the typical sparsity of T for medium, large and very large MDP problems.

However, 3D-lite-CSR format does have some drawbacks (that we consider

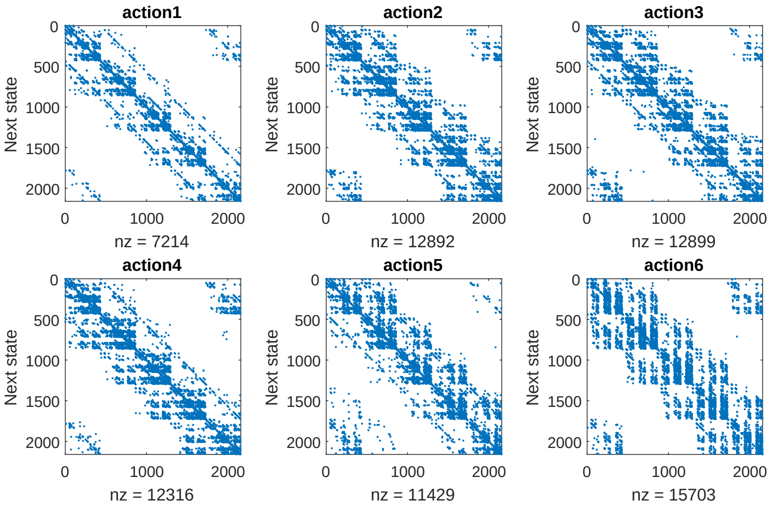


Figure 3.7: Transitions existence from the current state to the next state when attempting every possible action from every state. A blue dot on the graph indicates that there is at least one transition from s1 (current state, X-axis) to s2 (next state, Y-axis) by taking action a, where a takes values from 1 to 6, each corresponding to action stay, move forward, move to the left with displacement, etc. Here, nz indicates the number of nonzero state-action-state triples (number of blue dots). We define the sparsity of T as $sp = (NS \times NS - nz)/(NS \times NS)$.

less important than its benefits): one problem is that it produces memory divergences, a type of irregularity that causes load imbalance in the GPU (the GPU performance benefits most from coalesced memory access). Besides, when computing “Evaluate policy”, the robot can end up in a variable number of states and, as a result, `nextCell[idxState]` points to a varying number of possible state transitions, which produces control divergences, another type of irregularity that causes load imbalance in GPUs.

3.4.2 Taxonomy of VI Implementations

In total, we evaluate thirteen implementations, incrementally optimized for runtime, energy efficiency, and ease of use. The first version is sequential, SEQ, based on the pseudo-code in Fig. 2.1, and serves as a reference for correctness. The second and third variants (CPU-only) are optimized for multicore CPU execution. We call them OMP and TBB, for using the OpenMP and Threading Building

Blocks frameworks, respectively [85, 115]. We use TBB implementation as the baseline for comparing the efficiency of different heterogeneous implementations. In the fourth implementation (Hybrid-1), we program the heaviest kernel of VI with OpenCL, so we call it OCL. It runs VI in parallel on the GPU and on the multicore, but *not* simultaneously.

The nine remaining implementations are all heterogeneous, *simultaneously* exploiting the CPU and the GPU. They are the result of combining three different coding styles for the GPU (OCL, BUFF and USM) and three different heterogeneous schedulers (HO, HD, and HL), that we explain next. Therefore, these implementations are: HO-OCL, HO-BUFF, HO-USM; HD-OCL, HD-BUFF, HD-USM; and HL-OCL, HL-BUFF and HL-USM.

Regarding the three different coding styles, we first have OpenCL, OCL, that represent a low-level programming model in which HW aspects of the GPU are exposed to the user. This implies that the developer has to learn and manage data types as OpenCL `platform`, `device`, `context`, `queue`, `kernel`, etc. and deal with low-level functions to compile the GPU kernel, move data to and from the GPU, pass the kernel arguments, enqueue the GPU kernel, etc. These low level details are hidden if we use oneAPI, that offers two coding models:

- BUFF (SYCL style): explicitly declare `buffers` that encapsulate the data across the CPU and the GPU and `accessors` that give access to the data and define data dependencies.
- USM (Unified Shared Memory): a pointer-based alternative to BUFF, where the data is allocated using `malloc_shared()` and accessed as a regular array from both the CPU and the GPU. This particular data allocation used to be an Intel extension only available in the Intel's DPC++ compiler (part of the 2020 SYCL standard, Feb. 2021 release, when these experiments were performed). However, it has been included in the latest review of the SYCL standard. Using USM requires HW support for Shared Virtual Memory (a.k.a. SVM or unified virtual address space). The clear benefit is that data movement between the CPU and the GPU is avoided, although cache coherency can be a source of overhead.

Finally, our three scheduler implementations (HO, HD, and HL) have been devised to improve the overall heterogeneous performance. They extend the TBB implementation using increasingly complex scheduling strategies for CPU+GPU heterogeneous computing, and are described in Section 3.4.5.2.

3.4.3 Sequential Implementation

Our sequential implementation of VI has two parts: the MDP initialization block which encompasses the creation of the transition function T and the rewards matrix R, and the core computation of the VI Algorithm that produces the optimal policy. In Fig. 3.13, we show the general structure of our VI implementations. First, the parametric MDP is initialized using the navigation experience from the V-REP simulation stored in *Log file* (a CSV-format file containing a list of state-action tuples), the number of actions, *NX*, and the discretization parameters, *NT, NR, NG, ND* and, *NA*. This corresponds to the *MDP Initialization* block of Fig. 3.8. Next, given an MDP model, the *Value Iteration Algorithm* executes in a loop until an *Optimal policy* is produced.

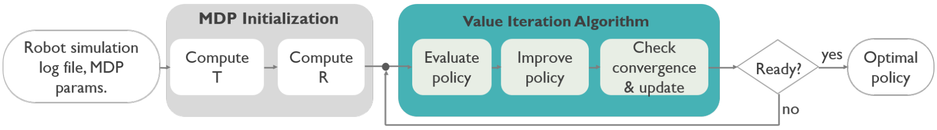


Figure 3.8: VI control flow graph.

The sequential version is a plain C++ implementation of the VI algorithm (Figs. 2.1 and 3.8) and serves as a reference for the correctness and relative improvement of further optimized versions. From now on, we focus only on the VI Algorithm.

The VI Algorithm consists of three kernels: “Evaluate policy”, “Improve policy” and “Check convergence & update”. “Evaluate policy” and “Improve policy” kernels solve the Bellman equation. In particular, the “Evaluate policy” kernel computes the expected value of each subsequent state: $V_k[s] = \sum_{s'} P(s' \mid s,a)(R(s,a)+\gamma V_{k-1}[s'])$, for the current state s (see line 8 in Fig. 3.10 for details). This kernel is the most computationally intensive with a complexity of $O(kMN^2)$. The k term represents the number of iterations required by VI to converge to an optimal policy; N and M are the number of states and actions of the MDP, respectively. Next, the “Improve policy” kernel searches for the action with highest value for each state and uses it to update the current policy: $\pi[s] = argmax_a \sum_{s'} P(s' \mid s,a)(R(s,a)+\gamma V_k[s'])$ (see line 12 in Fig. 3.10). This kernel has a complexity of $O(kMN)$. Finally, the “Check convergence & update” kernel has a complexity of $O(N)$.

The three kernels have sequential dependencies; therefore, it is necessary to execute them one after the other. We have identified “Evaluate policy” kernel –K1– as the most computationally intensive, followed by the “Improve policy” kernel –K2– (the first two blocks of Fig. 3.13). “Evaluate policy” kernel uses up to 80% of the computing resources, so in this work we focus our attention on optimizing the execution of this kernel. Due to the main data structure that is traversed by this kernel, T, a 3D sparse matrix stored in our 3D-lite-CSR format, two challenges arise: i) the memory access pattern is not coalescent; and ii) the computational load of each iteration of the parallel iteration space is different. Therefore, the CPU and GPU threads unavoidably have unbalanced workloads to process regardless of the workload distribution strategy.

3.4.4 CPU-Only Implementations

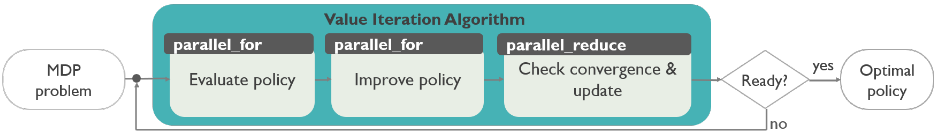


Figure 3.9: CPU-only: OpenMP and TBB VI implementations.

The first and simplest parallel strategy that we can use attempts to takes advantage of the all the CPU resources by using parallel threads in the multicore processor(s). We present two CPU-only implementations, based on OpenMP and TBB respectively (Fig. 3.9). Both OpenMP and TBB offer task-based frameworks based on lightweight runtimes that enable efficient implementations of parallel applications; however, the data accessed in the “Evaluate policy” kernel is irregular due to the sparse nature of the T matrix. We want to assess the efficiency of these two different frameworks for handling the load imbalance due to the irregular data accesses in the “Evaluate policy” kernel, as well as their behavior with the “Improve policy” kernel where data accesses are regular. One of our test platform’s CPU has a big.LITTLE architecture, which is quite common in low-power heterogeneous computing, adding further complexity to the issue of load balancing.

OpenMP provides a set of compiler directives and environment variables that allow shared memory parallel programming. It supports *dynamic* and *guided* scheduling strategies for orchestrating imbalanced workloads. They both use an internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread has finished, it retrieves the next block of loop iterations from the top of the work queue. The difference between the two is that the first one uses a fixed chunk size, while the second starts with a large chunk size that is decreased to handle better the load imbalance between the threads for the remaining iterations. This kind of scheduling involves additional overhead;

```
#include <omp.h>
int main(int argc, char **argv) {
  while(notReady) { // Value iteration algorithm
    // Evaluate policy (OMP)
    #pragma omp parallel for schedule(guided)
    for (s = 0; s < NS; s++) //for each state
        for (a = 0; a < NA; a++) // for each action
            $V_k[s] = \sum_{s'} P(s' \mid s,a)(R(s,a) + \gamma V_{k-1}[s'])$
    // Improve policy (OMP)
    # pragma omp parallel for schedule(dynamic)
    for (s = 0; s < NS; s++) //for each state
        $\pi[s] = argmax_a \sum_{s'} P(s' \mid s,a)(R(s,a) + \gamma V_k[s'])$
    // Check convergence condition and update (OMP)
    #pragma omp parallel for reduction(+:norm2PolicyValue)
    for (s = 0; s < NS; s++) {...}
    if $\forall s, |V_k[s] - V_{k-1}[s]| < precisionThreshold$ notReady = false
  }
```

Figure 3.10: OpenMP implementation of VI when selecting the *guided* scheduling strategy.

```
#include <tbb/tbb.h>
using namespace tbb;
int main(int argc, char **argv) {
  while(notReady) { // Value iteration algorithm
      // Evaluate & Improve policy (TBB)
      parallel_for(blocked_range<size_t>(0,NS), evaluatePolicy);
      parallel_for(blocked_range<size_t>(0,NS), improvePolicy);
      // Check convergence condition and update (TBB)
      parallel_reduce(blocked_range<size_t>(0,NS), norm2PolicyValue);
      ...
```

Figure 3.11: TBB implementation of VI.

thus it may not be advantageous for the “Improve policy” kernel (regular data accesses).

TBB is a template library for shared memory parallel programming that provides several template functions, among them, functions for performing parallel execution of a loop over a range of iterations. The default partitioner of these functions recursively performs binary splitting of the range of iterations into chunks, until a minimum threshold size is reached. Each chunk is then run as an independent task; the internal TBB runtime scheduler employs a work-stealing technique to balance the load of tasks across all CPU cores. Work stealing usually works better than dynamic or guided approaches in cases of imbalanced workloads. For instance, the library provides `parallel_for()` and `parallel_reduce()` function templates, which are functionally equivalent to the OpenMP `#pragma omp parallel for` and `omp parallel for reduction` directives used in our implementations.

Fig. 3.10 (lines 5, 10, and 14) and Fig. 3.11 (lines 6, 7, and 9) illustrate the use of the OMP directives and TBB functions for our CPU-only implementations in order to parallelise the "Evaluate policy" and "Improve policy" kernels, as well as the convergence check. The `operator()` for functor `evaluatePolicy` (functor `improvePolicy`) in line 6 (line 7) in Fig. 3.11 is equivalent to lines 6-8 (11-12) in Fig. 3.10.

An outline of this implementation is presented in Fig. 3.11. The scheduler of TBB uses an internal queue to give to each thread a chunk-sized block of the loop iterations. When a thread has finished its chunk, it retrieves the next one from the top of the work queue. The scheduler partitions the work in large chunks in the beginning and then it decreases the chunk sizes to better handle the load imbalance among the threads for the remaining iterations. Fig. 3.11 (lines 6, 7, and 9) illustrates in C++ style the use of TBB template functions for our multicore implementation to parallelize the "Evaluate policy" and "Improve policy" kernels, as well as the convergence check.

3.4.5 CPU+GPU Heterogeneous Implementations

The VI algorithm does not allow concurrent 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 one after the other. We propose here different heterogeneous implementations to accelerate the "Evaluate policy" kernel, which is the most computationally expensive part of the algorithm as stated before. These heterogeneous implementations will use the CPU and the GPU to collaborate in the execution of the iterations of the kernel parallel loop.

One of the main challenges of CPU+GPU heterogeneous implementations is how to partition and schedule the work among the devices that collaborate in the computation to avoid load imbalance between them. Another challenge is related to the sources of irregularity (memory and control divergencies) associated to the sparse matrix representation of T that affect the work offloaded to the GPU (load imbalance among the GPU threads, as explained in section 3.4.1). To study the impact that different work partitioning and scheduling strategies have in this highly irregular kernel, we study three implementations: -HO-, -HD-, and -HL-. In all of them, we use TBB as orchestrator to offload chunks (blocks) of parallel iterations to the GPU and to the CPU cores.

3.4.5.1 Hybrid-1: Exploring Functional CPU-GPU execution

```
 1 #include <tbb/tbb.h>
 2 using namespace tbb;
 3 int main(int argc, char **argv) {
 4   while(notReady) { // Value iteration algorithm
 5       // Evaluate policy (no scheduling: OpenCL vs oneAPI(BUFF or USM) on GPU)
 6       gpu_parallel_for(blocked_range<size_t>(0,NS),evaluatePolicy);
 7       // Improve policy (TBB-CPU)
 8       parallel_for(blocked_range<size_t>(0,NS), improvePolicy);
 9       // Check convergence condition and update (TBB-CPU)
10       parallel_reduce(blocked_range<size_t>(0,NS), norm2PolicyValue);
11       ...
12   }
13 }
```

Figure 3.12: Hybrid-1 implementation of VI.

In a first approximation, named OCL, we offload all the parallel loop iterations of the "Evaluate policy" kernel (Fig. 3.12, line 6) on the GPU using OpenCL and oneAPI, while the other kernels execute on the CPU multicore using TBB (Fig. 3.12, lines 8-10). In this implementation, we apply a form of functional parallelism in which each device computes different kernels at a given time. The problem with this implementation is that the CPU cores are idle while the GPU is computing the "Evaluate policy" kernel, and likewise, the GPU is idle when the CPU cores are computing the other kernels. So the resource utilization is not optimal.

The goal of our next implementations is to incorporate the CPU cores to collaborate simultaneously with the GPU in the computation of the "Evaluate policy" kernel to improve resource utilization.

3.4.5.2 Hybrid-2: Exploring Heterogeneous Scheduling

Here we explore three heterogeneous scheduling strategies that allow the simultaneous execution of the parallel iterations of the "Evaluate policy" kernel on the GPU and CPU cores. The three strategies are results of previous research of our group. In [80], we propose Oracle and LogFit, two scheduling strategies that enable simultaneous execution and efficient use of resources on CPU+GPU platforms. Also, in [93], we study a CPU+FPGA scheduler called Dynamic. These three scheduling strategies have been implemented in a library that extends the functionality of the Intel TBB function *parallel_for* to *heterogeneous_parallel_for* by including the possibility to simultaneously orchestrate the work between the

CPU cores and the GPU.

In these previous works [80, 93], the CPU code is implemented with TBB and the accelerator code with OpenCL. Here we re-implement the schedulers on top of oneAPI to ease the development of the GPU kernels. In Sec. 3.5, we discuss the benefits and costs of using the oneAPI programming model with respect to the initial implementation, but for now, let us see how these three schedulers work under the hood.

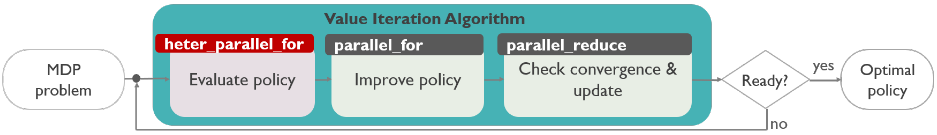


Figure 3.13: Control flow graph for heterogeneous implementation of VI.

Oracle scheduler (HO): makes a static, one-time, partition of iterations between the CPU and the GPU. HO divides the workload between them using a *RatioGPU* parameter in [0%..100%] to indicate the ratio of the iterations that goes to the GPU. This ratio is set as an input parameter to the scheduler (see lines 9 and 13 in Fig. 3.15). The scheduler sends the remaining iterations to 100% to the multicore. To obtain the optimal work balance between the two devices for HO, one should assess every possible partition. For a good approximation of the optimal work division, we have trained the scheduler by executing HO for all work-ratios from 0% to 100% with an increment of 10%. We show two examples of how the optimal GPU chunk size for HO is selected on two heterogeneous platforms, Kaby-Lake and Broadwell-Desktop, in Fig. 3.23 (upper-side graphs).

Dynamic scheduler (HD): works as a dynamic scheduling approach (similar to OpenMP). So the programmer has to set a GPU chunk size, *ChunkGPU*, which is passed as an argument to the scheduler (see lines 10 and 14 in Fig. 3.15). HD measures the time that the GPU needs to compute a chunk, and estimates another chunk size for the CPU cores that adapts to that time by using a heuristic. This heuristic aims to adaptively set the chunk size for a CPU core by ensuring that it is proportional to the ratio of GPU/CPU-core throughputs (see [93] for details). All devices are dynamically offloaded with their correspondingly sized chunk of iterations until the iterations have been executed.

For HD scheduler, we see the execution of a `heterogeneous_parallel_for` loop as a sequence of scheduling intervals $\{I_{G_0}, I_{G_1}..., I_{G_i}, ...\}$ for the GPU, and $\{I_{C_0}, I_{C_1}..., I_{C_i}, ...\}$ for each CPU core. At the *ith* interval, each computing device computes a chunk of iterations of size $Chunk(I_{G_i}) = ChunkGPU$ (GPU Chunk size) and $Chunk(I_{C_i})$ (the estimated CPU chunk size for a CPU core), respectively. The execution time for the assigned GPU chunk, $T(I_{G_i})$, or CPU chunk, $T(I_{C_i})$, is measured in the interval; this time is used to compute the throughput in the corresponding interval, $\lambda(I_{G_i}) = ChunkGPU/T(I_{G_i})$ for the GPU or $\lambda(I_{C_i}) = Chunk(I_{C_i})/T(I_{C_i})$ for a CPU core. We keep monitoring the throughput to adapt and estimate the chunk size for a CPU core in the next interval as, $Chunk(I_{C_{i+1}}) = ChunkGPU \cdot \frac{\lambda(I_{C_i})}{\lambda(I_{G_i})}$. This way, we ensure optimal resource utilization during each scheduling interval and avoid load imbalance [80]. We show two examples of how the optimal GPU chunk size for HD is selected on two heterogeneous platforms, Kaby Lake and Broadwell-Desktop, in Fig. 3.23 (lower-side graphs).

LogFit scheduler (HL): in contrast to HO and HD, which need offline training for optimal partitioning of the workload, HL is specially designed for irregular applications on heterogeneous CPU+GPU chips. It has an adaptive partitioning strategy that computes the near-optimal chunk size at runtime, both for CPU and GPU, without user intervention nor previous training. The CPU cores and GPU run at their own pace, while HL adaptively offloads chunks of the remaining iteration to each device so that the overall throughput is maximized. HL computes the CPU chunk in the same way as HD.

For computing the GPU chunk, HL uses a log fitting heuristic. This heuristic is composed of an *Exploration Phase* (EP), a *Stable Phase* (SP), and a *Final Phase* (FP). The EP initializes the GPU chunk to the number of Execution Units of the GPU. Next, the EP proceeds in three iterative steps: (1) the GPU chunk size is offloaded to the GPU, (2) the corresponding GPU throughput is measured (and the GPU chunk size recorded —$ChGPU$), and if this throughput improves more than 1% the throughput of the previous GPU chunk, then (3) the GPU chunk is duplicated and going back to step (1). Otherwise, the scheduler transitions to the SP. The SP also proceeds in three iterative steps: (1) it fits a logarithmic curve through the GPU throughput of the previously recorded chunk sizes ($ChGPU$), $a \cdot ln(ChGPU) + b$, and compute its elbow, i.e., the point with maximum curvature. The elbow point of the logarithmic curve allows us to determine a value for $ChGPU$ that is going to be the next optimal GPU chunk size. Then (2) the new GPU chunk size is offloaded to the GPU and (3) the corresponding GPU throughput is measured and recorded. The SP repeats steps from (1) to (3) until the remaining iterations are fewer than the GPU chunk size computed by step (1). In this case, the scheduler transitions to the FP. In FP, if there are sufficient remaining iterations, the scheduler splits them once between the two devices so that they finish the execution at the same time. Otherwise, it sends remaining iterations either to the multicore or GPU (more details in [80]).

One advantage of HD and HL is that they adapt better than a static scheduler as HO for irregular applications. This way, HD and HL ensure near-optimal resource utilization during each scheduling interval and avoid load imbalance [79], but at a cost. They introduce additional overheads (especially HL) due to the reiterative calls to the scheduler during the partition of the parallel loop and the costs of the fitting operation.

```
class ValueIteration {
public:
  //Serial version of the code for a CPU thread (TBB)
  void OperatorCPU(int begin, int end) {
    // Evaluate policy
    for (idxCell=begin; idxCell!=end; idxCell++) {...}
  }
  void OperatorGPU(int begin, int end) {
    // Set GPU (OpenCL) kernel arguments
    setKernelArg(kernel, 0, sizeof(cl_mem), &d_probability);
    setKernelArg(kernel, 1, sizeof(cl_mem), &d_nextCell);
    setKernelArg(kernel, 2, sizeof(cl_mem), &d_nextStateId);
    ...
    setKernelArg(kernel, 8, sizeof(int), &begin);
    setKernelArg(kernel, 9, sizeof(int), &end);
    //Launch GPU kernel
    clEnqueueNDRangeKernel(command_queue,evaluatePolicyKernel,...);
  }
};
```

Figure 3.14: ValueIteration class.

3.4.5.3 Programming Interface

From the programmer's perspective, the implementation of the VI algorithm for execution on a CPU+GPU platform implies calling our heterogeneous `parallel_for` template function. This function receives three input arguments: first iteration, last iteration and an object of a class that implements the `operatorCPU()` and `operatorGPU()` member functions. These two functions implement how a block of iterations are processed on the CPU and on the GPU, respectively.

In Fig. 3.15, we exemplify the initialization of the three schedulers (lines 5-15), which allows the simultaneous execution of the parallel iterations of the "Evaluate policy" kernel, both on the GPU and the multicore. The heterogeneous kernel is launched using the `heterogeneous_parallel_for` function, line 19) which receives as input the range of iterations that will be executed ($begin = 0$, and $end = NS \times NX$) and an instance of a functor class implementing the "Evaluate policy" kernel. We have implemented three variants for functor classes: one that

```
 1 int main(int argc, char **argv) {
 2   // ViBodyOCL vib;
 3   // ViBodyBUFF vib;
 4   ViBodyUSM vib;
 5   // Scheduler params
 6   Params p;
 7   p.numcpus = numCPUCores;
 8   p.numgpus = numGPUs;
 9   p.ratioGPU = RatioGPU; // Used in HO - Oracle
10   p.chunkGPU = ChunkGPU; // Used in HD - Dynamic
11   ...
12   // Heterogeneous scheduler (hs) HO, HD or HL
13   // Oracle* hs = Oracle::getInstance(&p);
14   // Dynamic* hs = Dynamic::getInstance(&p);
15   LogFit* hs = LogFit::getInstance(&p);
16   startTimeAndEnergy();
17   while(notReady) { // Value iteration algorithm
18     // Evaluate policy (Heterogeneous scheduling: CPU+GPU)
19     hs->heterogeneous_parallel_for(0, NS*NX, &vib);
20     // Improve policy, Check convergence & update (TBB-CPU)
21     ...
22   }
23   endTimeAndEnergy();
24   saveResultsForBenchmark();
25 }
```

Figure 3.15: Heterogeneous implementations using the Oracle/Dynamic/LogFit Schedulers and OpenCL/oneAPI programming for the heterogeneous kernel.

uses OpenCL for the heterogeneous kernel code (line 2), and two that use oneAPI for it (lines 3-4)—we give more details on their implementation in Figs. 3.16 and 3.17.

We explain the code snippets from Figs. 3.16 and 3.17 in parallel, as they are closely related. The first is the functor class used in the *-USM implementations, while the second is its *-BUFF counterpart. Their role is to define how shared memory objects are stored and accessed from the host and the device (see `allocateMemoryObjects` method in Fig. 3.16: lines 9-11 and in Fig. 3.17: lines 3-6) and to send work to the GPU and CPU (see `operatorGPU` and `operatorCPU` methods).

In particular, `ViBodyUSM` uses the Unified Shared Memory, or USM, feature of DPC++ to allocate and automatically manage data transfers and synchronization between the GPU and CPU. For our kernel, we need four arrays to represent the MDP model—`probability`, `nextCell`, `nextState`, and `R`—and two more to store intermediary results while computing an optimal policy with Value Iteration—`Q`, and `V`. We represent them all as `X` for brevity (line 10 in `ViBodyUSM` and lines 4, 5, and 9 in `ViBodyBUFF`). All of them are to be accessed for reading from both the CPU and GPU, and `Q` for writing. USM offers three types of allocation: `malloc_device` (can be accessed by the GPU), `malloc_host` (can be accessed by the host CPU and any other device), and `malloc_shared` (like `malloc_host`, additionally, it can migrate to/from the CPU and GPU). The

```
#include "CL/sycl.hpp"
using namespace cl::sycl;
queue q_cpu(cpu_selector{});
queue q_gpu(gpu_selector{});
auto ctx = q_gpu.get_context();
auto dev = q_gpu.get_device();
// [ DPC++ with USM ] functor class for heterogeneous_parallel_for
class ViBodyUSM {
  void allocateMemoryObjects { //6 memory allocations
    type* X = (type*) malloc_shared(sizeX,dev,ctx); //X = R\Q\V\probability\nextCell\nextState
  }
  void operatorGPU(size_t begin, size_t end, event& e){ // send work to GPU
    e = q_gpu.submit([&](handler& cgh) { // 3 LOC.
      PolicyEvaluationF kernel{begin,nextCell,probability,V,nextState,R,Q};
      cgh.parallel_for(range<1>(end-begin), kernel); });
  }
  void operatorCPU(size_t begin, size_t end, event& e){} // same as operatorGPU(), uses q_cpu
  ...
}
```

Figure 3.16: Pseudo-C++ code of the functor class `ViBodyUSM` required by the heterogeneous schedulers to execute the Policy Evaluation kernel of VI. `ViBodyUSM` is implemented with oneAPI & the USM feature of DPC++.

```
// [ SYCL buffers & accessors ] functor class for heterogeneous_parallel_for
class ViBodyBUFF {
  void allocateMemoryObjects { // 6 mallocs + 6 buffers
    type* X = (type*) malloc(sizeX); // X = R\Q\V\probability\nextCell\nextState
    buffer<type, 1> buf_X(X, range<1>(sizeX)); // X = R\Q\V\probability\nextCell\nextState
  }
  void operatorGPU(size_t begin, size_t end, event& e) { // send work to GPU
    e = q_gpu.submit([&](handler& cgh) { // 6 accessors
      auto a_X = b_X.get_access<read>(cgh); // X = R\V\probability\nextCell\nextState
      auto a_Q = b_Q.get_access<discard_write>(cgh);
      PolicyEvaluationF kernel{a_begin,a_nextCell,a_probability,a_V,a_nextState,a_R,a_Q};
      cgh.parallel_for(range<1>(end-begin), kernel); });
    }
  void operatorCPU(size_t begin, size_t end, event& e){} // same as operatorGPU(), uses q_cpu
  ...
}
```

Figure 3.17: Pseudo-C++ code of the functor class `ViBodyBUFF` required by the heterogeneous schedulers to execute the Policy Evaluation kernel of VI. It uses SYCL style buffers & accessors.

simplest way to meet our requirements is to use `malloc_shared` for all of them (see line 10, Fig. 3.16). As you see, the USM allocation types have a similar syntax to the standard C/C++ `malloc`. The difference is that they receive one or two extra arguments: the context (`ctx`, defined in line 5, Fig. 3.16), and additionally, for device and shared type, the device (`dev`, defined in line 6, Fig. 3.16). The memory objects allocated like this can be accessed as regular pointers in the kernel code of the CPU and GPU.

`ViBodyBUFF` uses the Buffer abstraction of the SYCL standard for data management. Our six arrays are allocated with `malloc` and later encapsulated in six buffers (Fig. 3.17, lines 4-5 where again we use `X` to reduce the number of lines in the pseudo-code). Buffer's data can be accessed from the CPU or GPU via accessors, which inform the runtime about the access type (e.g., read, write) and about the device that is actually accessing the buffer. For instance, inside `operatorGPU`, in lines 9 and 10, the helper function member `get_access` initialize the GPU accessors: `a_X` (for `X` in {`R`, `V`, `probability`, `nextCell` and `nextState`}) for reading in line 9, and `Q` for writing in line 10. We need to do the same in order to get access for the CPU in `operatorCPU`. We use `sycl::access::mode::discard_write`, abbreviated as `discard_write` (line 10) for `Q`, to point out that the GPU does not need an initialized copy of `Q` because it will be completely rewritten. The other 5 accessors are initialized with the `read` template argument (line 9). There are other access modes available for buffers that we do no use: `write, read_write, discard_read_write`, and `atomic`.

Once we have configured how data is managed for our kernel, we can send work to the GPU (or CPU). All work requests are done via queues. A queue attaches to a single device (e.g., CPU, GPU, Host, FPGA) and accepts work as a submission (line 13 for `ViBodyUSM` and 8 for `ViBodyBUFF`). We use the SYCL queues `q_cpu` and `q_gpu` to `submit` code to the GPU and CPU for execution. Inside the `submit` call we construct a kernel object (line 14 for `ViBodyUSM` and 11 for `ViBodyBUFF`) passing the data pointers (USM) or accessors (BUFF) to the constructor. With this we can invoke the SYCL `parallel_for` member function of the queue handler `cgh` that will run the code on the device. This method can only be called at command-group scope (`cgh` stands for command-group handler). Inside the kernel, data is accessed via pointers for the USM case or via accessors for the BUFF one.

Note that all oneAPI programs must include the "`CL/sycl.hpp`" header (Fig. 3.16 line 1), and to avoid wordiness, we use the `cl::sycl` namespace for both snippets (Fig. 3.16 line 2).

Next, we evaluate the performance of these three schedulers for our application in a low-power heterogeneous platform.

3.5 Evaluation and Experimental Results

In this section we will:

1. Present the the experimental setup, including the low-power platform used to carry out the experiments [Subsection 3.5.1].
2. Explore the computational limits of the platforms for different MDP problem sizes for the mobile robot navigation case study introduced in Section 3.2.4.
3. Discuss the impact on productivity of the different programming models used in the heterogeneous implementations: OpenCL vs. oneAPI with Unified Shared Memory (USM) and oneAPI with SYCL Buffers (BUFF) [Subsection 3.5.3].
4. Analyze the impact on the efficiency of the three heterogeneous programming models, as well as the impact of the scheduling strategies presented previously: static (Oracle, HO) vs. dynamic (Dynamic, HD) and adaptive (LogFit, HL) [Subsection 3.5.4].

3.5.1 Experimental Setup

We need to test different strategies for implementing VI for a particular MDP problem in engineering: the reactive navigation of an indoor mobile robot, which requires taking motion actions sequentially (with uncertain outcomes) towards a target while avoiding obstacles. Mobile autonomous robots have limited resources in terms of processing capabilities, memory and energy availability. Despite this, the use of diverse variants of MDPs and their algorithms has been frequent in this area for more than a decade [112] (mostly through approximate solutions).

For implementation and testing purposes, we used four representative HCPs, ranging from low to medium computing, memory, and power capacity. Relevant information about their vendor name, CPU, accelerator GPU, RAM, Thermal Design Power (TDP - power consumption under maximum theoretical load) and configuration can be consulted in Table 3.2. We identify the four platforms as TP0, TP1, TP2 and TP3. These devices are convenient test-beds for our experiments, as they allow us to evaluate the implementability of small to large-scale MDPs for diverse power and computational requirements.

Platform	CPU	Integrated GPU	RAM	TDP	Configuration
TP0 Odroid XU3	big.Little: - Cortex-A15 4C @2.1GHz - Cortex-A7 4C @1.4GHz	ARM Mali-T628 6 EUs @600 MHz	2GB LPDDR3	**4 -10W**	Ubuntu 14.04 LTS gcc 6.2.0 C/C++ OpenCL 1.1
TP1 Kaby Lake Refresh	i5-8250U 4C@1.60GHz	UHD 620 24 EUs @300MHz	8GB DDR4	**10-15W**	Ubuntu 16.04 LTS gcc 6.2.0 C/C++ OpenCL 2.1
TP2 Broadwell Mobile	i7-5557U 2C @3.10GHz	Iris 6100 48 EUs @300MHz	16GB DDR3	**23-28W**	MacOS High Sierra gcc 4.2.1 C/C++ OpenCL 1.2
TP3 Broadwell Desktop	i7-5775C 4C@3.30GHz	Iris Pro 6200 48 EUs @300MHz	32GB DDR3L	**37-65 W**	Ubuntu 18.04 LTS gcc 6.2.0 C/C++ OpenCL 2.1

Table 3.2: Description of low-power HCPs used for testing and evaluating our implementations.

In this research, it is interesting to compare Broadwell-Mobile (TP2) and Broadwell-Desktop (TP3) because they use very similar technology and have almost identical integrated GPUs. However, the Broadwell-Desktop platform has double the RAM, and memory is a limiting factor in solving large-scale MDPs. Although Broadwell-Desktop is more in the mid-power range, an energy efficiency tradeoff could be required if execution time is the application's bottleneck. This platform has twice as many cores as Broadwell-Mobile that can speed up computation. For the opposite use-case scenario, when the MDP model is very small, and the application is energy-bound, the ARM Odroid platform (TP0) or another computing platform like Jetson Nano, Jetson TX1, Jetson TX2. For the use-cases in between, with moderate memory, compute power, and relatively low TDP, we have Kaby Lake (TP1) and Jetson Xavier.

Finally, we do not explore platforms like Jetson Nano, Jetson TX1, Jetson TX2, or Jetson Xavier because at the time this work is being done, they do not have official support for portable and vendor-independent programming models with open standards such as OpenCL[3,4,5], nor SYCL, or oneAPI, as they use OpenCL as a backend. On the other hand, Codeplay[6] is working on offering support for DPC++ and SYCL for NVIDIA GPUs using CUDA and OpenCL backends. However, the drivers are still experimental, and currently, only a few discrete GPUs, such as the NVIDIA A100 GPUs, can run sycl code. We do

---

[3] https://forums.developer.nvidia.com/t/opencl-support/74071

[4] https://forums.developer.nvidia.com/t/opencl-support/74071

[5] https://forums.developer.nvidia.com/t/can-the-xavier-run-opencl-applications/70262

[6] https://www.codeplay.com/

not consider them because they are not in the low-medium power TDP range (their TDP ranges from 300W to 400W). Currently, a wider range of NVIDIA GPUs support the latest version of OpenCL 3.0 (please see the NVIDIA release note[7] on supported desktop and notebook GPUs) and are likely support open parallel programming models in the future, which would allow us to compare contemporary and competing platforms.

As mentioned in the methods section, we use two libraries to to monitor the *runtime* (in seconds) and the *energy consumption* (in Joules) on the CPU, GPU and Uncore components for a given application. We employ a custom-made library for ODROID-XU3 (TP0) and the PCM library for the Intel mobile (TP1, TP2) and desktop (TP3) platforms.

For performance evaluation, we have launched our VI implementations for different input sizes, and measured the execution time and energy consumption by calling the `startTimeAndEnergy()` method before the execution of the VI while loop, and the `endTimeAndEnergy()` and `saveResultsForBench()` methods afterwards, like in Fig. 3.15, lines 16, 23, and 24. Time and energy consumption measurements reported in this section have been computed using the average from fifteen executions.

3.5.2 Exploring the Limits of Mobile Platforms for Different MDP Sizes

For this study, we have generated a number of MDP benchmarks for autonomous navigation by sampling the MDP model. In particular, we tackle with the discretization parameters NT, NG, and ND of the MDP, and implicitly, with the navigation precision. The resulting MDPs are exponentially increasing (by a factor of two approximately) both in the number of states of the MDP and representation size in MB.

More precisely, our implementations of the VI algorithm have been evaluated for fourteen MDP sizes, named from IN0 to IN13, listed in Table 3.3. The goal of this first study is to explore the computational limits of the HCPs and assess the maximum solvable MDP problem size for each platform. For each benchmark, we provide its ID, its size on memory in Megabytes (Size), the corresponding number of MDP states (NStates) and an indication of whether the MDP is solvable (✓) or not (✗) on the test platforms (TP0, TP1, TP2, and TP3).

---

[7] https://us.download.nvidia.com/Windows/465.89/465.89-win10-win8-win7-release-notes.pdf

MDP-ID	Size (MB)	NStates	TP0	TP1	TP2	TP3	$R\pi$
IN0	1.1	2,160	✓	✓	✓	✓	238,587.46
IN1	2.0	5,400	✓	✓	✓	✓	302,509.90
IN2	3.9	17,496	✓	✓	✓	✓	563,574.68
IN3	7.9	50,544	✓	✓	✓	✓	602,792.93
IN4	16.0	131,625	✓	✓	✓	✓	1,013,505.31
IN5	32.0	295,750	✓	✓	✓	✓	1,148,107.50
IN6	64.4	631,750	✓	✓	✓	✓	1,316,330.75
IN7	127.9	1,296,000	✓	✓	✓	✓	1,321,889.87
IN8	255.4	2,628,288	✓	✓	✓	✓	1,549,531.12
IN9	510.0	5,302,368	✗	✓	✓	✓	1,408,671.75
IN10	1,100.0	11,520,000	✗	✓	✓	✓	1,761,181.25
IN11	2,057.6	21,600,000	✗	✗	✗	✓	1,750,877.75
IN12	3,962.5	41,600,000	✗	✗	✗	✓	1,790,669.12
IN13	7,881.1	82,800,000	✗	✗	✗	✓	1,951,489.75

Table 3.3: MDP problem sizes. Each row contains information regarding the ID, size, number of states and indication if the input data fit in the device memory. $R\pi$ column shows the expected exploitation reward of the optimal policy found in each case by VI (see main text).

The size of the MDP benchmark that is solvable heterogeneously on a platform is mainly limited by the available RAM and memory of the GPU, but also by how lightweight its operating system is. For instance, Kaby-Lake (Ubuntu) and Broadwell-Mobile (MacOS) can both handle MDP benchmarks that go up to IN11 in spite of the difference in RAM between the two. Broadwell-Desktop, the most resourceful from our selection, leaning towards medium-power use, can solve MDPs as large as IN13.

To generate MDPs of increasing size, we sample the robot interaction within the defined environment in the physically realistic V-REP simulator. The resulting data serves to model statistically representative transition functions for the real robot, as long as the model has $\approx$ 30,000 states. This size is far from being the largest solvable MDP on any of the test platforms. To go beyond, we have artificially generated larger MDP models using finer grained discretization of the continuous process (i.e. by setting the values for NT, ND, NR, NG and NA). Thus we not only show the adequacy of our results to robot navigation but to any other application that require more than 2,000,000 states (for all test platforms), or even reach over 80,000,000 states (for TP3).

In practice, a finer grain discretization of the MDP model does not necessarily help improving the policy. As explained in section 3.2.4, a way to find the optimal quantization is to compare the total estimated reward when exploiting the optimal policy in the long-term, $R\pi$ (see Eq. 3.4), for different levels of discretization. If this measure does not improve, then the use of a finer grain MDP is not justified.

$R\pi$ can be used for a fair comparison only as long as the reward definition does not change across different discretizations. In our navigation use-case, we have noticed that if the agent is rewarded with a much higher bonus for reaching the goal than the cost for colliding, the optimal discretization becomes IN6, and the resulting policy allows collisions in favor of reaching the target faster. On the contrary, if we penalize collision higher (not hitting people or obstacle while navigating is desirable in robot navigation), the policy benefits from a finer grain discretization, and, as a result, IN13 gives the highest valued $R\pi$. We have included the value of $R\pi$ in Table 3.3, calculated with a reward of 100 for reaching the goal state and a penalty of -500 for a collision in Eq. 3.1.

3.5.3 Productivity Evaluation: OpenCL vs. oneAPI

In this section, we discuss our findings regarding the evaluation of the Programmability of the programming models used for our implementations: OCL, BUFF, and USM. In particular, we want to characterize how productive, from a programmer's point of view, are the different approaches. In other words, we evaluate the ease of programming of each one. For it, we follow the methodology proposed in [39], where we find, among others, two quantitative metrics to measure the easiness of programming a code: the *Cyclomatic complexity* and the *Programming effort*. The Cyclomatic complexity (CC) is the number of predicates plus one, while the Programming effort (PE) is a function of the number of unique operands, unique operators, total operands, and total operators. The operands correspond to constants and identifiers, while the symbols or combinations of symbols that affect the value of operands constitute the operators. Higher values for both CC and PE metrics mean that it is more complicated for a programmer to code the algorithm.

Fig. 3.18 shows the results of the Programmability metrics (CC and PE) comparing OpenCL vs. the two oneAPI versions discussed in section 3.2, USM and BUFF. We break down the metrics for: i) each ViBody functor class (the kernel), ii) the scheduler engine part that is independent of the scheduling algorithm (ScheduleOCL vs. ScheduleUSM[8]); and iii) the total values of each metric when considering the kernel and the scheduler engine. As we see, oneAPI implementations achieve much lower complexity and programming effort than OpenCL. In particular, the most interesting metric, PE, shows that for the kernel implemen-

---

[8] ScheduleUSM has the same code as ScheduleBUFF, thus, their CC and PE are equal.

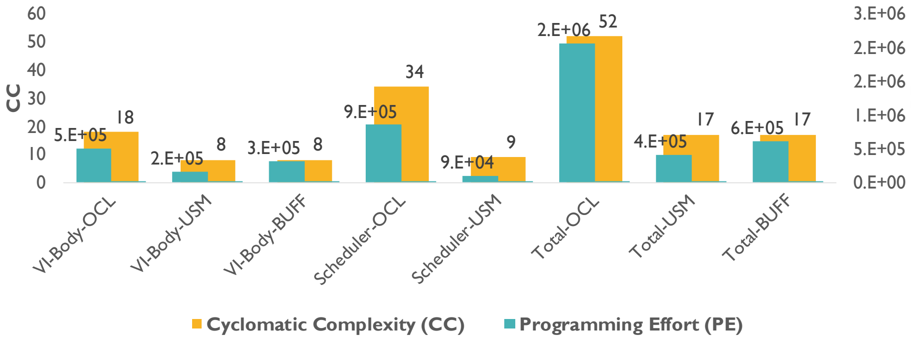


Figure 3.18: Evaluation of the *cyclomatic complexity* (left Y-axis) and *programming effort* (right Y-axis) of OpenCL (OCL) and oneAPI (BUFF, USM) based implementations. The lower, the better.

tation, BUFF and USM reduce 1,2x and 3x the programming effort, respectively, when compared to OCL. The reduction is even more significant when considering the whole application: 3.4× and 5.1×, respectively. Interestingly, the programming effort for the implementation of the kernel in USM is 86% lower than for BUFF, although the complexity is the same. Clearly, USM is the implementation with the least effort required.

From a programmer's point of view, the main difference between OpenCL and oneAPI, when coding ViBody and Scheduler are the following:

ViBody, Scheduler – The kernel source code in OpenCL is defined in a `kernel.cl` file. Next, a program is created using the device context and manually built into a program that can run on the device (`clCreateProgramWithSources`). Finally, we can create a kernel object with an OpenCL API call (`clCreateKernel`), and we are finally ready to set its arguments and enqueue work to it. When using oneAPI, one only has to submit a functor object to the GPU queue with the code (written in plain C++) that has to execute on the device. Also, the error checking code required for every OpenCL API call (`clCreateBuffer`, `clEnqueueWriteBuffer`, `clEnqueueReadBuffer`, `clSetKernelArgument`, ...) represents a considerable ratio of total lines of code of an OpenCL implementation. In oneAPI, we avoid this by wrapping the code in a try-catch-block, which captures the most frequent error codes and parses them to human readable messages.

Scheduler – All the code needed in OpenCL to get the platform, find the device, create a context and a command queue for the device are replaced by a single line of code in oneAPI (see line 4 Fig. 3.16).

3.5.4 Looking for the Sweet Spot: Productivity vs. Efficiency vs. Performance

Can we have energy-efficiency and performance while keeping the ease of programming? In this section, we discuss the impact on performance and energy efficiency of the different programming approaches [Subsection 3.5.4.1], as well as the impact of the scheduling strategies [Subsection 3.5.4.2].

3.5.4.1 Impact of the Programming Model

Here, we focus on the impact the programming model approach has on the performance and energy efficiency, factoring out the effect of the scheduling strategy. In Figs. 3.19 (a) and (b) we compare the time (in seconds) and energy consumption (in Joules) that each programming approach obtains in two scenarios: a) all the workload executes on the GPU, and b) the workload is statically distributed between the CPU multicore and the GPU devices at the beginning of the execution with the Oracle scheduler (HO). For HO, prior to the evaluation, we perform offline training to find the optimal partition between devices.

For both GPU-only and HO-* heterogeneous implementations, we find that BUFF implementations are the ones with worse performance, while both OCL and USM present similarly good results.

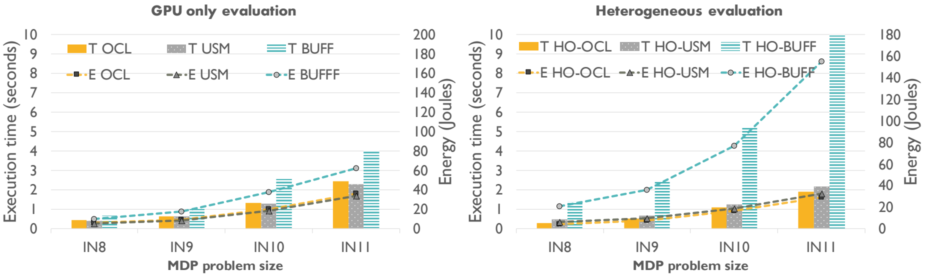


Figure 3.19: a) and b) Execution time (left Y-axis, bars) and energy consumption (right Y-axis, lines) for GPU-only and heterogeneous implementations based on Oracle scheduling (HO-*) using OpenCL (OCL), oneAPI & Unified Shared Memory (USM) and oneAPI & Sycl buffers (BUFF). The lower, the better.

In particular, if we focus on GPU-only implementations (Fig. 3.19.a) USM exhibits up to 5% and 78% more performance efficiency than OCL and BUFF, respectively, and up to 7% and 97% more energy efficiency than OCL and BUFF. Experiments with heterogeneous implementations based on the Oracle scheduler

(HO-*) (Fig. 3.19.b) reveal that HO-OCL always performs slightly better than HO-USM in terms of energy performance and execution time. They are both up to 380% more energy and time-efficient than HO-BUFF.

As we see, increasing the level of abstraction of the programming model to improve productivity may have a significant impact on the efficiency if we are not careful. In BUFF implementations, increasing the level of abstraction for data management through Buffer functionality helps to hide how different memory locations are mapped on different devices. But it degrades performance because the use of accessors suppose data movements (communication/copy operations). On the contrary, these data transfers are avoided in USM implementations, due to the hardware support of shared virtual memory among devices, so memory locations are accessed directly from the CPU and the GPU.

As USM performs better than BUFF for our set of benchmarks, from now on, we discard the HX-BUFF implementations for further tests and use HX-USM to represent oneAPI.

3.5.4.2 Impact of the Scheduling Strategy

Now let us discuss the impact of scheduling strategies in performance and energy efficiency. Fig. 3.20 shows the energy improvement and speedup (Y-axis) for the heterogeneous schedulers studied in this work: HO, HD, and HL when solving MDPs of sizes IN8 to IN11 (X-axis) on Kaby Lake. We compute the energy improvement and speedup against the baseline SEQ1 implementation (see Table 3.4). For the HO and HD schedulers, we perform offline profiling in which we explore the *RatioGPU* and *ChunkGPU* that achieve the maximum throughput for each input, and for them, we report the speedup and energy improvement we see in the figure. Let us recall that HL adaptively computes the optimal chunk sizes for the GPU and the CPU cores automatically, without user exploration. Table 3.5 reports the optimal *RatioGPU* for HO-* schedulers, optimal *ChunkGPU* for HD-* schedulers and average GPU chunk size for HL-* ones. We also show the final GPU ratio for Dynamic (HD-*) and LogFit (HL-*). Interestingly, HL-* schedulers tend to finally offload to the GPU a percentage of the workload similar to the optimal ratio manually found with HO-* (note that this search is done in steps of 10%).

In Table 3.4, we show the mean execution time (in seconds) and energy consumption (in Joules) for the VI execution of the sequential implementation (SEQ), which we use as the baseline for speedup and energy improvement. We show the results from IN8 to IN11 MDP sizes when executing on Kaby-Lake.

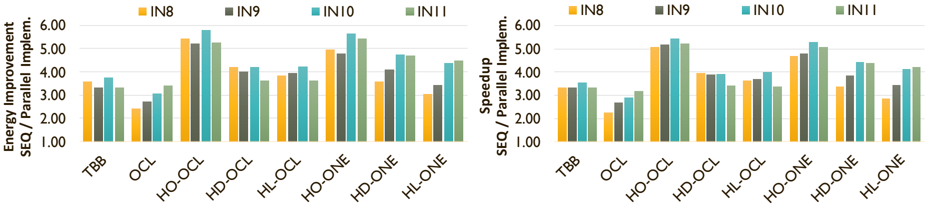


Figure 3.20: Energy improvement and Speedup of heterogeneous using Oracle (HO), Dynamic (HD), and LogFit (HL) schedulers implementations with oneAPI (*-USM) and OpenCL (*-OCL) on Kaby Lake. We include for reference the energy improvement and Speedup for CPU-only (TBB) and GPU-only (OCL) implementations. All the results are compared against the sequential (SEQ) code. Higher is better.

Execution Time (s)				Energy Consumption (Joule)			
IN8	IN9	IN10	IN11	IN8	IN9	IN10	IN11
1.17	2.30	5.07	8.98	14.38	28.43	62.72	111.96

Table 3.4: Mean execution time and energy consumption of SEQ1 implementation for benchmarks IN8-IN11. Used as baseline to compare the heterogeneous implementations (Fig. 3.20).

An interesting observation can be drawn from Fig. 3.20: for all four platforms and heterogeneous schedulers, the time and energy performance results are correlated, i.e., time and energy improve at a similar rate for all input sizes, although energy efficiency tends to achieve better values than speedup for HD and HL schedulers. When comparing the overall scalability of heterogeneous schedules with respect to TBB (see Fig. 3.21), we also observe that for HO rutime and energy improvements are highly correlated on all platforms. On the other hand, for HL, and especially for HD, energy improves at a higher rate than runtime. It is important to notice that except for Odroid-XU3, all heterogeneous implementations improve time and energy performance (TBB/Hx runtime and energy ratios are greater than one). On Odroid-XU3, we achieve a significant energy efficiency improvement only with HD, other strategies do not outperform TBB.

As we see from Fig. 3.20, the time and energy performance results are correlated, i.e., time and energy improve at a similar rate for all implementations and input sizes, although energy efficiency tends to achieve slightly better values than speedup. Also, all heterogeneous implementations outperform CPU-only (TBB) and GPU-only (OCL) implementations for execution time and energy consumption. We see that CPU+GPU heterogeneous strategies can be up to 54% (61%)

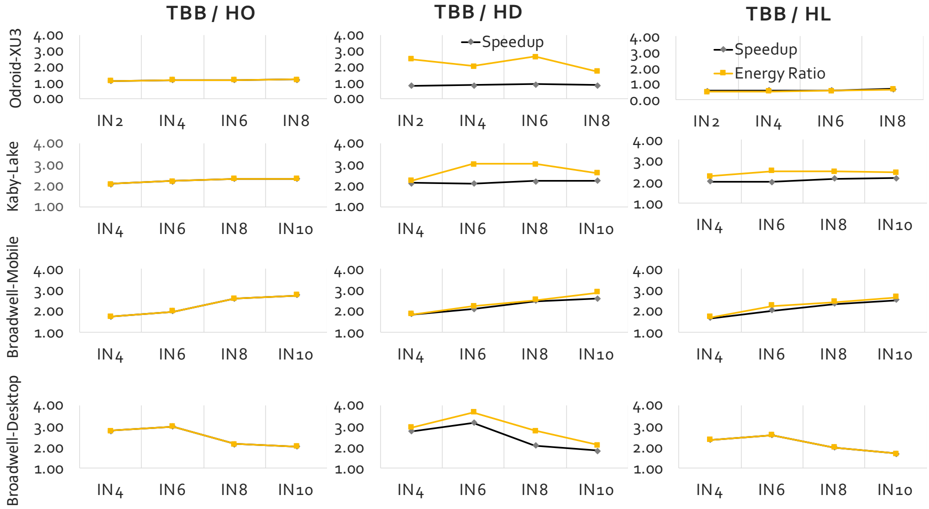


Figure 3.21: Time and energy ratios for heterogeneous vs CPU-only TBB implementations on the four SoC platforms.

faster and 57% (65%) more energy efficient when compared to CPU-only (or GPU-only) implementation. We also notice that Oracle scheduling tends to provide the best results, both for OCL and USM programming models. In other words, in spite of the application irregularities, a static partition can provide good results avoiding the partitioning overhead that Dynamic and LogFit incur. For any benchmark size, HO-OCL slightly outperforms HO-USM (see Fig. 3.22).

Both HD-* and HL-* tend to provide similar performance and energy efficiency, although HD-USM and HL-USM obtain better efficiencies than their OCL counterpart for larger benchmark sizes. This is visualized in Fig. 3.22(a),

	HO-OCL	HD-OCL	HL-OCL	HO-USM	HD-USM	HL-USM
IN8	70%	4194304 — 53%	3555388 — 60%	50%	1048576 — 31%	1417346 — 62%
IN9	70%	4194304 — 52%	7230334 — 61%	50%	1048576 — 43%	8696996 — 69%
IN10	70%	8388608 — 36%	9907622 — 61%	50%	2097152 — 41%	3800115 — 50%
IN11	70%	8388608 — 37%	11643141 — 60%	50%	4194304 — 45%	3361717 — 52%

Table 3.5: Optimal workload distribution for different scheduling strategies when executing IN8 to IN11 on Kaby-Lake. GPU ratio is shown for HO-OCL and HO-USM; (Chunk size | final GPU ratio) is shown for HD-OCL and HD-USM; (Average GPU Chunk Size | final GPU ratio) is shown for HL-OCL and HL-USM.

where we show the execution time ratio of OCL vs. USM for all schedulers and benchmarks sizes from IN8 to IN11. We mark the 1 ratio with a red horizontal line, indicating no difference between the execution time of x-OCL and x-USM. The energy ratios are similar for all inputs (not shown). From this figure, we conclude that for the three schedulers, the overhead introduced by using USM high-level programming over OpenCL decreases with the benchmark size, and as USM achieves better locality with bigger data sets, eventually it outperforms OCL.

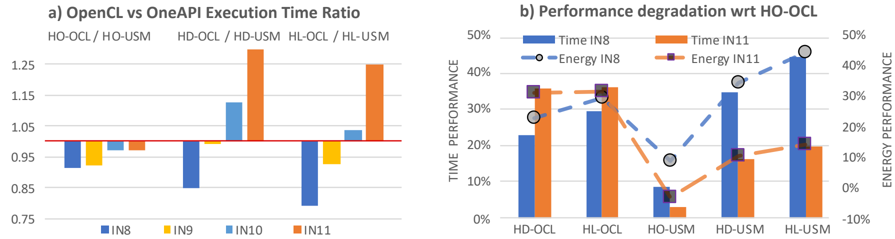


Figure 3.22: a) Execution time ratio of OpenCL (x-OCL) vs oneAPI (x-USM) based heterogeneous schedulers (HO, HD, HL). b) Time (bars) and energy (lines) degradation (IN8, IN11) with respect to HO-OCL, the implementation which gives the best energy and execution time results. The lower the better.

Fig. 3.22(b) assesses the performance degradation of our heterogeneous implementations with respect to the best one: HO-OCL, and for IN8 and IN11. We see that for the larger benchmark size, HD-USM and HL-USM only degrade time by 15% and 18%, and energy consumption by 16% and 19%, respectively, while higher degradation values are seen for HD-OCL and HL-OCL.

An important conclusion that we can draw from this discussion is that for large enough benchmarks, dynamic and adaptive heterogeneous schedulers benefit more from using oneAPI & USM than from using OpenCL. This probably happens because of the USM feature of oneAPI, which best exploits the locality of shared data thanks to the hardware support of shared virtual memory (e.g., memory allocated with *malloc_shared*(.)) and using the Level-Zero backend instead of OpenCL. Sycl buffers are also mapped to the Level-Zero backend under the hood; thus the data can migrate between for faster access [9].

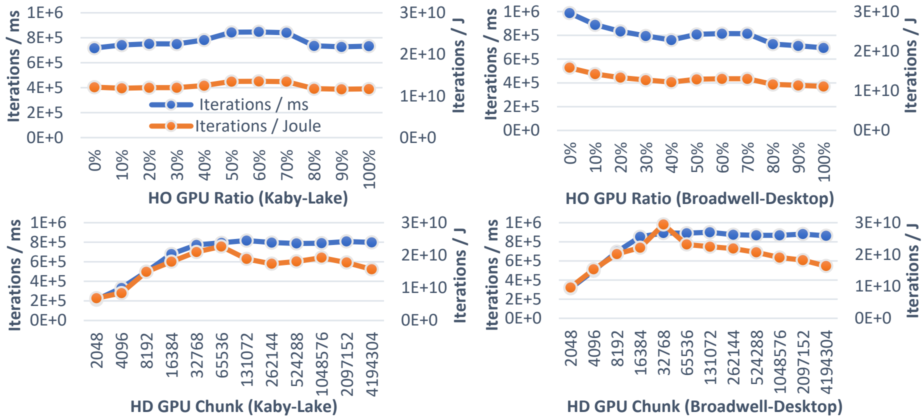


Figure 3.23: HO and HD Chunk Size exploration for IN10 on Kaby-Lake and Broadwell-Desktop. In the primary and secondary Y-axis, we show the throughput for sampled GPU chunk sizes (X-axis), measured as the number of parallel iterations per millisecond (left Y-axis) and the number of parallel iterations per Joule (right-Y axis).

3.5.4.3 Study of Offline-Tuned Schedulers: HO and HD

We start our analysis with the schedulers that require offline training to find the best work partition between the CPU and GPU: HO and HD. In Fig. 3.23, we illustrate how HO and HD work out the GPU ratio and GPU chunk size, respectively, for IN10 and Kaby-Lake and Broadwell-Desktop. Similar graphs are obtained for the other inputs. Note that, while for a static partitioner as HO, the GPU ratio that typically produces the highest iterations per Joule is the same that achieves the highest iterations per millisecond, this is not the case for a dynamic partitioner as HD.

For dynamic schedulers, we must compromise between energy and runtime when choosing the integrated GPU chunk size. Also, sending a larger chunk size to the GPU does not necessarily provide a better overall performance, as it happens with discrete GPUs, and this is especially true for the energy. Usually, throughput (iterations/ms) is conserved for increasingly large chunk sizes, but the energy efficiency (iterations/Joule) degrades when the chunk size increases over a certain point. For instance, on Kaby-Lake, chunk sizes larger than 65536 conserve the throughput relatively constant, while the energy steadily degrades.

---

[9] https://intel.github.io/llvm-docs/MultiTileCardWithLevelZero.html

The same happens on Broadwell-Desktop for chunk sizes larger than 32768.

To summarize, in the case of HO, the throughput and energy efficiency are proportional for all ratios in all platforms and MDP sizes. In the case of HD, the smallest chunk size that gives the highest throughput also provides the best energy efficiency. This is the insight that justifies why, when designing HL, we look for the smallest GPU chunk size that guarantees near-optimal throughput: it shall also guarantee near-optimal energy efficiency.

3.5.4.4 Study of the Adaptive Scheduler: HL

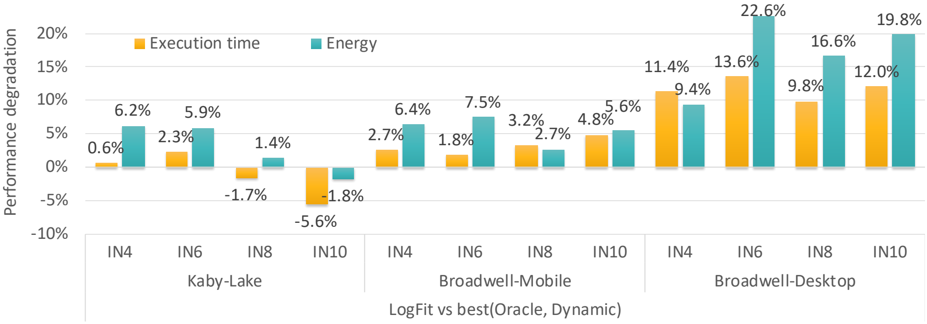


Figure 3.24: Adaptive vs offline-tuned Static and Dynamic scheduling: performance degradation of HL scheduler with respect to the experimental results of Oracle and Dynamic schedulers execution.

In this section, we discuss the properties of HL, our adaptive scheduler, and evaluate its performance in terms of scalability, energy and runtime performance against HO and HD. We report the Adaptive vs offline-tuned Static and Dynamic scheduling relative performance in Fig. 3.24, where we indicate the time and energy performance degradation of HL proportionately to the best result of HO and HD.

We note that HL over-performs HO and HD implementations only for energy consumption on Kaby-Lake and for IN6 on Broadwell-Mobile (see Fig. 3.24). On average, LogFit has the least performance loss on Kaby-Lake: it gains 1% for execution time and a has a loss of 3% in energy use. Next is Broadwell-Mobile, which presents a 3.1% performance degradation in execution time and a 5.5% in energy use. Last is Broadwell-Desktop platform with an 11.7% loss in execution time and 17.1% in energy. HD and HO give the best results for energy and time, respectively on this platform.

This degradation in performance is an indication of the overhead incurred in this scheduler due to the online training performed by the Exploration Phase, as well as the impact of the Final Phase. In both phases, the scheduler offloads to the GPU chunk sizes smaller than the optimal one, which produce sub-optimal performance. We explore this issue in more detail by looking at the throughput evolution we get for IN10 in Kaby-Lake –Fig. 3.25– and Broadwell-Desktop –Fig. 3.26–, the two platforms with the highest difference in performance degradation.

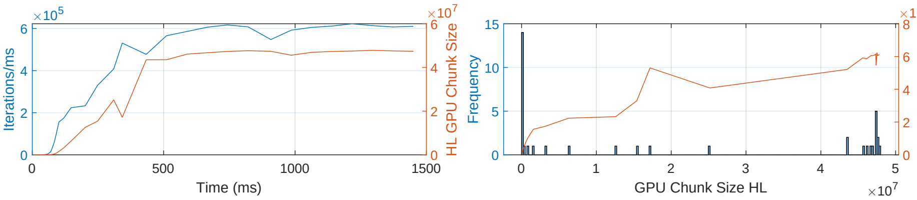


Figure 3.25: HL GPU Chunk Size exploration for IN10 on Kaby-Lake. In the left-side graph, we picture the throughput (parallel iterations/ms, left Y-axis) and GPU Chunk Sizes (right Y-axis) through the execution of VI (X-axis). In the right-side graph we show a histogram of the chunk size distribution through the execution of VI (left Y-axis) and the mean throughput of each chunk (right Y-axis).

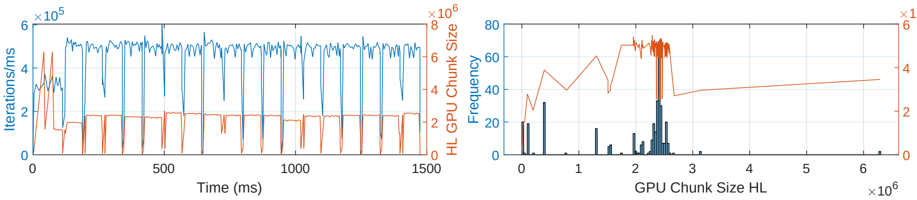


Figure 3.26: HL GPU Chunk Size exploration for IN10 on Broadwell-Desktop. Axis as in Fig. 3.25.

We observe two behaviors of HL, depending on the platform by studying Figs. 3.25 and 3.26. The left-side graph in Fig. 3.25 shows the GPU throughput/chunk size evolution (blue vs orange line) of the VI execution using HL on Kaby-Lake, while the right-side graph shows the histogram of the GPU chunk sizes distribution (bars) and their corresponding average throughput (orange line) for that execution. Similar information is displayed for Broadwell-Desktop in Fig. 3.26.

1. On Kaby-Lake, the Exploration Phase (EP) explores 15 samples and takes about two time-steps[10] (400 ms). Eventually, in the following time-steps LogFit remains in the Stable Phase (SP), where it delivers chunk sizes of $\approx 4.3 - 4.8 \times 10^7$ parallel iterations.
2. On Broadwell-Desktop, the Exploration Phase takes less time (just eight samples, less than 100 ms). Next, LogFit enters the Stable Phase, where it provides chunk sizes of $\approx 2.4 - 2.8 \times 10^6$ parallel iterations, which is close to the optimal we found training HD (3.2×10^6 iterations). We notice that the GPU chunks in the Stable Phase are an order of magnitude smaller than on Kaby-Lake for the same MDP size. We can also clearly see how the GPU chunk size actively adapts in response to the changes in throughput. Every peak and fall in throughput is mirrored with the GPU chunk size. We notice a periodic behavior due to the time-steps. In fact, at the end of each time-step HL enters the Final Phase (FP), where it splits the remaining iterations of the time-step between the GPU and the CPU. In the case of Broadwell-Desktop, this last GPU chunk size is too small and, therefore, sub-optimal, translating into a sharp drop in throughput.

As we see in Figs. 3.25 and 3.26 (especially), a significant number of chunks used in the EP and FP are small and produce lower throughput, in other words, are sub-optimal. This explains the higher degradation of throughput and energy efficiency for HL in Broadwell-Desktop. Thus, alternative implementations of the Exploration and Final Phase to avoid sub-optimal GPU chunk assignments must be studied for these platforms.

3.5.5 Time and Energy Consumption Evaluation

Here we discuss the performance of the CPU-only, Hybrid-1 and Hybrid-2 VI implementation strategies. To analyse and compare their performance, we have executed each one a hundred times (a statistically meaningful number of times for all platforms), recording the total execution time, total energy consumption, and the energy consumption corresponding to the CPU(s), GPU and Uncore components separately. We have then conducted a three-way ANalysis Of VAriance (ANOVA) on the results to determine whether there are any statistically significant differences both in the execution time and in the energy consumption measurements of the different VI implementations, input sizes and platforms. Our study includes a Tukey's post-hoc test to decide the ordering of elements when differences that are significant are detected [72]. Previously, we verified

---

[10] a time-step corresponds to a call of the *heterogeneous_parallel_for* function.

that the main ANOVA assumptions [11] are satisfied.

Please recall that the evaluated CPU-only implementations are *SEQ1*, *OMP* and *TBB* (see section 3.4.4). OMP and TBB use all available cores to execute VI on the CPU multicore: eight cores in TP0 from its two quad-core processors (big.LITTLE: Cortex-A7 and Cortex-A15); two cores in TP2 from a dual-core i7, and four cores in TP3 from a quad-core i7. The heterogeneous implementations are denoted *OCL-* (Hybrid-1), *HO-OCL*, and *HD-OCL* (Hybrid-2) (H stands for heterogeneous execution—please see section 3.4.5.1; O and D stand for using the Oracle, Dynamic or LogFit policy for the heterogeneous scheduling, thus enabling the simultaneous CPU-GPU exploitation—please see section 3.4.5.2). SEQ1 represents the best sequential implementation and it is used as reference. All of our implementations use the sparse 3D-lite-CSR representation for T.

In the following subsections, we investigate how the parallel VI implementation strategies behave for different MDP sizes across our three HCPs. We start by explaining how we tune the heterogeneous schedulers for optimal time and energy use and then continue with a global analysis of time and energy performance for all the parallel implementations. We end the current section with a summary of our findings and lessons learned.

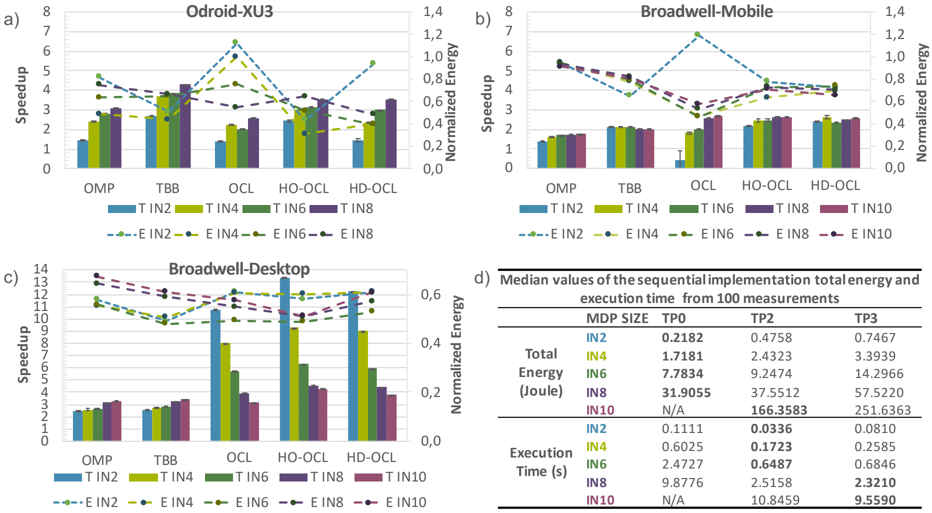
3.5.5.1 Scalability of Implementations

Let us examine now how well the parallel implementations scale up when increasing the MDP size by looking at Fig. 3.27. In plots a), b) and c) we distinguish the average speedup (left Y axis, bars) and normalized energy consumption (right Y axis, lines) for solving MDPs IN2, IN4, IN6, IN8 and IN10 on platforms TP1, TP2 and TP3. The sequential version (SEQ1) is used as baseline to compute the speedup and normalise the energy. The corresponding average execution time and total energy for SEQ1 are noted in Table *d)* (the bottom-left side of Fig. 3.27). The minimum average execution time and energy consumption for each input size are in bold.

Alternatively, Table 3.6 provides the energy and time per parallel iteration (mJoules/iteration and msec./iteration, respectively), which give us an idea of the minimum task grain (computational load) for each input problem (from a few msec. to several hundred msec.). In any case, for this task grain, the overhead due to scheduling management or kernel launching is negligible in the heterogeneous implementations (less than 0,04% in our experiments). Also, we would like to

---

[11] The measurements are normally distributed with homogeneous variance, and the measured observations are independent.



Median values of the sequential implementation total energy and execution time from 100 measurements

	MDP SIZE	TP0	TP2	TP3
Total Energy (Joule)	**IN2**	**0.2182**	0.4758	0.7467
	IN4	**1.7181**	2.4323	3.3939
	IN6	**7.7834**	9.2474	14.2966
	IN8	**31.9055**	37.5512	57.5220
	IN10	N/A	**166.3583**	251.6363
Execution Time (s)	**IN2**	0.1111	**0.0336**	0.0810
	IN4	0.6025	**0.1723**	0.2585
	IN6	2.4727	**0.6487**	0.6846
	IN8	9.8776	2.5158	**2.3210**
	IN10	N/A	10.8459	**9.5590**

Figure 3.27: a)-c) Average speedup and normalized energy consumption for CPU-only (OMP and TBB) and heterogeneous (OCL, HO-OCL, HD-OCL) implementations on the three test platforms, TP1, TP2 and TP3, with different problem sizes. Higher speedups and lower energy values are better; d) Average time and energy consumption evaluation for the sequential (SEQ1) implementation.

note that our heterogeneous implementations are based on the zero-copy buffer mechanism provided by OpenCL 1.2, so there is no communication overhead due to data movement. For TP2 and TP3 the main source of overhead is due to the map/unmap OpenCL operations that implement the zero-copy semantics, but they represent less than 2% of the execution time for any input problem. TP0 only supports OpenCL 1.1, so here it is mainly the device-to-host operation invoked for synchronising the Value array the responsible of an overhead of up 30% for the evaluated inputs.

1 Core	Odroid-XU3				Broadwell-Mobile					Broadwell-Desktop				
Execution	IN2	IN4	IN6	IN8	IN2	IN4	IN6	IN8	IN10	IN2	IN4	IN6	IN8	IN10
Energy/Iter (mJ)	14.55	114.54	518.89	2127.03	31.72	162.15	616.49	2503.41	11090.56	49.74	226.26	953.11	3834.80	16775.75
Time/Iter (ms)	7.41	40.17	164.84	658.51	2.24	11.49	43.24	167.72	723.06	5.40	17.23	45.64	154.73	637.27

Table 3.6: Energy and time task grain for SEQ1.

We can draw three main conclusions from Fig. 3.27. First, platforms that provide minimum execution time and platforms with minimum energy consumption do not coincide. The best results for execution time are found for TP2/TP3, while the best energy results are obtained in TP1. In other words, execution time and energy performance are platform dependent. Moreover, if we look at the performance of our parallel implementations at a given platform (plots a)-c)), the implementation that achieves the best speedup is not always the same as the one that offers the best energy result. Second, the heterogeneous implementations OCL, HO-OCL and HD-OCL usually obtain better results than CPU-only implementations, both for execution time and energy consumption. However, the heterogeneous scheduling that performs best depends on the platform and input size. Third, if we focus on CPU-only implementations, we can appreciate that TBB performs better than OMP, both for time and energy, in all cases.

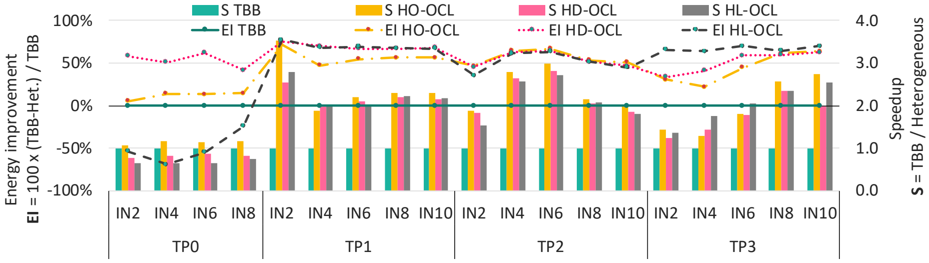


Figure 3.28: a) Speedup and energy improvement of Hybrid-2 implementations w.r.t. the best of CPU-only (TBB).

Now we examine how well the heterogeneous implementations scale up when increasing the MDP size with respect to the TBB-based implementation (best of multicore). In the graphs from Fig. 3.28 we distinguish the energy improvement and speedup (left Y-axis) of solutions based on heterogeneous schedulers: HO, HD, and HL for solving MDPs of sizes corresponding IN4, IN6, IN8 and IN10 (i.e., 16 MB, 68 MB, 274 MB, and 1190MB MDP models, X-axis) on our three test platforms, Kaby-Lake, Broadwell-Mobile, and Broadwell-Desktop. We compute the energy improvement and speedup against the baseline TBB implementation (see Table 3.7). For the HO and HD schedulers, we perform offline profiling in which we explore the *RatioGPU* and *ChunkGPU* that achieve the maximum throughput for each input on each platform, and for them, we report the speedup and energy improvement we see in the figure. Let's recall that HL adaptively computes the optimal chunk sizes for the GPU and the CPU cores automatically without user exploration.

In Table 3.8, we also report the mean execution time (in seconds) and energy consumption (in milliJoules) for the heterogeneous schedulers when using IN10 input on the three test platforms. In the table, for each platform, we mark the minimum time and energy consumption in bold.

CPU-Only Results	MDP Size	Kaby Lake	Broadwell Mobile	Broadwell Desktop
Execution Time (second)	IN4	0.066	0.089	**0.040**
	IN6	0.200	0.303	**0.140**
	IN8	0.732	1.136	**0.527**
	IN10	3.093	4.860	**2.234**
Energy Consumption (milliJoule)	IN4	4.138	5.568	**2.506**
	IN6	12.525	18.948	**8.743**
	IN8	45.774	71.023	**32.925**
	IN10	193.325	303.735	**139.601**

Table 3.7: Mean time and energy of TBB implementation. In parentheses the total number of cores of each platform.

Heterogeneous Results	Scheduler	Kaby Lake	Broadwell Mobile	Broadwell Desktop
Execution Time (second)	HO	**1.340**	**1.776**	**1.122**
	HD	1.415	1.880	1.247
	HL	1.410	1.942	1.363
Energy Consumption (milliJoule)	HO	83.788	111.011	70.185
	HD	81.344	**110.297**	**65.642**
	HL	**80.060**	116.176	81.575

Table 3.8: Mean time and energy of heterogeneous schedulers for IN10.

A first observation is that the speedup and energy improvement (Fig. 3.28) scale up for all heterogeneous implementations when increasing the MDP size on the platforms with the lower TDP (Kaby-Lake, Broadwell-Mobile). On the contrary, on the medium-power platform (Broadwell-Desktop), we see a deterioration in performance for MDPs larger than IN6. This can be more clearly appreciated in Fig. 3.27. In any case, we improve the execution time by up to 2.8x and, most importantly, reduce the energy consumption, by up to 3.2x, which clearly illustrates the importance of exploiting all computational devices in a heterogenous low-power SoC.

A second observation is that the static HO scheduler tends to achieve the best speedup for any input and platform, while dynamic schedulers HD or HL tend to provide the highest energy efficiency on all platforms. On average, HO performs best in terms of speedup and worst in terms of energy improvement on all platforms. HD gives the best results for energy, while HL second best, except on Kaby-Lake, where it has the best energy performance.

The main conclusions from Fig. 3.28 and Tables 3.7 and 3.8. is that, except for TP0, the heterogeneous implementations HO, HD and HL always improve the CPU-only implementation, both for execution time and energy consumption.

However, the implementation that performs best in terms of time is Oracle (Static scheduling), but in terms of energy consumption is either Dynamic or LogFit (Dynamic scheduling) —depending on the platform and input size.

3.5.5.2 Time Efficiency vs Energy Efficiency

To provide a global view of the time and energy efficiency of our implementations, we represent energy efficiency versus runtime efficiency of the parallel implementations in Fig. 3.29: OMP - ▽, TBB - *, OCL - +, HO-OCL - ×, and HD-OCL - **o**, and for representative problem sizes (IN2-blue, IN4-red, IN6-turquoise, and IN8-green). The energy and time values are normalized against the average energy consumption and execution time of the sequential implementation (SEQ1), so the lower values, the better, meaning both less energy consumption and smaller execution time.

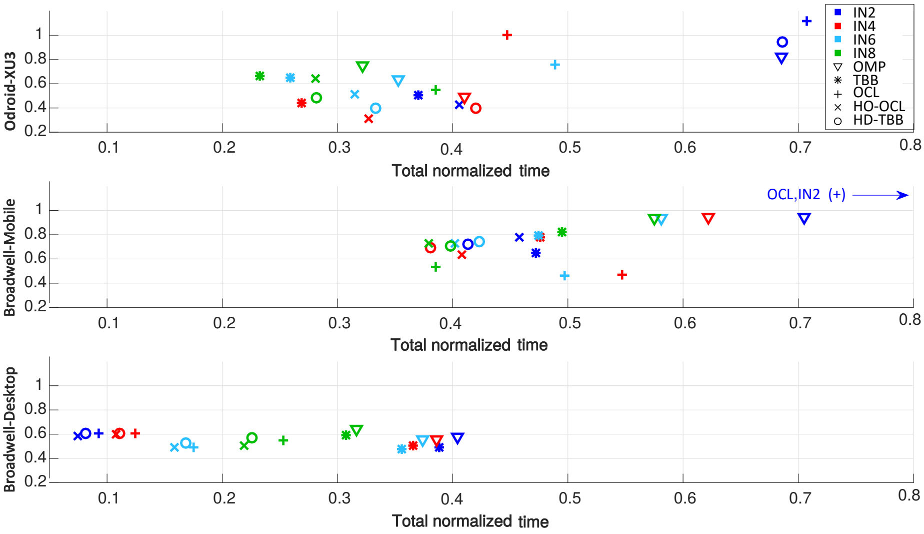


Figure 3.29: Normalized energy (Y axis) *vs* execution time (X axis) for different problem sizes, implementations and platforms. Bottom-left is better.

One interesting global observation is that parallel implementations show, in general, a high time efficiency in TP3 (the high performance computing platform) with values usually below 0.4), while in TP2 time efficiencies are worse (usually above 0.4). On TP0 (the low-power platform with lowest TDP), parallel implementations tend to show good time efficiency, except for small input sizes.

In particular, on TP0, all implementations with small MDPs have a low performance for time and energy (e.g., IN2, see top-right corner, blue), due to the not negligible device-to-host overhead that for these cases can be up to 30%. For larger MDPs (red, green, blue) this overhead is much smaller and the heterogeneous implementations HO-OCL and HD-OCL give better energy efficiency (i.e., they are placed at the bottom).

On TP2, heterogeneous implementations are similar in terms of energy and time behavior, while OMP implementations perform the worst for all MDP problem sizes but the smallest (IN2). Surprisingly, OCL gives the worst performance of all for input IN2 (due to a more noticeable overhead of map/unmap operations for this small input), but for larger input sizes OCL is the implementation with the best energy efficiency.

On TP3, the platform with the highest TDP, we observe something exciting: the energy efficiency is quite constant for all MDP sizes and implementations, while the time efficiency varies. Small problem sizes have better time efficiency here.

Table 3.9 shows average energy consumption (Joules), the execution time (seconds) and μWatts per Iteration for the implementations that perform the best for each input size on each platform. In parenthesis, we indicate the best implementation. In some cases, the best result is obtained by more than one implementation, i.e., they are indistinguishable under the statistical ANOVA procedure we have applied. For instance, HO-OCL and HD-OCL are both the best in terms of ex-

	MDP-ID	**TP0 Odroid-XU3**	**TP2 Broadwell-Mobile**	**TP3 Broadwell-Desktop**
Total Energy (Joule)	**IN2**	**0.0945 (GO-TBB)**	0.3091 (TBB)	0.3669 (TBB)
	IN4	**0.5364 (GO-TBB)**	1.1375 (G-TBB)	1.7187 (TBB)
	IN6	**3.0906 (GD-TBB)**	4.2591 (G-TBB)	6.8607 (GO-TBB,TBB)
	IN8	**15.4417 (GD-TBB)**	19.9627 (G-TBB)	29.2663 (GO-TBB)
	IN10	N/A	**96.0208 (G-TBB)**	128.7177 (GO-TBB)
Execution Time (s)	**IN2**	0.0415 (TBB)	0.0139 (TBB, GO-TBB, GD-TBB)	**0.0061 (GO-TBB, GD-TBB, TBB)**
	IN4	0.1622 (TBB)	0.0656 (GD-TBB)	**0.0278 (G-TBB, GO-TBB, GD-TBB)**
	IN6	0.6412 (TBB)	0.2606 (GO-TBB)	**0.1085 (GO-TBB, GD-TBB)**
	IN8	2.2973 (TBB)	0.9538 (GO-TBB)	**0.5085 (GO-TBB)**
	IN10	N/A	4.0252 (G-TBB)	**2.2261 (GO-TBB)**
$E/(T \times \lvert S \rvert)$ (μWatts/ Iteration)	**IN2**	123.6378 (HO-OCL)	1151.7860 (TBB)	666.6623 (TBB)
	IN4	20.1153 (HO-OCL)	131.3184 (OCL)	138.2441 (TBB)
	IN6	5.9316 (HD-OCL)	20.9176 (OCL)	99.6891 (HO-OCL)
	IN8	2.1110 (HD-OCL)	7.8317 (OCL)	21.9246 (HO-OCL)
	IN10	N/A	2.0707 (OCL)	5.0292 (HO-OCL)

Table 3.9: Performance of the best implementation for our test platforms and some representative problem sizes. We report the average total energy (Joules), time (sec.) and μWatts per Iteration. In bold the best result across platforms for each input. The lower the better.

Energy Bounds (Joule)	**Odroid-XU3**				**Broadwell-Mobile**					**Broadwell-Desktop**				
	IN2	IN4	IN6	IN8	IN2	IN4	IN6	IN8	IN10	IN2	IN4	IN6	IN8	IN10
Lower	**0.044**	**0.468**	**2.828**	**15.368**	0.292	1.025	3.924	18.967	**90.415**	0.301	1.462	6.223	27.309	123.386
HO-OCL	0.0957	0.516	4.056	15.502	0.330	1.123	4.571	21.767	99.467	0.395	2.043	6.517	29.199	128.786
HD-OCL	0.180	0.686	3.077	15.430	0.336	1.331	6.991	26.458	108.995	0.398	2.067	6.913	32.884	152.705
Upper	0.358	2.054	6.772	24.617	**0.739**	2.364	9.130	36.107	153.565	0.553	**2.559**	**9.688**	**38.923**	**242.727**

Table 3.10: Energy bounds (in Joules) for all platforms and MDP sizes. We mark in bold the inter-platform best (Lower bound) and worst (Upper bound) expected energy. We also report the energy consumption for HO-OCL and HD-OCL.

ecution time when executing a problem of size IN6 on platform TP3. Again in bold, we mark the minimum energy, execution time or μWatts per iteration for each input size. As we see from the table, heterogeneous implementations that exploit both devices simultaneously (HO-OCL and HD-OCL) usually achieve the best performance concerning time and energy. Interestingly, energy efficiency improves on each platform when the problem input size increases. The improvement is more pronounced on the platforms with lower TDP: TP0 and TP2.

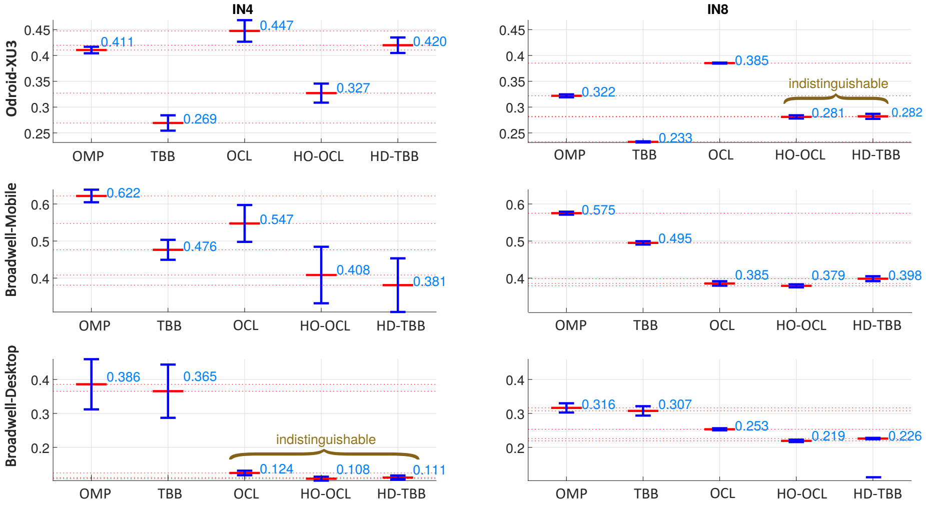


Figure 3.30: Normalized execution time for medium (IN4) and large (IN8) problem sizes. Averages are red bars; blue bars indicate 1-σ intervals.

To better appreciate how good the heterogeneous scheduling approaches perform from an energy point of view, we compare HO-OCL and HD-OCL to the best (Lower) and worst-case (Upper) scenarios, which can be found in Table 3.10.

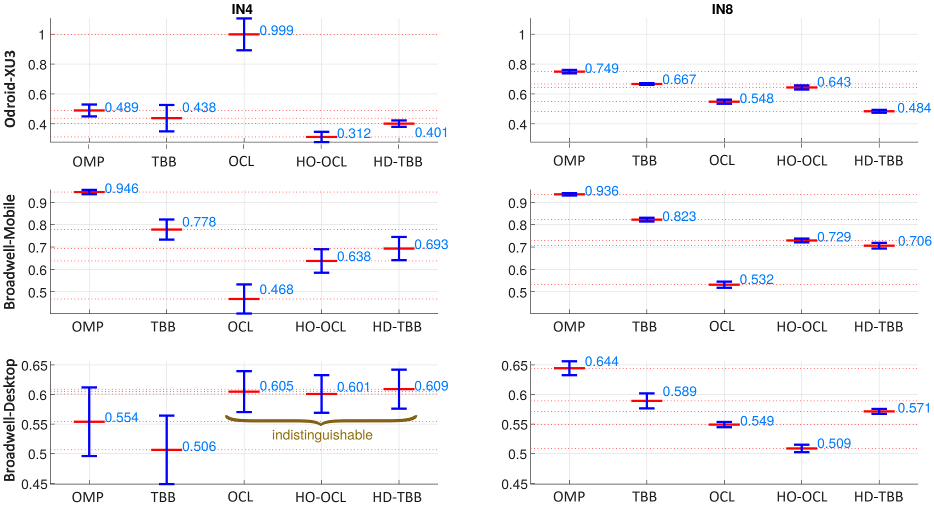


Figure 3.31: Normalized energy for medium (IN4) and large (IN8) problem sizes. Averages are red bars; blue bars indicate 1-σ intervals.

These Upper[12] and Lower energy bounds have been obtained empirically, using the minimum and maximum energy measured for all parallel strategies for a particular platform and MDP size. We remark that on average, HO-OCL (HD-OCL) is 18% (18%), 12% (36%) and 17% (25%) less energy efficient than the ideal scenario for TP0, TP2 and TP3, respectively. On the other hand, they demand around 2× less energy than the most energy-greedy parallel implementation, which clearly helps us to quantify the benefits of heterogeneous scheduling for the HCPs studied.

We end this section peeking at the behavior of our measurements of the total execution time and energy of the VI method. We also comment how our statistical analysis detects when the execution time and energy distributions of the implementations are statistically indistinguishable or not. We exemplify in detail the normalized time (Fig. 3.30) and energy (Fig. 3.31) for two of the most representative MDP sizes, IN4 and IN8. We have performed a similar study for IN2, IN6, and IN10 and observed a significant deviation for smaller inputs measurements ($\leq$IN4). We note that for large inputs ($\geq$IN8), the deviations in

---

[12] A theoretical energy upper bound of each platform and MDP size can be obtained by multiplying the maximum TDP (Watts) of the platform with the execution time of the slowest parallel implementation. This theoretical upper bound is always higher than the empirical upper bound we report in the paper, so we use the measured empirical value in the discussion.

the measurements, both for time and energy, remain small. In the figures, we also see that results are all different except for IN4 and the heterogeneous implementations in platform TP3 (time and energy), and for IN8 and HO-OCL and HD-OCL in TP0 (time). In these cases, the indicated implementations produce, statistically, the same results.

3.5.6 Summary of Results and Lessons Learned

From a quantitative perspective, concerning energy, TP1 is 1.3 to 3.3 times more efficient than TP2, and 1.9 to 3.9 times more efficient than TP3. Regarding time performance, TP3 executes 4.5 to 6.8 times faster than TP1 and 1.9 to 2.3 times faster than TP2.

Considering the CPU-only implementations for the evaluated benchmarks and platforms, TBB is always more efficient than OMP, both for time and energy, even though they use the same hardware resources. This aspect is most evident for TP1, the test platform with asymmetrical CPUs, where TBB executes 38% to 83% faster than OMP while using 12% to 61% less energy. It is important to mention that the TBB implementation gives the best time performance on TP1 for all implementations. These results seem to confirm that TBB work-stealing approach works better than dynamic or guided OMP approaches in our case of study. *Lesson Learned 1*: TBB is better suited for ARM big.LITTLE heterogeneous computing architecture. In future work, it would be valuable to evaluate the performance and energy efficiency of OpenMP and TBB for different platforms with asymmetrical CPUs.

The heterogeneous implementations, HO-OCL, and HD-OCL, perform better than TBB (the best of CPU-only) in all cases except for runtime in TP0 (Table 3.9, values in bold) or energy when solving small MDPs (with task granularity of a few msec.) on TP2 and TP3. *Lesson Learned 2*: low power platforms with asymmetrical computing units are more energy efficient exploiting all the units - heterogeneous implementations - but can be more time efficient exploiting only the faster units. The best heterogeneous implementation reduces the energy requirements of VI by up to 64% on TP0, 72% on TP2 and 19% on TP3, when compared to TBB (the most energy efficient CPU-only implementation). Regarding time, the heterogeneous implementations speedup up to 33% and 420% on TP2 and TP3, respectively, when compared to TBB. *Lesson Learned 3*: heterogeneous implementations obtain higher energy gains in platforms with lower TDPs, and higher time gains in platforms with higher TDPs.

Regarding the heterogeneous scheduling strategies (HO-OCL and HD-OCL), HD-OCL performs best for large MDPs (hundreds of msec.) on TP1, which is the most heterogeneous platform. In this case, the power use is reduced in ≈25% of the cases). HO-OCL either works better, or it is statistically indistinguishable from HD-OCL in the rest of the cases. *Lesson Learned 4*: the more heterogeneous the platform, the more benefit we can obtain from a dynamic work partition approach like the one applied in HD-OCL, versus the static partition approach applied in HO-OCL, being one additional advantage for HD-OCL that it does not need to assess all possible static partitions to find the optimal.

3.5.7 Rules of Thumb

Our study allows us to propose a simple set of rules to help programmers select the appropriate programming model and scheduling strategy for MDP-based solutions suitable for using the on-board low-power mobile platform:

- From a productivity's programmer point of view, oneAPI BUFF and USM implementations are easy to program. They result in much simpler codes than OpenCL. However, from a performance and energy efficiency point of view, USM is the choice to go, if the platform supports shared virtual memory by hardware.
- When considering the scheduling strategy, fine-tuned static scheduling performs best, both from time and energy efficiency point of view.
- On average, fine-tuned dynamic scheduling is the best compromise from an energy & performance point of view.
- From a practical point of view, both static (HO) and dynamic (HD) scheduling require time-consuming offline training to find the optimal GPU ratio/chunk size, which needs to be repeated every time the execution conditions or the input change.
- Adaptive scheduling (HL) offers a good trade-off between performance and ease of use, as it requires no previous training, given that the problem size is large enough and the HCP supports USM.

3.6 Conclusions

We seek to widen the applicability of decision-making methods that embed VI as an inner component to real-world problems, so we study the impact on the performance and energy efficiency of implementing VI with different heterogeneous programming approaches and scheduling strategies.

The implementation of a heterogeneous application that makes the most out of a CPU+iGPU SoC is a daunting task due to low-level considerations: data sharing, synchronization, load balancing, scheduling, etc. Usually, the user is responsible for providing the implementation of the code that processes a block of iterations (from the parallel iteration space) on the CPU and the GPU. To make this task more approachable, we have developed a high-level heterogeneous_parallel_for template [80] that takes care of many of the low-level details, like data partitioning, CPU-GPU synchronization, and scheduling.

For most of the SoCs with integrated GPU, OpenCL is the main alternative to write the GPU code, and C/C++ the widely accepted language to write the CPU code. This leads to a "dual-source programming paradigm" in which C/C++ has to interact with low-level OpenCL, and the developer has to master two different languages and learn how they interplay. A "single-source programming paradigm" alternative is the just-released Intel oneAPI [51]: a unified programming model that aims to simplify coding across multiple architectures, including CPUs, GPUs, FPGAs, and AI accelerators. It is an open standard based on the cross-architecture language Data Parallel C++ (DPC++). DPC++ complements SYCL [42] language and extensions for Unified Shared Memory (USM), ordered queues, reductions, subgroups (on CPU and GPU implementations), and data flow pipes (for FPGAs) support are being integrated and absorbed in the SYCL standard as they mature. DPC++ includes an abstraction layer on top of SYCL, which is a C++ abstraction layer over OpenCL.

The main benefit of using oneAPI over OpenCL is the single programming language approach, which enables us to target multiple devices using the same programming model, and, therefore, to have a cleaner, portable, and more readable code. Although oneAPI simplifies the development of heterogeneous applications, it does not automate the distribution and scheduling of the workload among the CPU cores and the GPU. To add this capability, in this work, we implement the heterogeneous_parallel_for template from [80] with oneAPI, and experimentally evaluate the pros and cons of oneAPI vs OpenCL for a heterogeneous application [27].

We have tested and analyzed the feasibility of solving large-scale Markov Decision Processes exactly with Value Iteration in low-power heterogeneous computing platforms for a set of robot navigation use-case benchmarks. From our experimental evaluation of the impact of the programming models on productivity, namely of OpenCL –OCL–, oneAPI & SYCL style buffers –BUFF– and oneAPI & USM feature –USM–, we have learned that oneAPI implementations can be up to 5x easier to program than OpenCL while incurring a negligible overhead in performance.

Although increasing the level of abstraction of the programming model improves productivity, this may significantly impact the efficiency if we are not careful: BUFF can be 380% less time-energy efficient than OCL. USM appears as a good alternative if the platform supports shared virtual memory by hardware. Last, from the three scheduling strategies evaluated, static –HO–, dynamic –HD–, and adaptive –HL–, the static scheduling performs best in terms of performance and energy efficiency, though it requires exhaustive offline searching. Adaptive scheduling provides good results with no previous training, in particular when using the USM approach to code the kernels and scheduler, and for large problem sizes.

In the next chapter, we branch from this research line and dive into a more complex decision-making framework—Partially Observable MDPs.

4

Online Planning on Mobile Platforms for POMDP Agents

4.1 Introduction

Used to model and solve real-world problems where uncertainty is always a given, partially observable decision-making processes (POMDPs) find applications in all sorts of domains, from management in agro-food industry [43], to stock market [4, 24], robotics [48, 36, 5, 71], air traffic control [59], and medicine [45, 47], to mention just a few. Karl Johan Åström first described the general framework of POMDPs in 1965 as a *Markov Decision Process with incomplete state information* [3]. It then became a hot topic ever since it was adapted for AI in autonomous decision-making and planning by Leslie Kaebling and Michael Littman [56]. However, in spite of having been intensely studied for half a century, the challenge of solving large POMDPs on platforms with limited resources remains open.

Solving *exactly* a POMDP to get the *optimal plan* is an EXPTIME-complete problem if the agent is a finite state machine with infinite memory [22]. In practice, POMDPs are computationally intractable due to the "curse of dimensionality" and the "curse of history". The two curses define the dimensions of complexity when planning for a POMDP and are often independent [87]. On the one hand, the curse of dimensionality, as first defined in [56], states that the belief space B grows exponentially with the size of the state space $|S|$. On

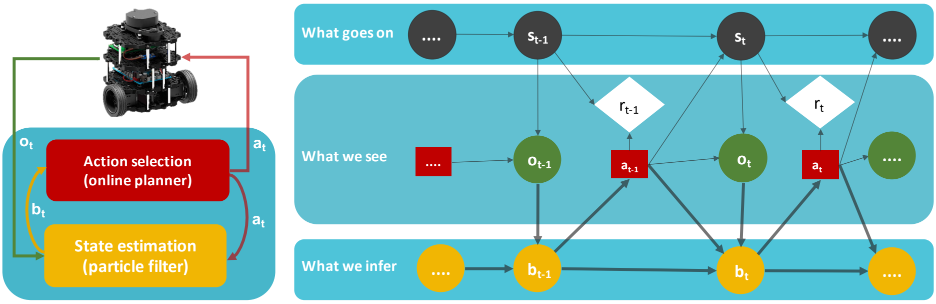


Figure 4.1: Online planning for a POMDP agent. Solving or planning for a POMDP consists in finding a policy that maps the current belief b_t to an action a_t, so that it maximizes some definition of a value, such as the expected total discounted reward r_t. The belief infers (approximately estimates) the underlying state of the agent, s_t, based on the observations of the agent, $o_t, o_{t-1}, ...$, and its previous actions.

the other hand, the curse of history states that computing an optimal policy for action selection grows exponentially with the planning horizon and the number of distinct possible action-observation histories [87]. In practical terms, we can compute an optimal policy only for toy problems whose state, action and observation spaces have few elements (e.g., tiger [19], crying baby [60]), and also for some medium-sized problems that admit an *offline* solution, when we have the luxury to compromise memory for time.

Unfortunately, for physical agents that have to make decisions *online* in the real-world, where an almost immediate response is required, we can only rely on approximate methods, which in the best case converge asymptotically to optimal solutions [44, 86, 78]. Online methods can scale to solve very large POMDPs, but often they are still slow for real-time online planning (they require roughly one second to plan for an action as of current technology) [37]. That is the issue we are addressing in this work. For completeness, we have included a detailed description of the POMDP formalism and methods in Section 2.2. How a POMDP models uncertainty and how an online planner works are depicted in Fig. 4.1, right and left respectively.

By analyzing the most relevant state-of-the-art works on online planning in POMDPs, such as POMCPOW, DESPOT, POMCP, or PBVI [107, 5, 100, 86], we have noticed that none of them attempts at saving and reusing previous planning experience. At every time-step, they simulate from scratch the execution of long sequences of actions and gathering of observations starting at the current belief state, in order to build a large approximation of the decision tree (step 1), then use it to choose a single action in the current time-step (step 2), and completely discard it afterwards (step 3). This process repeats until the agent reaches its goal or the stop condition. We think that online planning can be done more efficiently by tweaking with the last step and salvaging useful experience instead of discarding and forgetting the policy every time.

We claim that an experience memory based on lightweight data structures can be the link between seamless online search, offline policy computation and learning from example. Planning in the following three scenarios will likely benefit if the intelligent agent could retrieve precomputed policies or part of them based on similarity search to query for an entry similar to the current belief state:

- *Offline training*: We can incrementally store in an experience graph partial or approximated policies learned during offline training (from simulation) to provide a base of knowledge to the agent. Information about policies in this data structure can be added, updated and queried using the fingerprint of their belief state representation.
- *Learning from examples*: The same memory structure has the potential to serve as the base for learning admissible (not necessarily optimal) policies from example from expert and qualified operators. Another option is learning from playbacks of datasets that include valid sequences of histories for navigation, be it from simulation or on a real robot. By history we mean a sequence of actions and observations.
- *Online exploration and exploitation*: The base of knowledge built offline can be later accessed to get policy recommendations (or default actions) using similarity search queries for the current belief. Furthermore, the knowledge base can be further updated during online execution if a better policy with higher estimated value is found for the current belief.

Our goal is to do real-time planning for POMDP agents on low-power computing platforms, suitable for running on-board of physical agents that need efficient planning, such as mobile robots. Given this demanding context, we aim to reduce the computational cost of online planning for medium-large POMDPs.

The main result of our research presented in this chapter is a new method for online planning, the *Recall-Planner*. This proposal enhances existing online planners with a fast and lightweight memory based on Bloom filters, which allows the planner to store and recall experience. This experience can be obtained through direct interaction with the environment (on-policy) or simulation in previous planning steps (off-policy).

The rest of the chapter is organized as follows. We proceed with a Section 4.2

on related work. Then, we discuss the challenges of an online memory-based POMDP solver and our plan to make the memory proposal feasible in Section 4.3. Next, we present our proposal in Section 4.4, where we introduce a new algorithm for online planning and considerations for its implementations. We end the chapter with a systematic evaluation of the proposed algorithm in Section 4.7. Finally, we present an overview of the results in 4.7.6 and conclusions in Section 4.8.

4.2 Related Work

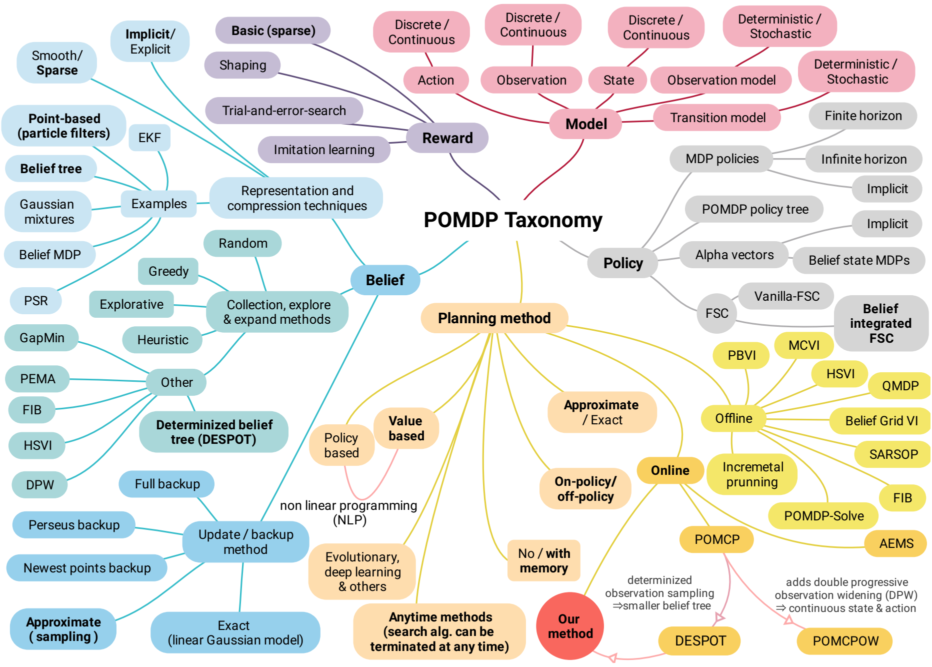


Figure 4.2: Taxonomy of related work in POMDP literature, showing the dimensions that define a POMDP problem and its solution: *model*, *belief*, *reward*, *policy*, and *planning method*. Marked in a red circle is our proposal for improving online planning methods, *Recall-Planner*. It is a type of *Online* planning method. Concretely, with an arrow-line directed from *DESPOT* to *Our method* we indicate that Recall-Planner builds on top of DESPOT. The same relationship rules apply to all elements in the diagram. We mark in **bold** the subset of methods in the POMDP literature that have been used in and are directly related to our proposal.

Chronologically, we start this work with an *exploratory research phase* to find which POMDP methods are more suitable for planning and decision making under uncertainty for mobile robot applications. The result of this phase is outlined in a taxonomy of POMDP methods presented in this section.

The proposed taxonomy of POMDPs is obtained by categorizing the different approaches to represent a POMDP problem (data structures) and the methods (algorithms) employed to solve it, which leads to five axes: model, belief, reward, policy, and planning (and decision making) method, as shown in Fig. 4.2.

As in MDPs, the starting point to use the POMDP framework is defining the **model** of the problem to be solved: i) the problem state, action, and observation domains, which can be each either discrete or continuous, and ii) the transition and observation functions, which can be either deterministic or stochastic. The difference with MDPs lies in the observations: we know from the previous chapter that MDPs plan and make decisions about the real underlying state of the system, but instead of the actual underlying state, POMDPs operate with a compact representation of the observable history of actions and observations of the agent, known as the **belief state** [12]. In an early application of POMDPs for "spoken dialogue management as planning and acting under uncertainty", the authors define the POMDP as "an MDP in which the agent can only make an observation based on the action and resulting state, without knowing the current state" [127].

The belief (blue axis in our taxonomy) can be represented either explicitly, as a probability distribution over the state space, or implicitly. Innovations in the belief representation and in the methods for *collecting, updating and expanding* the belief have lead to major improvements in POMDP literature [86, 95, 106, 15, 105, 107]. In particular, in state-of-the-art online planning [107, 37], point-based methods and particle filters are key to represent and propagate the belief. Common strategies for belief expansion in that case include stochastic simulation with *random* action, *greedy* action (choose the belief b^a that brings most reward), *exploitative* action (choose the belief b^a that is farthest from any $b \in B$) [86], *heuristic* actions and combinations, such as PEMA [88], GapMin [89] and FIB [104], etc.

Also similarly to MDPs, a POMDP can be defined as an optimization problem whose objective is to maximize some value described by a **reward** function (the mauve axis in our taxonomy). In other words, the reward function specifies the goal and behavior of the POMDP agent. The way the rewards are represented, chosen, or taught to the "intelligent" agent makes a whole field in AI research —closer to reinforcement learning. Some interesting lines of research on modeling the reward function include reward shaping [81], imitation learning, and try-and-error search. Regardless, the POMDP literature rarely ventures into *learning* the reward model; most common use-cases employ basic sparse rewards and so do we.

Comparing the existing methods, we conclude that *online anytime point-based planning methods* [98] have the potential to suit our needs as a foundation for planning in real-time for robot navigation use-cases. For more details on point-based methods, please check Section 2.2 in the background chapter. Consequently, we focus our efforts to improve the state of the art in this line of research. In particular, by enhancing online planning methods with a memory mechanism.

Modern online planners like DESPOT mitigate the two curses (dimensionality and history) and control the branching factor for both actions and observations by combining branch and bound techniques with sparse sampling. They end up with an approximation instead of an optimal policy (unlike VI for MDPs). All these online planners are anytime point-based methods, i.e., they approximately represent and update the belief using particle filters. They are widely used to compute approximately optimal solutions for POMDPs with high dimensional belief spaces and have been particularly successful in online planning. Some examples of point-based methods, such as PBVI, Perseus, HSVI, AEMS, POMCP, POMCPOW and DESPOT, are listed in Table 4.1 together with other representative planning methods.

Our contribution, the Recall-Planner, is located in the **planning methods** axis (orange). It is a type of online, value based, approximate, anytime, memory-based planning method. We do not draw all the connections for clarity, but we mark all relationships to the proposed method in bold. Looking at the taxonomy in Fig. 4.2, you may note that Recall-Planner is based on DESPOT [105, 37], which uses a point-based determinized belief tree to explore from the current belief. In turn, DESPOT is based on POMCP [100].

The novelty to our method is the use of a memory that can be used to store and recall off-policy belief-action entries, having a similar effect on planning to belief-integrated Finite State Controllers (FSC) [120]. Therefore, the current policy can be decided both based on experience from online planning (on-policy) and from prior experience (off-policy). The resulting plan is known as a **policy**, which is the last axis in our taxonomy (represented in gray color in Fig. 4.2). Anytime methods guarantee to return a policy in a given time and with a bounded regret. This property is desirable for online real-time applications.

Solver	**Reff.**	**Online**	**Offline**	**Conti. States**	**Cont. Actions**	**Cont. Observ.**
Belief Grid Value Iteration	[67]	-	✓	N	N	N
Point Based Value Iteration (PBVI)	[86]	✓	-	N	N	N
A hybrid between MDPs and POMDPs (QMDP)	[2]	-	✓	N	N	N
Anytime Error Minimization Search (AEMS)	[94]	✓	-	N	N	N
Successive Approximations of the Reachable Space under Optimal Policies (SARSOP)	[62]	-	✓	N	N	N
Monte Carlo Value Iteration (MCVI)	[6, 65]	-	✓	Y	N	Y
Partially Observable Monte Carlo Planning (POMCP)	[100]	✓	-	Y	N	N
POMDPSolve	[18, 74]	-	✓	N	N	N
Heuristic Search Value Iteration (HSVI)	[104]	-	✓	N	N	N
Fast Informed Bound Solver (FIB)	[89]	-	✓	N	N	N
Incremental Pruning	[20, 124]	-	✓	N	N	N
Determinized Sparse Partially Observable Tree (DESPOT)	[105, 16, 37]	✓	-	Y	N	N
POMCP with with Observation Widening (POMCPOW)	[107]	✓	-	Y	Y[*]	Y
Predictive State Representation MCTS (PSR-MCTS)	[66]	✓	✓	N	N	N
Information Particle Filter Tree (IPFT)	[35]	✓	-	Y	Y	Y
Recall-Planner	This work	✓	✓	Y	N	Y

Table 4.1: Representative POMDP methods with their main features. [*] Convergence to an optimal solution is not proven, and is not guaranteed to work for multidimensional action spaces.

4.3 Efficient Memory Mechanisms for Storing and Retrieving Online Planning Experience

Here, we study the implications of using a memory in combination with a point-based online planner, and possible implementation strategies for it. We are faced here with two design challenges:

1. When exploring the belief space of a POMDP, it is unlikely to revisit the exact same belief state: an exact memory mechanism would be useless because everything is new and different even by minor details. A “similarity”-recalling procedure is more practical. For instance, for a mobile robot, following a wall, turning around a corner, or entering through a door require very similar basic motion policies, regardless of the colors, textures and even shape of these elements. Thus we are looking for a way to systematically abstract the differences from states that require the same policy from a practical point of view.
2. Memory, computation and energy cost are of the utmost importance in the kind of physical agents we are interested in—we have to design a lightweight experience controller with a low computational and memory footprint to be useful in practice. The “compute, use, forget” planning strategy is the choice design in state-of-the-art methods because storing and recalling history statistics is prohibitive when considered in addition to the curse of

dimensionality.

We dedicate the following two sections to discuss different strategies we consider to minimize the effect of these major challenges on a memory-planner implementation that is usable in practice. Ultimately, we propose a new algorithm for online planning which uses a memory mechanism based on a combination of Bloom filters and similarity hashes.

4.3.1 Similarity Search in Belief Space

We need to design an efficient memory with minimum query delay and lightweight data structure to store and retrieve the policies already computed from explored belief states. Ideally, it should be able to generalize from past experience in order to recommended actions, even for previously unexplored beliefs, given that a belief similar enough to the one queried is recorded in memory.

We look for approximate search data structures and algorithms that can deal with high dimensional spaces, such as K-NN. We have identified three main groups of methods base on: space partition, locality sensitive hashing, and Bloom filters.

Space partition methods cluster all high-dimensional elements into multiple spaces and converts them into 1D space. K-d trees, R-trees, and SS-tress are such space-partitioning data structures used for organizing points in a k-dimensional space. K-d trees, for instance, work well on 1D and 2D data sets, but unfortunately, they handle poorly insertion and search for large multidimensional data sets [68, 102], i.e. they require sorting lists to add and delete items.

Either of these methods can do a lookup operation in multi-dimensional $O(N)$ time, N, being the number of entries in memory (for details, please consult Chapters 9 and 10 on K-d trees and similarity search tress in the *Advanced Algorithms and Data Structures* book [64]). Although randomized K-d tree improve the search and query performance and promise $O(log(N))$ [69], given that these methods are highly affected by the curse of dimensionality, they are not the most suitable for mapping and searching for multi-dimensional beliefs in POMDPs. Said this, we render them as as too computationally expensive to serve our purpose.

Locality sensitive hashing (LSH) methods solve the nearest neighbor problem by grouping points in bins or buckets; given a distance metric, those points that are close to each other end up in the same bin with high probability [102]. Implementations of LSH may take as input documents, images or other kinds of objects which can be represented as vectors in some metrical space. The output is a hash table. Objects with the same index in the hash-table are likely to be similar and objects with different indexes are likely to be "dissimilar". This "similar" function is defined by the distance metric and a chosen threshold for similarity.

Query and space are expensive, as these approaches store buckets of similar items and search through them. Fortunately, we do not need to store individual elements in buckets, just their signature and some associated data that is common to similar items (with the same or very similar signature, given a threshold). The complexity of LSH depends on the distance metrics used. A well designed hash table should allow an entry lookup in $O(1)$ time. LSH has been successfully applied to domains such as near-duplicate web page detection, genome-wide association study, and audio fingerprint [53, 113, 13].

Bloom Filter (BF) methods are based on probabilistic data structures, generally represented as bit vectors, and support membership queries using one or more hash functions. The BF data structure is called probabilistic because it can tell with 100% accuracy if an element is *not* a member of the set, but has an accuracy—controllable by design—lower than 100% when queried whether the element is *in* the set. With careful design, a number of d BFs can be used to store in memory and do membership retrieval of d-dimensional data elements in constant, $O(1)$, time. The most simple BF acts like a set that supports only two operations: add and check if an element is in the set. Functionalities such as counting and deleting elements can be implemented if necessary with layered Bloom filters [125], but with additional computational cost.

It is important to note that basic BFs are equivalent to lighter and faster *hash sets*, and we can only use them to store if a key is present or absent. The BFs' counterpart of *hash tables*, allowing us to store and consult key-value pairs, are the so-called *Bloomier filters* [23]. BFs, although not as popular as space partitioning and LSH methods, have seen much success in optimizing network and memory applications, such as routing, crawlers, and caching [14, 38, 30, 82].

We pay special attention to BFs because they have several interesting properties that make them a good candidate in the design of our experience memory controller:

- The time needed to add or check the membership of an element in the BF can be as small as a fixed constant, $O(K)$, if the hash functions are simple enough. K is the number of hash functions.
- The K hash functions of look-ups are independent, thus they can be run concurrently.

- Fewer than 10 bits per element guarantee a false positive probability lower than 1% for any set size and independently of the number of elements in the set [10].

Regarding to the false positive issue, there have been proposed multiple strategies in the literature to improve the precision metric and reduce the false positive rate [125]. Some prescribe using a larger number of hash functions, others suggest that instead of using a fixed size BF, the BF size can adapt and grow as needed to accommodate more entries in the set while maintaining the precision metric within the prescribed range. Another interesting strategy is to combine multiple Bloom filters. This improves both on the accuracy and parallelism of the implementations, but at the cost of multiplying the storage needed for the BF.

Our proposal, briefly introduced in the next subsection, and presented in detail in Section 4.4, uses a combination of the last two methods: Bloom filters and locality sensitive hashing. BFs are traditionally used in combination with cryptographic hash functions, which assure that small differences in the inputs lead to very different outputs. LSH, like perceptual hashes, have the opposite effect, i.e., similar inputs are mapped to similar outputs. We opt for using Bloom filters with locality sensitive hashing for being a generic approach that might as well be applied to varied domains such as documents of text, points in space, or audio files, to mention just a few examples. This versatility would make our experience memory proposal work for all sorts of POMDP problems.

The metrics for similarity embedded in these approaches make the difference between specific problems and implementations. In the case of robot navigation, we work with beliefs represented as multi-dimensional data points which approximate *poses* (positions+orientations) in space. In online planning the belief is represented as discrete probability distributions, while in the BF memory we operate with beliefs compressed in binary format. Considering this, we list the candidates for distance metrics we could use to quantify the similarity between two discrete probability distributions $P = (p_1, \dots, p_N)$ and $Q = (q_1, \dots, q_N)$, or, correspondingly, for binary strings A and B of size N:

- *Manhattan Distance*: It is computed with the L_1 norm, as $d_M(\mathbf{P}, \mathbf{Q}) = \|\mathbf{P} - \mathbf{Q}\|_1 = \sum_{i=1}^{N} |p_i - q_i|$
- *Euclidean Distance*: It is computed with the L_2 norm: $d_E(\mathbf{P}, \mathbf{Q}) = \|\mathbf{P} - \mathbf{Q}\|_2 = \sqrt{\sum_{i=1}^{N} |p_i - q_i|}$
- *Random Projection*: It partitions the space with hyperplanes and counts how many times the data falls on the same side of the same hyperplane. The closer together in space two points are, the more likely it is they will

not be separated by a random hyperplane and end on either side together. This metric is useful for K-NN applications.

- *Hamming Distance*: It is symmetric and easy to compute for binary strings A and B of N bits: $d_{HD} = count\_ones(XOR(A, B)))$.
- *Hellinger distance*: It is computed as $d_H(P, Q) = \frac{1}{\sqrt{2}} \sqrt{\sum_{i=1}^{N} (\sqrt{p_i} - \sqrt{q_i})^2}$ It has several desirable properties: it is symmetric, non-negative and bounded - close to zero for similar distributions and close to $\sqrt{2}$ when they are different.
- *Kullback–Leibler divergence*: Also known as relative entropy, it is a measure of how one probability distribution is different from another. For two distributions Q and P, the KL-divergence is defined as $d_{KL}(P||Q) = \sum_i p_i \ln \frac{p_i}{q_i}$ (change the sum by an integral for random continuous variables). It is a relatively easy to implement for point-based beliefs. It is popular in robotics and POMDPs, but it is not commutative nor symmetric, and it is not bounded either. It has been used in Monte Carlo POMDPs by Sebastian Thrun [111].
- *Jensen–Shannon divergence*: Also known as information radius, it is another method for measuring the similarity between two probability distributions. It is based on the Kullback–Leibler divergence, but it is symmetric and it always has a finite value.
- *Jaccard Similarity*: Also known as MinHash, it computes the intersection of points in two data sets, divided by their union.
- *Cosine Similarity*: Also known as SimHash, it represents data as vectors in multidimensional space; the distance between two such elements is computed as the cosine of the angle between them.

Preferably, the chosen metric for distance or similarity should be symmetric, i.e., $d(a, b) = d(b, a)$ in order to promote proper mapping for similar beliefs in both ways. Also, if we use the belief as such (sets of sampled states with their weights), it is very likely to always have near void intersection of two any pairs of beliefs, even overlapping ones, as required for Jaccard Similarity and Hamming distance. Euclidean Distance and Cosine Similarity are more likely to detect similarity correctly in this case.

To reduce the computational complexity of the metric, we need a lower dimension representation of the belief. A point-based representation already reduces the belief to a number of sampled points in the vicinity of likely states, but this may have to be further reduced. We could do this using different features of the belief probability distribution, such as the mean, standard deviation or other moments. These features will be the d dimensions used as input for the d BFs.

4.3.2 Considerations in Designing the Experience Memory

Being more specific, our experience memory design should implement these two requirements:

1. Generalize over the belief space, so that, given a belief, b, it returns the action associated to a belief b', previously visited and now stored in memory, that is at least s percent similar to b.
2. Efficiently store and recall experience entries from a bank of knowledge. By efficient, we mean compact data representation, timely insertion and query operations for the POMDP problem being solved.

We combine a state-of-the-art online planner with a Bloom filter memory using locality sensitive hashing—BF memory—into what we call the Recall-Planner. The motivation for proposing a design based on Bloom filters is the memory and computational cost of directly storing and searching for beliefs in memory. A belief is implemented as a set of N weighted particles, a particle is made of a sampled state and its likelihood of being the true state, and each state has d dimensions. If we were to search for the most similar belief to the current belief in a history of length L, we would need $N^2 \times d^2 \times L$ comparisons, which is very costly provided that each belief is represented as a large set of particles.

Bloom filters are lightweight and simple data structures with many desirable properties for our experience memory design (details in Sec. 4.3.1), but they are normally implemented with hash functions which assure that even slightly different inputs result into very different encodings. We need the exact opposite behavior in our memory for planning use-case, i.e., the hashes should encode similar inputs into similar encoding, allowing us to store similar experience indexed close to each other. This is exactly what locality-sensitive hashes do, and we naturally combine the two best candidates for solving our problem: BFs and LSHs.

To implement the BF memory proposal for a particular problem, we first need to identify the dominant features in the belief state that discriminate a belief from another. We also need a set of locality-sensitive hash functions for the problem at hand and corresponding features. The hash functions should assure that two similar beliefs, b_1 and b_2, having defined a threshold for similarity s, have the same encoding or coincide in the majority of the on bits with a high probability (ideally, with a well defined and bounded error for mismatching beliefs). The idea is to first encode (and compress) the queried belief based on its features, then use this encoding as a search key to answer similarity queries. If there is no match, the same key or encoding will be used to add a new entry to the memory (with the policy computed by the planner). Both the belief encoding and hash functions must be as simple and efficient as possible.

There are many questions we need to answer to produce an efficient design for our BF memory and the answers highly depend on the problem we want to solve, the hardware used to implement it, and what we consider a good solution. Let us start with the problem. We focus this work on 2D navigation (e.g., mobile robotics, autonomous vehicles), hence our states will always feature some distance metric to a target position relative to a universal coordinate system. In a sample-based multi dimensional belief such as this, the most practical features are the mean values and standard deviation in each dimension of the state. If the state of the agent is defined only by its position and orientation in space, the belief about the underlying state could be reduced to only 6 numbers: 3 mean values and 3 standard deviations, corresponding to the position x, y, and the orientation α relative to the target state.

Another critical question in our design is how many bits should the Bloom filter have? The size depends on what we consider an admissible discretization of the state space in our use case scenario. For the online case, we could limit the BF to adding only reachable belief points in the fashion of a discrete sliding window on top of the state space. Also, we keep in mind to balance costs of the memory storage, access, search and (re)computation of a policy from scratch.

Finally, how many independent hashes do we need for a BF? In general, at least as many as the number of features the encoded data has. It is common practice to apply the same hash to all features, given a different, constant seed for each dimension as an additional input to the hash function. This works fine for a prototype with a sequential implementation, but it is difficult to scale and use it for large high dimensional state problems because multiple hashes will have to access and write over the same positions in memory. As an alternative, instead of using a single large BF for all features, with many different hash functions that read and write to overlapping regions in memory, we opt to use multiple, smaller BFs, each with few hash functions. This design decision makes the solution scalable for larger problems. It simplifies load balancing tasks for parallel and heterogeneous computing, which is particularly useful, as we will implement the Bloom memory solution on a CPU+GPU SoC.

4.4 Online Planning with Bloom Filter Memory

Recall that our proposal is based on Bloom filters and similarity search, introduced in section 4.3.1. In this section, we describe in detail how we use these methods to design a lightweight memory that can store and retrieve online planning experience.

4.4.1 Data Structures – Bloom Filter Memory

Bloom filters are simple and efficient probabilistic data structures that can be used to add and test the membership of an element to a set. A basic BF is composed of two elements. First, a bit vector, which is a data structure with indexed binary values (zeros and ones) of size m. Second, k LSH functions, $h_0, h_1, ...h_{k-1}$. Each hash function gets input data (e.g., a string) and optionally, a seed, and returns an integer $i \in \{0, m-1\}$. Optionally, a set of seeds for the hash functions may be used.

We use the following notations and formulas to define the BF memory in our proposal:

- $m = [\frac{n * ln(p)}{ln(1/2^{ln(2)})}]$ - Size for the bit vector used to represent the BF (number of bits in the filter), given the number of elements in the sets, n, and the desired probability for false positives, p.
- $n = [-\frac{m}{k/ln(1-e^{ln(p)/k})}]$ - Capacity or maximum number of different elements that can be stored in the set.
- $E = \{e_1, e_2, ..., e_{n*}\}$ - Set of elements or entries in the BF memory. $n^* = -\frac{m}{k} \ln \left[1 - \frac{X}{m}\right]$ approximates the number of elements in the BF, given that X is the number of bits set to 1 in the BF [108].
- $p = \epsilon \approx \left(1 - e^{-kn/m}\right)^k$ - The false positive error tolerance (probability of false positives, $\epsilon \in [0, 1]$).
- k - Number of independent locality sensitive hash functions. The optimal number of hash function that minimizes the false probability, for a given m and n, is $k = \frac{m}{n} * ln(2)$. In particular, we use two hashes for each belief state dimension.
- $f = 2d$ - Number of features, is equal to twice the number of dimensions of the belief representation.

Some of these notations can be identified in the graphical description of the BF memory from Fig. 4.3. From left to right, we depict how a belief state is transformed into the belief marker using the $2 \times d$ LSH hash functions. This belief marker is used as a key to identify an entry the BF memory (BF key, colored rectangles).

We illustrate the main components of the BF experience memory and how it is used for online planning in Fig. 4.3. From left to right, we see how the belief features are extracted from the current belief and encoded with LSH to obtain a **BF key**. This key is compared to the memory trace BFs and depending on the result—whether the query key and memory trace match—there are three possible outcomes:

- **Outcome [1]**, in green: happens if the key matches and it is a true positive, the agent is returned the key-value pair stored in memory and can use the recalled policy.
- **Outcome [2]**, in red: happens if the key matches but it is a false positive. False positives occur with a small probability, controlled by design with the false positive tolerance parameter, ϵ. This case has a close to zero probability, but in the event of it, the recall returns “not found” (line 15 in Alg. 1) and the agent proceeds to compute the policy with online search.
- **Outcome [3]**, in blue: it happens if the key does not match the memory trace BFs and is a true negative. When this is the case, we have a 100% probability of accuracy (the memory stores no entry whose key is similar to the queried belief). Similarly to case 2, the agent proceeds to finding the

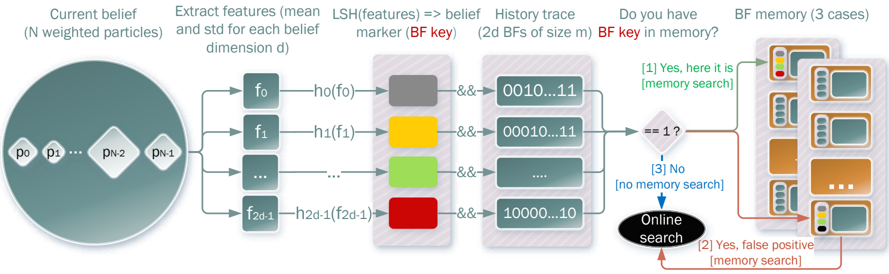


Figure 4.3: Representation of the Bloom filter memory use to recall a belief. From left to right, we have the input for a BF Memory query: the current belief, whose fingerprint is extracted in a feature vector and hashed into a *BF key* (belief marker). Next, the BF key is used to query the existence of a similar entry (a BF key of previously visited belief) in the *history trace* BF set. Finally, depending on the query result, the *BF memory* (hash table) may be searched or not to return the policy/value of a similar BF key.

policy with online search, but without the cost of searching the memory first.

4.4.2 The Recall-Planner Algorithm

The Recall-Planner is an anytime POMDP solver based on DESPOT-α [37], which is our baseline, enhanced with an experience memory controller. By "anytime", we mean that the search for a good policy can be interrupted at any time, and the algorithm would still return a valid action to execute. The algorithm requires as input at least the current belief b and the BF memory, which is composed by the memory trace (mem_trace) and experience memory ($memory$). The memory trace is implemented with a number of n_bfs Bloom filters where each BF has a bit vector of size m.

Our implementation uses a large number of parameters. Here, we explain only the most relevant in our evaluation. The first four (N, d, t, γ) are used in any standard online planner—for in depth details, please consult the background Section 2.2 on POMDPs and point based methods for online planning—and the rest $(n\_bfs, s\_th, m, n\_on, r\_d)$ are specific to our memory-based implementation:

1. N: Number of particles used to approximate the belief state (default: 500).
2. d: Maximum depth of the search tree (default: 90).
3. t: Search time per move - the search stops after t seconds, and the best valued action found thus far is returned (default: 1 second).
4. γ: POMDP discount factor (default: 0.95).
5. c_th: Percentage of value range that is allowed in STD deviation for an experience to be stored in memory (default: 0.05).
6. n_bfs: Number of BFs used. It equals $2 \times f$, being f the number of belief features. The features used in our 2D-navigation benchmarks include the mean and standard deviation for each dimension of the agent state in the POMDP model, i.e., the *pose* (x, y, α) of the robot.
7. m: Size of the bit vector of the BFs (default: 2048).
8. n_on: Fixed number of bits set to 1 by the hash functions, used to encode the belief features into the belief marker (default: 24).
9. s_th: Similarity threshold, measured as the percentage of belief overlapping between memory and planning for recalling the former. Values within the range 0-1 (default: 1).
10. r_d: The search tree depth at which the solver stops storing experiences in memory. If it is set to one, it only stores experience from direct interaction with the environment. Higher values enable the memory controller to in-

clude experience from Forward Search with Monte Carlo Simulation. We keep this parameter low to limit the possibility of storing experience entries from unreachable belief states, unlikely to be recalled in real interaction with the environment (default: 3).

Algorithm 1: Recall-Planner

```
Input: model = {S, A, T, Z, R, $\gamma$}                        // POMDP model
memory_trace(n_bfs, m)                                        // m BFs of size n
memory                                                        // experience memory
c_th                                                          // belief certainty threshold
s_th                                                          // belief similarity threshold
$s_t$                                                         // the target state
$b = sample(b_0, N)$                      // sample N particles from initial belief $b_0$
Output: updated memory and memory_trace
while $s_t$ is not reached do
    if certainty(b) > c_th then
        /* Only if certainty(b) > c_th we may recall from and store into the
           experience memory                                                    */
        found = false
        b_f = extract_features(b) // mean, std
        b_marker = sim_hash(b_f, n_on, m)
        if b_marker & memory_trace == 1 then
            /* Bitwise and operation between the corresponding belief marker
               (multiple bit vector encodings) and memory trace (composed of
               an equal number of BF bit vectors) equals 1.                     */
            found, a = memory.recall(b_marker, s_th)
        end
        if not found then
            a = solve(model, b, t, d, r_d)
            memory.store(b_marker, a)
            /* Update memory trace with bitwise or operation of the
               corresponding belief marker and memory trace vectors.            */
            memory_trace |= b_marker
        end
    else
        a = solve(model, b, t, d, r_d)
    end
    /* Robot executes action a                                                  */
    execute(a)
    /* Get observation from sensor readings                                     */
    o = observe()
    /* Update belief estimate using a weighted particle filter                  */
    b = update(b, o)
end
```

In Alg. 1 we show how a POMDP agent could use our BF memory in combination with an online planner. The robot executes the operations in the while loop until the stop condition is reached, which could be either a response time constraint or that the task is completed. The agent starts in an initial belief $b = b_0$ with an empty experience memory. In every time step of the loop, the agent plans for the next action, executes it, observes the outcomes of its action in the environment, and, based on this information, updates its state estimation. If the certainty of this state estimation (belief b in the algorithm) is over a threshold *c_th*, the robot uses the experience memory to recall the best action a. Otherwise, it calls the action selection method *solve*, which chooses the action with highest value.

The *solve(.)* method simulates forward in time, taking all possible actions and likely observations from the current belief (the root node of the decision tree) to a depth d or until timeout t, and returns the action with the highest expected (discounted) reward. Some examples of methods used in the literature for searching, planning or solving a POMDP online are UCB, Rollout Search, greedy UCB, DESPOT, and POMCPOW. In Alg. 2, we present a generic pseudo-code for the *solve(.)* method.

```
<bool, int> recall(bitset<m>* b_marker, float s_th){
  int max_sim_score(0), max_sim_id(0), action(0);
  // for each experience entry in memory
  for (i = 0:memory.size()){
    int sim_score = 0;
    // compute similarity score of experience entry wrt b_marker
    for (j = 0:n_bfs) {
      // sum similarity score for each feature
      sim_score += (b_marker[j]) & memory[i].b_marker[j]).count();
    }
    //find the id of the most similar experience entry to b_marker
    if (sim_score > max_sim_score) {
      max_sim_score = sim_score;
      max_sim_id = i;
    }
  }
  bool found_sim_experience = s_th <= max_sim_score;
  if (found_sim_experience)
    action = memory.getAction(max_sim_id);
  return <found_sim_experience, action>;
}
```

Figure 4.4: Kernel used to find in memory the experience entry most similar to the current belief. The similarity score (sim_score) is computed with the Hamming weight (counting the number of ones) of a bit set resulting from the AND operation between belief markers.

The core kernel of the Recall-Planner is represented by the recall(.) method. The kernel searches for the experience entry in memory that is most similar to

Algorithm 2: Generic online point-based POMDP planner

```
Input: $model, b_0, t, d$
Output: $a^*$
depth=0
$B \leftarrow \varnothing$
Sample $N$ particles $p$ from $b_0$ and set their weight to $1/N$
$B \leftarrow B \cup b_0$
/* Expand belief (decision) tree from $b_0$:                                  */
while depth of the tree $< d \wedge$ elapsed time $< t$ do
    for each leaf node $b \in B$ do
        for each $a \in A$ do
            /* Expand the belief tree, branching from node $b$                */
            for each particle $p_i = (s_i, w_i)$ in $b$ do
                Apply action $a$ and obtain $s'_i, r_i, o_i$ through transition and
                  observation models
            end
            for each observation $o_i$ do
                Create a new belief $b_{ao_i}$ (child of b through $a$ and $o_i$) from all
                  samples with $state = s'_i$ and $weight = w_i \cdot P(o_i|s'_i, a)$
                Annotate $r_i$ and $w_i$ in the new belief node $b_{ao_i}$
                $B \leftarrow B \cup b_{ao_i}$;
            end
        end
        depth++
    end
end
/* Backup:                                                                    */
for all leaf nodes $b \in B$ do
    Annotate $V(b) = 0$ in all leaf nodes $b \in B$
end
for all nodes $b \in B$ do
    /* Computed in post order (from leaves to root), first all children,
       then the parent                                                        */
    Annotate the node $b$ with $V(b)$ and the arc to its offspring with $Q(b, a)$:
      $V(b) = argmax_a Q(b, a), \quad Q(b, a) = \sum_{i=1}^{N} w_i / \sum_{w_i} (r_i + \gamma \cdot V(b_{ao_i}))$
end
Compute best action for $b_0$ as $a^* = argmax_a Q(b_0, a)$
Return $a^*$
```

the current belief and returns its corresponding policy (action) or *false* if there is no entry similar enough to the current belief. The method takes as input the current belief marker, *b_marker*, and a similarity threshold, *s_th* (see line 13 in Alg.1). In Fig. 4.4 we show its pseudo-C++ code.

For a graphical view of how the Recall-Planner algorithm works, we present in Fig. 4.5 a sequence diagram of the main classes in our C++ implementation,

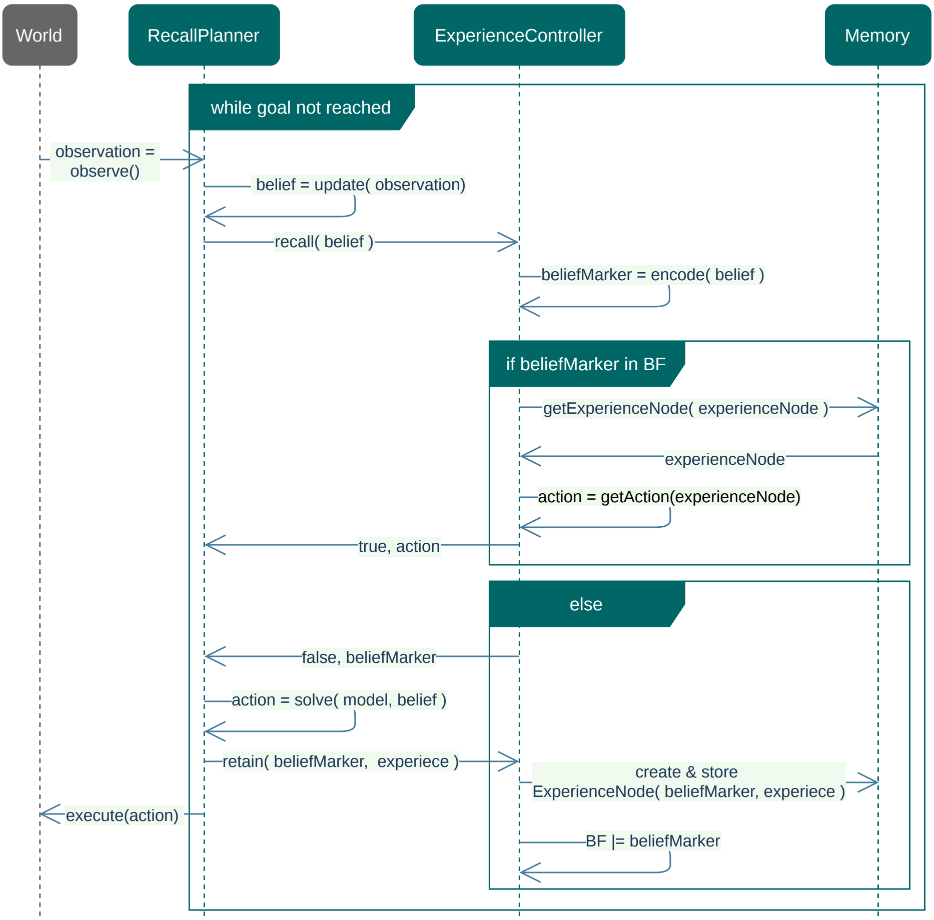

World
RecallPlanner
ExperienceController
Memory
while goal not reached
observation = observe()
belief = update(observation)
recall(belief)
beliefMarker = encode(belief)
if beliefMarker in BF
getExperienceNode(experienceNode)
experienceNode
action = getAction(experienceNode)
true, action
else
false, beliefMarker
action = solve(model, belief)
retain(beliefMarker, experiece)
create & store ExperienceNode(beliefMarker, experiece)
execute(action)
BF |= beliefMarker

Figure 4.5: Sequence diagram of Recall-Planner: the modules in blue correspond to the agent controller which interacts with the world through actions and observations.

and how they interact. As in any online planning problem, we have an agent that sequentially observes and acts in a given environment. In the diagram, we show the planner components in green, and the environment—the world in which it operates—in gray. First, Recall-Planner processes the agent observation to actualize a belief state about the world. Second, based on this new belief and prior entries in the memory, it 1) computes an action that takes the agent closer to reaching its goal and 2) updates its memory in the process. Last, the agent interacts with the environment and possibly changes the state of the world by executing the action. The planner reiterates over these steps until the goal is reached.

4.5 Implementation Strategies and Guidelines

The Recall-Planner combines a state-of-the-art online planner with BFs and LSH, as described in the previous section (see Fig. 4.5 for reference). In the following, we enumerate the key strategies used in our implementation:

1. Never store the full belief —we minimize memory use by storing only a "belief marker", the learned action, and the estimated value of that policy for each experience entry. We represent the belief marker by the smallest number of features that distinguish any two beliefs requiring different actions. We choose these features depending on the problem. For instance, in mobile robot 2D navigation, we use the mean and standard deviation for each dimension of the particle belief. Thus, we store only a belief marker of $2 \times d$ values.
2. Do store and recall operations only when we have sufficient confidence about the underlying state. We set this parameter experimentally to $c\_th < 0.05$, which translates to more than 95% confidence. In every other case, we use a state-of-the-art memoryless online planner based on DESPOT and DESPOT-α [105, 37].
3. Store only those experience entries likely to be revisited in a few steps of arbitrary sequences of actions and observations gathered in the decision tree, starting at the current belief state.
4. We define the *memory trace* as a set of BFs, one for each feature and dimension of the belief (6 BFs in the 2D navigation example, since the state has 3 dimensions: x, y and the orientation α). All BFs bit vectors are initially set to zeros.
5. Always update the memory trace (history fingerprint) when storing a new experience entry in memory; otherwise, the BF becomes useless. We do this

with a binary `or` operation between the corresponding BF and the belief feature encoded with its LSH function.

6. Make sure that similar beliefs have encodings that end up close to each other in memory so that when one entry is recalled, the neighbors (similar experience entries) are cached in memory. For instance, experience entries of two similar beliefs, b and b', should be mapped to neighboring keys (indexes) in memory, as 100 and 101. Assuming that we can predict the maximum number of entries for a problem and there is enough space from them in RAM, the use of statically allocated arrays whose indexes map to experience keys would be highly efficient.
7. Whenever we need to recall an experience entry from memory, we only do the search operation if the current belief marker has hit in the memory trace. In other words, the binary `and` operation with the memory trace and current belief encoding results in one. These simple operations save us from more than 99% of the unnecessary searches in the experience memory, resulting in no experience recall. The 1% experience entries we erroneously identified as hits in the memory trace but are not stored in memory is the price we pay.
8. To mitigate the possibly damaging effect of false-positive recalls, we can shorten the duration of the actions. Two or more consecutive false positives have a close to zero chance of occurring; thus, the following action will immediately compensate for undesired deviations in the robot trajectory.
9. Keep in mind that the encoding and storing of real numbers is done with limited precision arithmetic.
10. Take advantage of the parallel nature of the BF memory.

4.6 Parallel Implementation of Recall-Planner

The Bloom filter memory of the Recall-Planner has been implemented in C++, as the baseline online solver, and evaluated for standard 2D robot navigation POMDP benchmarks.

After evaluating the execution of the Recall-Planner with **perf** [1] profiler when executing the Recall-Planner with the default parameters, we have observed the most effective way to improve it is by parallelizing the *recall(.)* method. This method takes from 2% to 87% of the total execution time, depending on the benchmark and whether past experience is reused between executions.

---

[1] `https://www.brendangregg.com/perf.html`

This high variation has been expected: if we reload the experience memory dump between experiments (episodes), there are more opportunities for recall. But if the experiment starts with en empty memory, or the new scenario is completely different to prior experience, there is nothing to recall and the BF assures the memory is not searched (the method *recall(.)* is not called) except in the rare case of a false positive. On the contrary, if the experience have many entries from varied scenarios, this translates into a higher likelihood for BF hits and calls of the *recall(.)* method to retrieve the best action from the experience memory, instead of calling the online *search(.)* method.

We have used the results from solving the benchmarks with no prior experience in memory, experience from 1 previous episode, and experience accumulated from 2 previous episodes to get these numbers. In the first case, as the agent starts with an empty experience memory, most of the execution time is dedicated to the online planner - the *solve(.)* method, while in the last case, this method takes the least of the execution time.

The *recall(.)* kernel has the highest impact on runtime when we scale up the problem (larger number of states, observations, particles, etc.), so it is our first candidate for parallelization.

Candidates with less impact are the *extract_features(.)* and *sim_hash(.)* methods. The former is used to extract the features of the belief and uses from 7.5% to 29.3% of the runtime. The latter, with 1% to 8%, encodes the belief features (float numbers) into a belief marker (set of bit vectors, each with a length of m).

For the parallel implementation of the core kernel in *recall(.)* method, we use the oneDPL (please see Fig. 4.6). From lines 21 to 24, you may see the use of oneDPL template library *for_each*. The *for_each* implementation takes as first input parameter an execution policy. This policy, is defined as in lines 19 and 20, and instantiated for a particular device by calling the function *make_device_policy*. On line 19, the constructor uses the *gpu_selector*, so this policy executes the code in brackets on the GPU. In the same way, in line 19, we use a *cpu_selector* to execute the code in parallel on the CPU.

We also make use of oneDPL zip iterators—the *start* and *end* variable (lines 17 and 18). Using zip iterators allows us to iterate over several containers at the same time. In our case, the memory entry vector, *experienceNodes* and the *similarityScores* vector. The last one stores the similarity measure between current belief and what is in memory.

As you have seen in Fig. 4.6, the is compact and it is easy to make it run in parallel on a desired device.

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <CL/sycl.hpp>
...
using namespace oneapi::dpl;
class Memory {
    std::vector<ExperienceNode> experienceNodes;
    ...
    ExperienceNode *recall(bitset<2048> *curBeliefMarker) {
        size_t size = experienceNodes.size();
        vector<int> similarityScores(size);
        //1. compute similarity score each experience entries in memory to current belief
        {
            cl::sycl::buffer bufSim(similarityScores);
            cl::sycl::buffer bufExp(experienceNodes);
            auto start = make_zip_iterator(similarityScores.begin(), experienceNodes.begin());
            auto end = make_zip_iterator(similarityScores.end(), experienceNodes.end());
            auto policy = execution::make_device_policy(queue(gpu_selector{}));
            // auto policy = execution::make_device_policy(queue(cpu_selector{}));
            std::for_each(policy, start, end, [=](auto sim_exp_entry) {
                using std::get;
                get<0>(sim_exp_entry) = get<1>(sim_exp_entry).similarityScore(curBeliefMarker);
            });
        }
        // 2. get index of entry with the highest similarity to current belief & return
             corresponding experience
        int maxOverlapID = std::max_element(similarityScores.begin(), similarityScores.end()) -
              similarityScores.begin();
        return &experienceNodes.at(bestOverlapID);
    }
    ...
}
```

Figure 4.6: Parallel implementation of the BF Memory class with oneAPI DPC++ Library (oneDPL).

For the parallel implementation of the next most compute intensive kernel, *extract_features*, we use a simple *parallel_for* and *parallel_reduce* strategy, similar to our Value Iteration implementation in TBB of the *improve policy*, and the *check convergence & update* kernels.

In this work, we do not tackle the problem of parallelizing the *solve()* method. We use the original C++ implementation of [37] and only adapt it to support continuous observation POMDPs, instead of only discrete representations.

4.7 Evaluation and Experimental Results

In this section we will:

- Present the methodology and the experimental setup [Subsections 4.7.1 and 4.7.2].
- Evaluate how the use of the BF memory affects the planning performance [Subsections 4.7.3 and 4.7.4].
- Characterize Recall-Planner and study its optimal parameters and limitations. [SubSection 4.7.5].
- Evaluate the Recall-Planner performance with respect to a state-of-the-art online planner [SubSection 4.7.6].

4.7.1 Methodology

To compare the performance of different online planning methods and implementations, we use the following metrics: the number of *time steps* the agent needs to reach the target state, the *total discounted reward* and the *execution time*. We characterize and evaluate the Recall-Planner for three benchmarks and two testbeds. We run each POMDP benchmark for 10 episodic scenarios ($S_0, S_1, ..., S_9$) on each testbed. To better understand how different amounts of experience affect planning and performance, we experiment with two test cases:

1. Test case 1 - on-policy planning (PM_0): start experiments with clean memory. The Bloom filter memory is empty at the beginning of each episode. We call these PM_0 experiments, PM stands for *Prior Memory*, and *0* indicating the empty memory at the beginning of the episode execution.
2. Test case 2 - off-policy planning ($PM_{x>0}$): start experiments with prior experience in memory. The experience memory persists between consecutive executions, i.e., a PM_1 experiment starts with the experience resulting from an PM_0 experiment, PM_2 starts with the accumulated experience of experiments PM_0 and PM_1, and so on. In practice, we write the memory contents in a file at the end of the experiment PM_x, and at the beginning of PM_{x+1}, we load the contents of PM_x in the BF memory.

The first test case is designed to reveal the possible overhead of using the BF memory on top of the baseline and if it brings any advantage with respect to the state-of-the-art. Recall-Planner starts from the same common ground with the baseline; they are both memoryless, to begin with, and are evaluated in identical conditions. Both solvers have the same parameters configuration, and we make sure that the solved POMDP has the same initial state, scenario, and seeds for random number generation for transitions the state and observation models. In this first test case, we evaluate the following metrics, given the average values obtained from 10 PM_0 experiments running Recall-Planner for 10 scenarios, $S_0, ..., S_9$, and the same for the baseline (we perform 100 experiments per planner, benchmark, and platform):

1. The average number of timesteps needed to complete an episode.
2. The average total discounted reward.
3. Average execution time per timestep needed to compute the policy, t_{policy}, to store an entry of experience entry to memory, t_{store}, and to recall (retrieve) it from memory, t_{recall}.
4. Average number of new experience entries and recalls per timestep.

The second test case is aimed to reveal the limitations of the Recall-Planner, by studying how t_{policy}, t_{store}, t_{recall}, and reward vary with the number of entries and recalls from the BF memory. We start with an empty BF memory, PM_0, and continue running experiments $PM_1, PM_2, PM_3, ...$, with new episodes corresponding to different scenarios (different initial state). This could go on until the BF memory becomes saturated, i.e., the number of entries in memory is equal or greater than the maximum capacity of the BF set, n (Section 4.4.1). Here, we evaluate the average values obtained from measurements obtained from 10 PM_0 experiments of running Recall-Planner for S_0, 10 PM_1 for S_1,..., and 10 PM_9 for S_9 (summing up to 100 experiments for each testbed and benchmark).

We use as a baseline an implementation of DESPOT-α [37]—a memoryless, state-of-the-art online planner. Our different implementations of Recall-Planner embed at their core the baseline planner plus the BF memory. In every experiment, both the baseline and Recall-Planner have set the maximum planning time parameter, t, to one second. As both are anytime planning methods, $t = 1$ second means that they return the best action they can find in memory or from forward search in one second or less. We limit the planning time to one second because it is a loose enough requirement so that the baseline can compute a good policy in soft real-time, while assuming the agent does not need higher frequency decision making. However, for real-time performance in robotics, the planner must compute a good navigation policy within a fraction of a second, and this is what we are aiming for.

4.7.2 Experimental Setting

We experimentally evaluate the Recall-Planner using three well-established benchmarks in the POMDP literature [86, 103], namely Tag, LaserTag, and RockSample. Each one models a mobile robot as a discrete POMDP that has to complete a particular task in a two-dimensional grid world, similar to a chess table.

- **Tag** - the robot must find and tag a target that runs away in a 7 × 11 grid world while avoiding fixed obstacles. The two agents start in random positions, and the robot only knows its position, one of the 29 free positions in the grid world. The robot can observe the target only if they are both in the same grid cell. The robot can choose to stay in the same position or move to an adjacent cell in one of four cardinal directions, given that an obstacle does not occupy it. Every move costs -1, a successful tag has a reward of 10, and failing to tag the target results in a -10 penalty.
- **LaserTag** - an extension of the Tag problem, with the difference that the robot does not know its position, but it is equipped with a laser range finder which allows it to localize itself. An observation produced with the range finder is composed of eight distance measurements from the robot position to obstacles in the eight cardinal directions. Each distance is generated from a normal distribution centered at the true distance of the robot to the nearest obstacle. Initially, the robot position is normally distributed over the grid map cells.
- **RockSample(N,X)** - in this case, the robot is a rover whose job is to sample X rocks randomly placed in a grid world of $N \times N$. The robot may move to an adjacent cell, sense a rock, or sample a rock at each step. Moving and sensing have a reward of 0, sampling a good rock gives a reward of $+10$, and sampling a bad rock a penalty of -10. The probability of making a correct observation when sensing a rock (whether it is a "good" or a "bad" rock) decreases exponentially with the robot's distance to the rock.

	Tag	**Laser Tag**	**RockSample(7,8)**	**RockSample(11,11)**
$\|S\|$	870	4830	12544	247808
$\|A\|$	5	5	13	16
$\|Z\|$	30	1.5×10^6	3	3

Table 4.2: Discrete POMDP problems used in our experiments: Tag, Laser Tag and Rock Sample. For each benchmark, we note its corresponding number of states $|S|$, actions $|A|$ and observations $|Z|$.

For all three benchmarks, the state, action, and observation spaces are discrete and represented as integer values. Their particular dimensions are indicated in Table 4.2. We have chosen them to cover problems with increasing state spaces, but also a combination of small, medium and large observation spaces.

In the discrete POMDPs literature, the size of the action space is typically a one-digit number and rarely goes over a dozen, but why? This is not always the case in practice; it is a limitation imposed by state-of-the-art online methods, which are exclusively based on Monte Carlo Tree Search. The search tree grows exponentially with the number of actions and observations, so memory and computation power are limiting factors for how deep the exploration can go through Monte Carlo Simulations. An ideal online method should provide timely and informed actions for decision-making. Still, even the best state-of-the-art solutions are unusable for this type of use-case scenario, becoming either unresponsive or greedy. This leaves way for future improvements.

In this phase of the research, we are using only two testbeds for different technical reasons. The compute and memory requirements of the evaluated benchmarks surpass the resources available on Odroid-XU3; therefore, we do not consider it in this evaluation. The PC running Broadwell-Desktop has been retired; thus, it cannot be included in this evaluation.

Platform	**CPU**	**GPU**	**RAM**	**TDP (Watt)**	**OS**
Kaby Lake	i5-8250U @1.60GHz 4 cores	UHD 620 @300MHz 24 EUs	12GB of DDR4	10 to 15W	Ubuntu 18.04
Tiger Lake	i7-1165G7 @2.8GHz 4 cores	Iris(R) Xe @1.3Ghz 96 EUs	16GB of LPDDR4x	12 to 28 W	Ubuntu 20.04

Table 4.3: Description of the low-power HCP for testing and evaluating our implementations. It uses oneAPI DPC++ 2021.1 Compiler (C++14) and Intel(R) NEO (Gen11) Graphics Driver.

We implement the Recall-Planner in C++ and evaluate it on the two low-power heterogeneous computing platforms described in Table 4.3. Both embed a quad-core processors and an integrated GPUs on chip. We employ the Intel VTune and Advisor tools for profiling and detecting opportunities for improving the performance of the code. We use oneAPI DPC++/C++ compiler and the oneAPI base toolkit with the oneAPI Data Parallel C++ Library (or oneDPL) to parallelize some of the more computationally expensive kernels of the BF memory, as explained in Section 4.6. oneDPL allows us to run many of the C++ STL algorithms in parallel on a CPU, GPU device, even an FPGA.

4.7.3 Preliminary Exploration and Tuning

Our first implementation of Recall-Planner is sequential. In a preliminary evaluation of this implementation for PM_0 test case (on-policy planning), we note two things by looking at Fig. 4.7.

1. The average execution time per time step of the Recall-Planner (green line) and baseline (blue line) are indistinguishable (both take the maximum time available to search for the action, t = 1 second).
2. *By carefully setting the BF memory parameters*, the Recall-Planner may converge in fewer time steps and produce a superior policy. As an example, we can see in Fig. 4.7 that the robot tags its target in 13 moves (time steps or actions) when it uses the BF memory, while it requires 18 moves with the baseline online search implementation. This is equivalent to summing up less penalty (each move has a fixed cost) and getting the reward (red line versus orange line) for reaching the goal sooner. It is important to note that this is a rare occurrence. Recall-Planner and the baseline obtain indistinguishably "good" policies in most of the experiments.

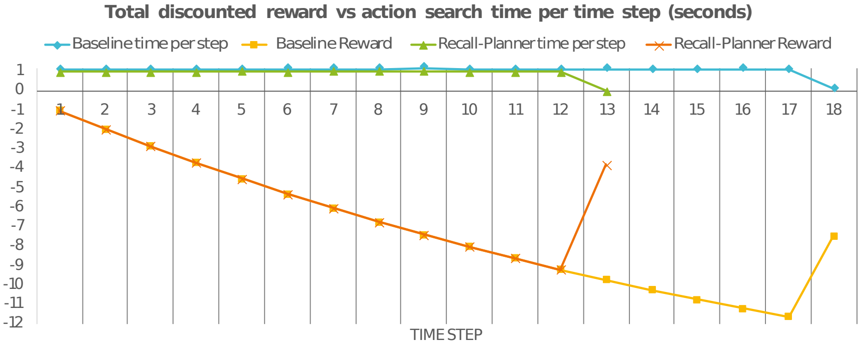


Figure 4.7: Preliminary results for Tag benchmark on Kaby Lake. We use the default values for all Recall-Planner parameters, except for N = 100, d = 50. The experiment starts with an empty BF memory (test case 1 - PM_0). The graph shows the action search time (green and blue lines, lower values are better) and total discounted reward accumulated (yellow and orange lines, higher values are better) by the robot agent in each time step for memoryless online planner (Baseline) and our proposal (Recall-Planner) with the default configuration.

Also, in the exploration stage, we have observed that the BF memory provides the policy for the current belief by recall only occasionally (once every ten timesteps). However, in every timestep, the BF memory is consulted millions of times during the belief tree simulation, in the *solve(.)* method calls, to store and recall entries. Unfortunately, the bulk of these calls happens very deep in the search tree. This type of store and recalls (deep in the search tree) bring little to

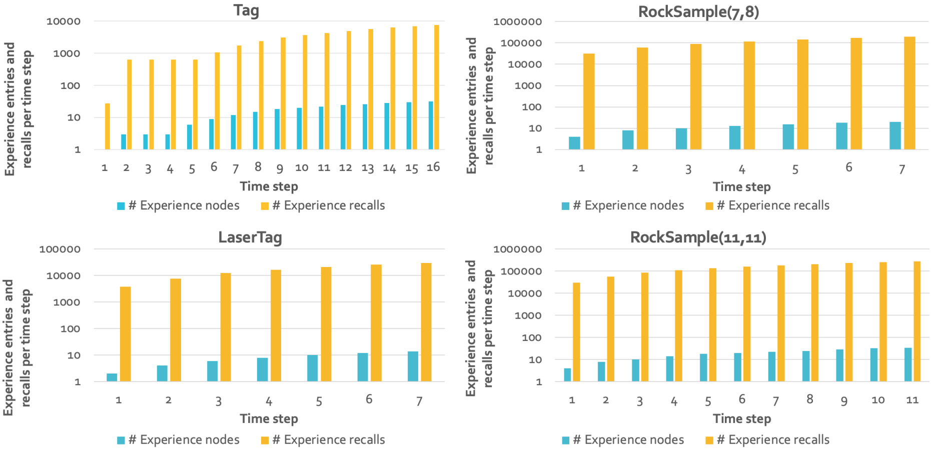
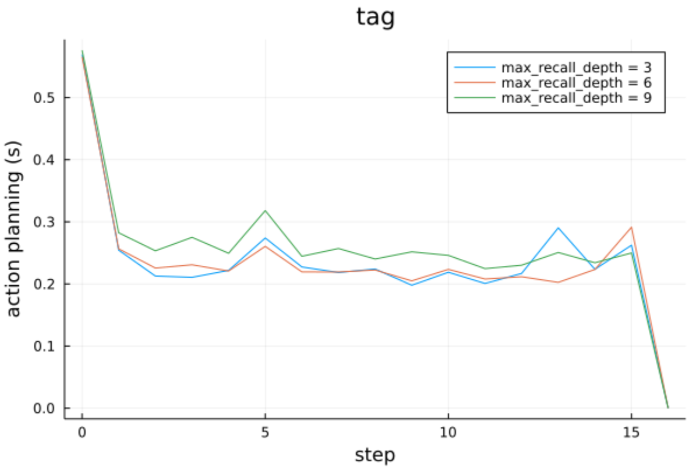


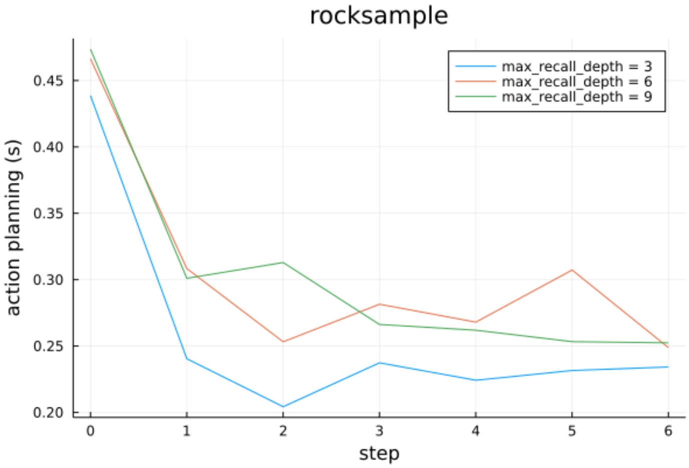
Figure 4.8: The four graphs illustrate how many times experience entry entries are recalled on average per time step (yellow bars) and the average of experience entries available for recall at the end of the corresponding time step. Here we show the measurements from Kaby Lake for Tag, Laser Tag, RockSample(7,8) and RockSample(11,11) benchmarks. Each experiment starts with an empty BF memory (PM_0 - test case 1). In the experiments, we use the default parameters for Recall-Planner (parallel implementation), and set the recall depth threshold, r_d to 3.

no benefit to the planning performance. Moreover, they saturate the BF memory with entries that are never recalled. Consequently, we have tuned the store/recall maximum depth parameter, r_d to 3. The value of 3 for this parameter is chosen experimentally and strategically to get the best planning time per timestep (see the graphs from Fig. 4.9) while reducing the overhead incurred from encoding and storing simulated experience deep in the search tree. This overhead is explored and quantified in Sec. 4.7.4.

The other reason for setting a limit to the store/recall depth during search is to maximize the number of recalls per entry in the BF memory. This is important in our design because it directly affects the second requirement (please see Sec. 4.3.2) for the experience memory design. Not setting a limit implies we have an infinitely large memory to store all simulated experience nodes from every belief tree, which is not the case, especially in the context of mobile physical agents. Limiting the store/recall depth results in a small and fast experience memory that stores only those experience entries that are likely to be recalled in the next iterations.



(a)



(b)

Figure 4.9: Two examples of how the recall depth affects the planning time in the parallel implementation when using the default planner parameters and vary only the maximum recall depth, r_d, to 3, 6, and 9. The X-axis shows the time step and the Y-axis the planning time in seconds (lower is better). In example a), setting r_d to 3 and 6 gives similar performance, while for $r\_d = 9$ the action planning time takes longer. The lower value, $r\_d = 3$, is preferred to avoid filling the BF memory with entries of simulated beliefs that are far from the root node (current belief). The farther the simulated belief from the root, the less likely it will be useful to obtain a policy in practice. In example b), we have a closeup look at the planning time and see that $r\_d = 3$ is clearly the better option. Here we show the measurements from Tiger Lake for Tag, and RockSample(7,8) benchmarks. Each experiment starts with an empty BF memory (PM_0 - test case 1).

Finally, we use a simple and effective strategy to maximize the number of recalls per timestep: we reduce the overall execution time of the *recall(.)* method with parallel computing (see 4.6). In the following evaluations, we use our most time-efficient parallel implementation of the BF Memory, which is based on the lessons learned from the previous chapter (Sec. 3.5.7). As a result of introducing these two measures—setting recall/store depth threshold to 3 and using a parallel implementation for *recall(.)* method—the number of entries stored in the BF memory in PM_0 experiments run for the four benchmarks is reduced to 12-80 entries, depending on the benchmark. At the same time, the number of recalls is kept at a maximum from depth 0 to 3. Please see Fig. 4.8 as a reference. Note that the recall and store operations on the BF memory are completely turned off for belief nodes at $d > 3$ in the decision tree exploration (in the *search(.)* method of Recall-Planner).

4.7.4 BF Memory Overhead Evaluation

We propose a thought experiment to understand how and when it might benefit us using the BF memory in combination with an online planner. For it, we assume that 1) *per timestep, it takes far less time to recall from our BF memory than performing an online search to plan the following policy* ($t_{recall} \ll t_{search}$), and that 2) there is an experience entry to be recalled for the current belief. Given this, the use of the BF memory has four possible effects on online planning.

Case 1: If the recall happens at the root node of the decision tree, it replaces the online planning call altogether for the current timestep, and the policy is available for execution as early as it can be recalled. This is the ideal use of memory, returning a plan in t_{recall}. As a counterexample, see Fig. 4.10, depicting the baseline online planner with no use of memory. In case 1, the search tree would never be constructed.

Case 2: The recall happens at a small depth in the decision tree (in the vicinity of the root node or current belief node, at a depth of 1, 2, or 3). In that case, the BF memory recall will boost the online planning by pruning the search for the belief nodes whose value can be consulted from the BF memory, replacing the node value estimation through simulation. In the best-case scenario, all direct children of the root belief ($d = 1$) are recalled from memory, each one would be labeled with the corresponding values stored in the BF memory, and the most valued policy can be provided in $|A| \times |O| \times t_{recall}$. The time is proportional with the number of actions and observations of the POMDP agent, which may still be lower than t_{search}. We illustrate this case in Fig. 4.11.

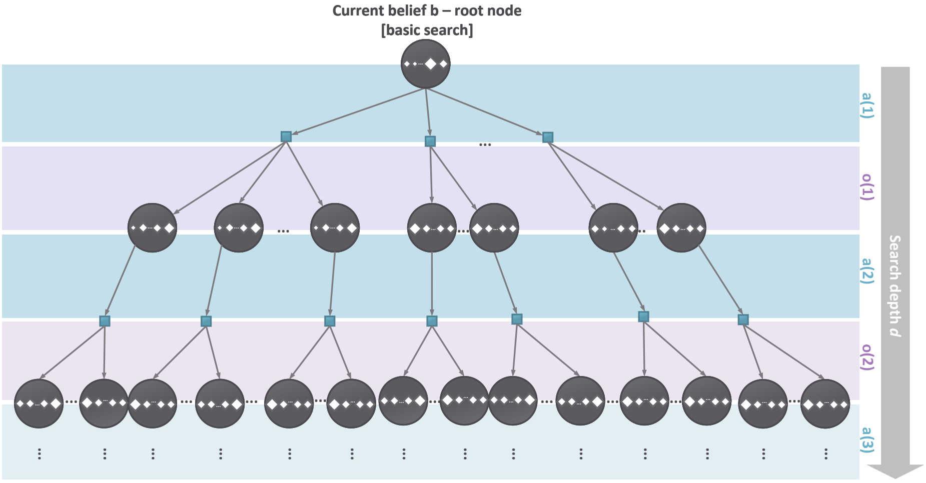


Figure 4.10: Basic online planning using forward search and no pruning. Note that all modern online planners use some form of pruning to avoid exploring the branches estimated to bring poor reward. Here we discuss a new pruning criterion that does not necessarily enter this category. For instance, it might prune the exploration both on the most and least rewarding paths. The main difference with traditional pruning techniques is that the pruned paths are not discarded when deciding the best path. Their expected reward is set from the corresponding experience entry value.

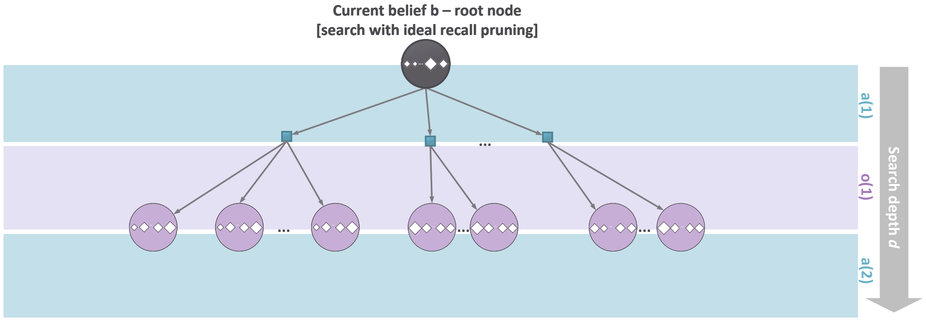

Current belief b – root node
[search with ideal recall pruning]
a(1)
o(1)
a(2)
Search depth d

Figure 4.11: Recall planning with ideal pruning factor. The forward search is pruned by recall from the experience memory for the belief nodes marked in purple.

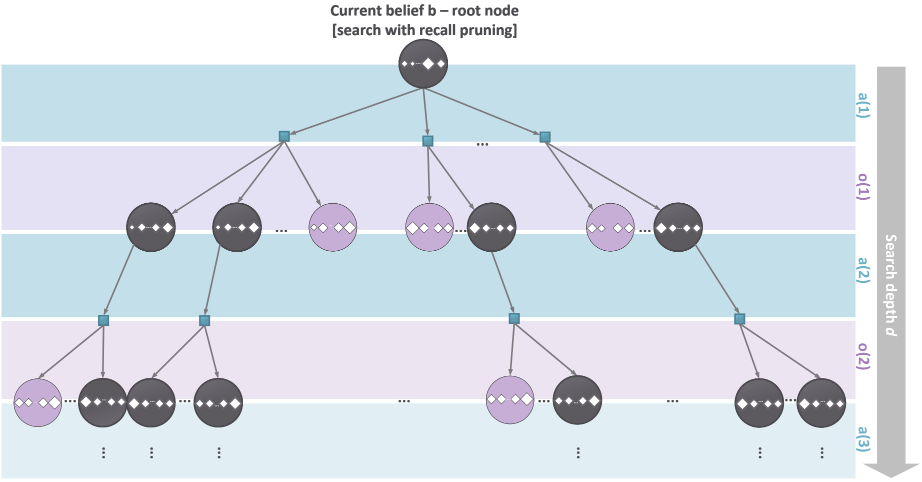


Figure 4.12: A typical use-case of Recall-Planner with recall pruning (purple belief nodes) at varying depths.

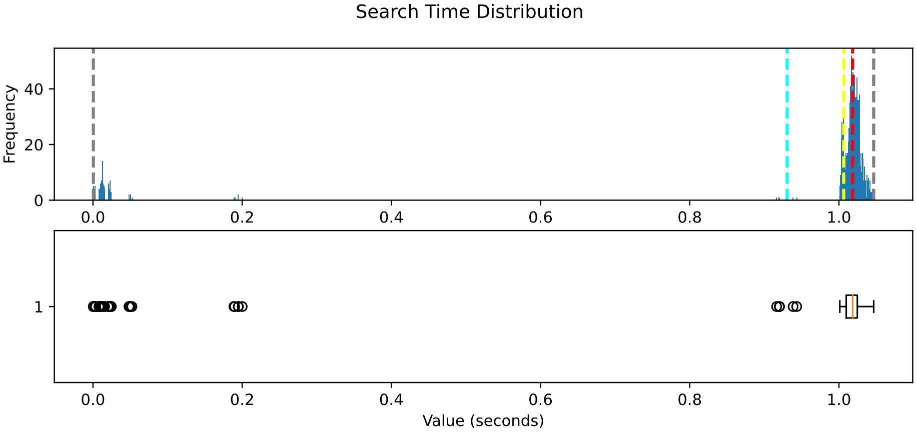


Figure 4.13: Memoryless-search execution time distribution, $t_{policy} = t_{search}$, as illustrated in Fig. 4.10. We show the time measurements in a histogram (upper graph) and box-and-whiskers plot (lower graph). We see that 99% of the values are in the range of $[0.91, 1.04]$ seconds, with some outliers on the left side of the whiskers-box plot.

Case 3: The recall happens deeper in the decision tree, in the vicinity of the leaf nodes, (e.g., at a depth ranging from 50 to 100). Increasing the planning horizon, d, the memory-less forward simulation will become cheaper relative to the combination of planning and querying the node value from the BF memory with the Recall-Planner. We expect the number of recalls to increase exponentially with the number of actions, observations and forward simulation depth, resulting in a planning time for $t_{recall+search}$ that is higher than t_{search}. Consequently, the BF memory should not be used deeper than a few levels in the forward search simulation within the online planning, for it would only hinder the performance of the already costly planning function.

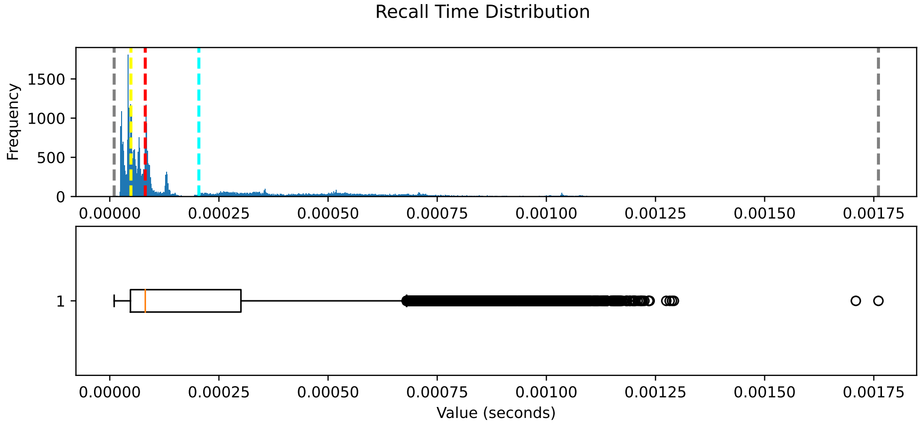


Figure 4.14: Recall execution time distribution, $t_{policy} = t_{recall}$. We show the measurements for the parallel Recall-Planner execution when the policy is obtained by recall only (no search(.) call). The data shows that 99% of the measurements are under 1 millisecond for all combinations of benchmarks and testbeds. The maximum recall time measured has a value of ≈ 0.0017 seconds (right gray bar). Note that these time measurements are exclusively from the *recall(.)* method execution (presented in Fig. 4.6), and do not account for the time needed to encode and store the recalled entries to the BF memory.

Case 4: Is a combination of cases 2 and 3, and the most likely of all cases. Here, we strategically allow only recalls up to depth $r\_d = 3$ in the decision tree and prune accordingly any belief node whose value can be recalled. All remaining (unpruned) nodes are explored to depth d using memoryless online planning. The effect of using the BF memory, in this case, needs to be explored experimentally, which we do in Section 4.7. We illustrate this case in Fig. 4.12.

Before proceeding with the evaluation of the off-policy test case ($PM_{x>0}$) of our proposal, we check experimentally if the main assumption for our thought experiment—the recall time (t_{recall}) is much smaller than the search time (t_{search})—is true. To see the bigger picture, we have generated histograms and box-and-whiskers plots with the recall time (t_{recall}), search time (t_{search}), and search plus the recall time ($t_{recall+search}$) per time step. These graphs summarize the three measurements on all benchmarks (Tag, LaserTag, RS(7,8) and RS(11,11)) on Tiger Lake for the default Recall-Planner configuration, presented in Sec. 4.4.2 and test case 1 (PM_0). Please see Fig. 4.14, Fig. 4.13, and Fig. 4.15, respectively. On the top graphs of each figure (the histograms), we mark the minimum and maximum values in gray, mean values in cyan, median values in red, and the modes in yellow. The results on Kaby Lake are equivalent.

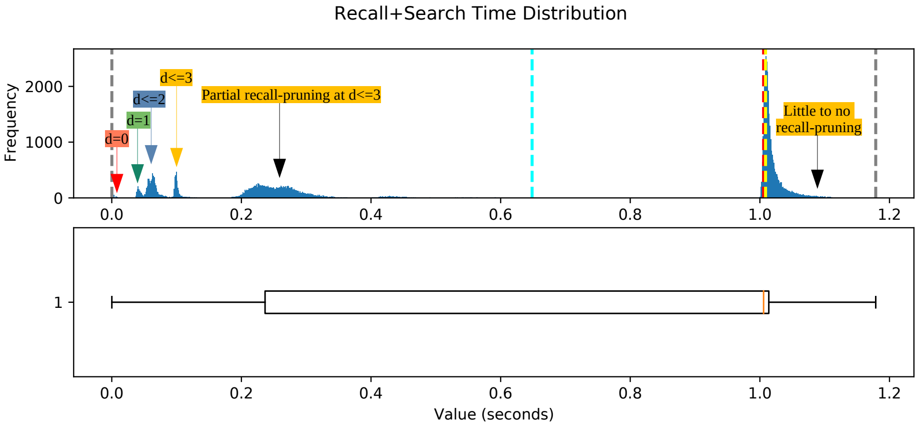


Figure 4.15: Execution time distribution for online search combined with memory recall, $t_{policy} = t_{recall} + t_{search}$. The measurements for running the online planner used in combination with BF Memory to obtain a policy spread in the range of $[0.0001, 1.17]$ seconds.

In a nutshell, these figures confirm that $t_{recall} \ll t_{search}$ is true in all cases. This can be deduced by comparing the corresponding histograms and whiskers-box-plots of online planning with (Fig. 4.15) and without (Fig. 4.13) recall. Therefore, the implications of using the proposed BF memory in combination with online planning, as presented in Cases 1-3, are also likely to be true.

The most interesting of the three figures for us is Fig. 4.15. The histogram shows a multimodal distribution, whose values we group into three regions, from left to right. Region 1 has roughly 20% of values in the range of $[0.0001, 0.1]$

seconds and has 4 modes (marked by the red, green, blue, and yellow arrows). Mode 0 corresponds to *Case 1* in our thought experiment (recall at $d = 0$, red arrow). Mode 1 corresponds to *Case 2* in our thought experiment (ideal recall-pruning at $d = 1$, green arrow). Mode 2 and 3 correspond to search with complete recall-pruning at depths $d = 2$ (blue arrow) and $d = 3$ (yellow arrow), respectively. Region 2 corresponds to *Case 4* (partial recall-pruning at $d \leq 3$) and contains roughly 30% of the total measurements, which are valued in the range of $[0.2, 0.32]$ seconds. Region 3 has approximately 50% of the measurements tightly packed in the range of $[1, 1.1]$ seconds and the samples are skewed towards 1 second. Region 3 also corresponds to *Case 4*, but it has little to no recall-pruning at $d \leq 3$, i.e., most calls to *recall(.)* method return false.

Comparing the measurement from the baseline memoryless-search in Fig. 4.13 to the Recall-Planner in Fig. 4.15, we observe that in the worst-case scenario, there is a planning time degradation of up to $130ms$ when using Recall-Planner. This value is computed by subtracting the maximum values for the two cases: $overhead = t^{max}_{recall+search} - t^{max}_{search} = 1.17 - 1.04 = 0.13\ s$. Fortunately, in 95% of the experiments, the recall overhead is under $50ms$. The maximum overhead measured here is acceptable in 2D navigation problems for indoor use-cases, where the speed of the agent is limited. However, it would be of concern if Recall-Planner were used to plan for navigating a drone in a cluttered 3D space.

4.7.5 Exploring the Limits of Recall-Planner

In this section, we focus on the effects of using experience entries from prior experiments, besides the entries stored in memory while planning in the current episode—the $PM_{x>0}$ off-policy planning test case. Following the procedure presented in the methodology (Sec. 4.7.1), we evaluate the BF memory and computing required for Recall-Planner using the following metrics: the average total discounted reward, the planning time, and the number of iterations (timesteps) needed to reach the goal for an episode, and how these values evolve in a series of episodes with cumulative memory.

Our aim here is twofold:

- To chart the limitations for BF memory size and the largest POMDP problem size solvable on a low-power platform.
- To find a recipe for which parameters of the BF work best.

To this end, we systematically and experimentally evaluate the Recall-Planner performance for each benchmark and a range of parameters. In these experiments, most parameters are permanently set to the values that give the best

BF_SIZE (m bits)	**Nr. bits set by hash (n_on)**	**Nr. unique keys in BF set (n')**	**Nr. hashes per BF (k)**	**False positive rate (p)**	**BF memory capacity (n)**	**Benchmark matching (Name ($\|S\|$))**
256	24	233	2	0.05	41	-
512	24	489	2	0.05	82	-
1024	24	1001	2	0.05	164	Tag (870)
2048	24	2025	2	0.05	328	-
4096	24	4073	2	0.05	656	-
8192	24	8169	2	0.05	1312	LaserTag (4830)
16384	24	16361	2	0.05	2624	RS(7,8) (12544)
32768	24	32745	2	0.05	5249	-
65536	24	65513	2	0.05	10498	-
131072	24	131049	2	0.05	20996	-
262144	24	262121	2	0.05	41991	RS(11,11) (247808)
524288	24	524265	2	0.05	83983	-
1048576	24	1048553	2	0.05	167966	-
2097152	24	2097129	2	0.05	335932	-
4194304	24	4194281	2	0.05	671864	-
8388608	24	8388585	2	0.05	1343727	-
16777216	24	16777193	2	0.05	2687454	-
33554432	24	33554401	2	0.05	5374909	-
67108864	24	67108833	2	0.05	10749818	-

Table 4.4: Precomputed BF parameters by plugging in the formulas from Sec. 4.4.1. We use this table to decide the configuration of the parametric BF memory for a given POMDP problem. Although we only use the configuration with an m ranging from **1024** to **262144** in the evaluation, it is interesting to note that for a configuration of $m = 256$, the BF memory guarantees a false positive rate of $p = 5\%$ only for up to 41 experience entries.

results for the baseline planner. We keep constant the number of particles used to approximate the belief state ($N = 500$), the maximum depth of search tree ($d = 90$), the maximum search time per move ($t = 1$ second), the POMDP discount factor ($\gamma = 0.95$), and a set of seeds used to obtain a deterministic sequence of random numbers, and generate identical episodes both for the baseline and Recall-Planner. Some specific parameters to Recall-Planner are also set to a fixed value experimentally, such as a maximum recall depth, $r\_d = 3$. We vary the following parameters:

1. e_n: The current episode (experiment) number in a series starting with 0 (no prior experience used). Episode 1 loads experience from the prior experiment execution (episode 0), episode 2 loads the accumulated experience from 0 and 1, and so on. We evaluate $e\_n \in [0, 1, 2, ..., 9]$.
2. m: Size of the bit vector of the BFs. We evaluate $m \in [1024, 2048, 4096, 8192, 16284, 32768, 65536, 131072, 262144]$.

For convenience in this evaluation, we have precomputed 19 combinations of

BF parameters of interest, which are based on the formulas presented in Sec. 4.4.1 and shown in Table 4.4. In the first two columns, we choose configurations for m and n_on that have a computational advantage (sizes multiples of 8 bits) and so that the BF set (the memory trace data structure in our implementation) can accommodate sufficient distinguishable entries in the BF memory. The values for m and n_on are set to guarantee a false positive rate p lower than 5%—5'th column of the table.

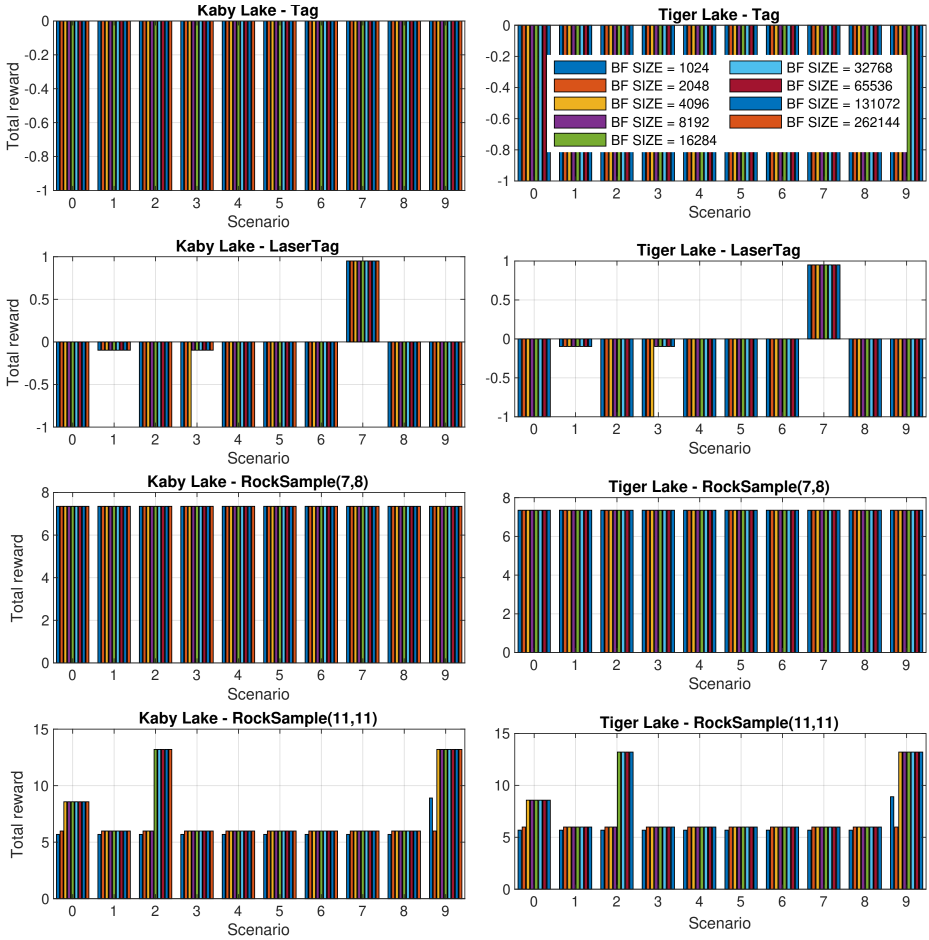


Figure 4.16: Reward exploration for different BF sizes. On the X-axis we group the experimental results by the scenario number (each scenario has different initial conditions). Higher reward values are better.

The BF will guarantee a false positive rate lower than 5% as long it contains less than n entries, which also is the real BF memory capacity—the penultimate column. As you might have noticed, n is much lower than $n' = m - n\_on + 1$—the number of unique keys that can be stored in the BF set, 3'rd column [2].

We propose a simple heuristic for choosing a configuration for the BF for a POMDP: select a BF so that $n' \geq |S|$, where $|S|$ is the number of states of the POMDP. In the last column of table 4.4, we map the BF configurations to concrete benchmarks, according to the BF n' and and POMDP $|S|$. You may consult the details of the benchmarks in Table 4.2. Nevertheless, we explore all combinations of BF size (from 1024 to 262144) and benchmark, and for the evaluated benchmarks, we find that in all cases, any BF size larger than 16284 does not bring more improvement in the planning time nor in the reward.

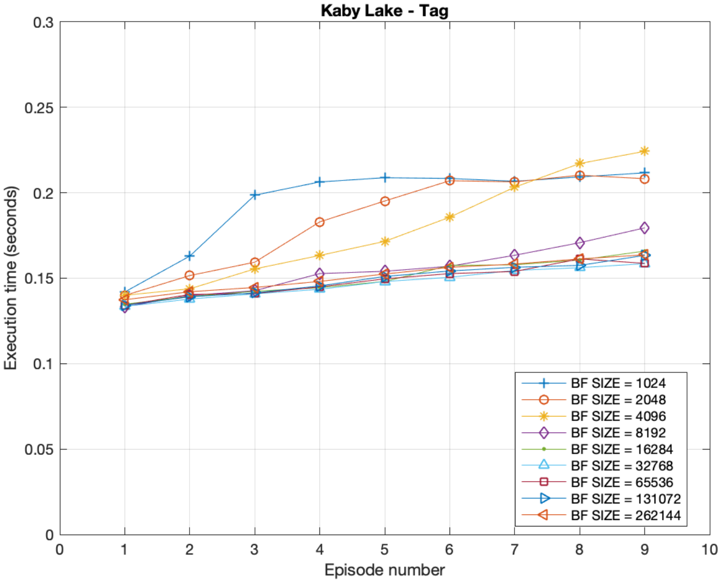


Figure 4.17: Kaby Lake - Tag: X-axis shows the episode number in the PM_x experiment and the Y-axis indicate the planning time for different BF sizes, ranging from 1024 to 262144. Lower values are better. The planning time improves as the BF size increases, but only up to a value of 16284.

---

[2] A direct implication of $n \ll m$ is that the *memory_trace* data structure in our implementation is composed of sparse bit arrays, and this could be leveraged to fit more experience entries in memory.

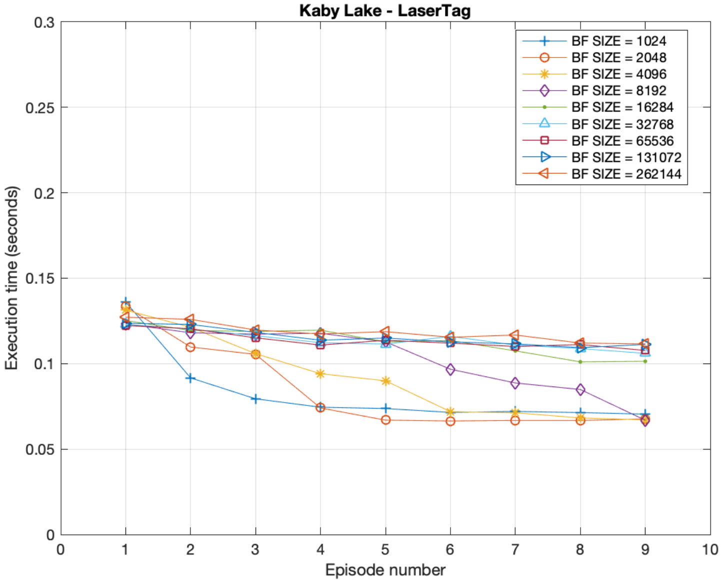


Figure 4.18: Kaby Lake - LaserTag: X-axis shows the episode number in the PM_x experiment and the Y-axis indicate the planning time for different BF sizes, ranging from 1024 to 262144. Lower values are better. For PM_1, the two smallest BF sizes, give a slightly worse planning time than all the rest, but over all PM_x experiments, these configurations (1024 and 2048) bring the best planning time performance.

For each benchmark and platform, we search for the smallest bloom filter size that gives the best reward and planning time for all scenarios. We first evaluate how the bloom filter size affects the reward (the quality of the policy) for the different scenarios and summarize the results in Fig. 4.16. Then, we evaluate how the planning time behaves and show the most interesting results in Figs. 4.17, 4.18, 4.19, 4.20. For the reward, higher values are better while for planning time, smaller values are better.

A first observation is that for the same testbed, benchmark, scenarios, and BF size, the reward is consistent, while the planning time varies slightly depending on the testbed. For instance, for Tag benchmark, the optimal BF size—the smallest BF size that gives the highest reward for all scenarios—is equal to 1024 bits, both on Kaby Lake and Tiger Lake. For LaserTag, the optimal BF size is 8192,

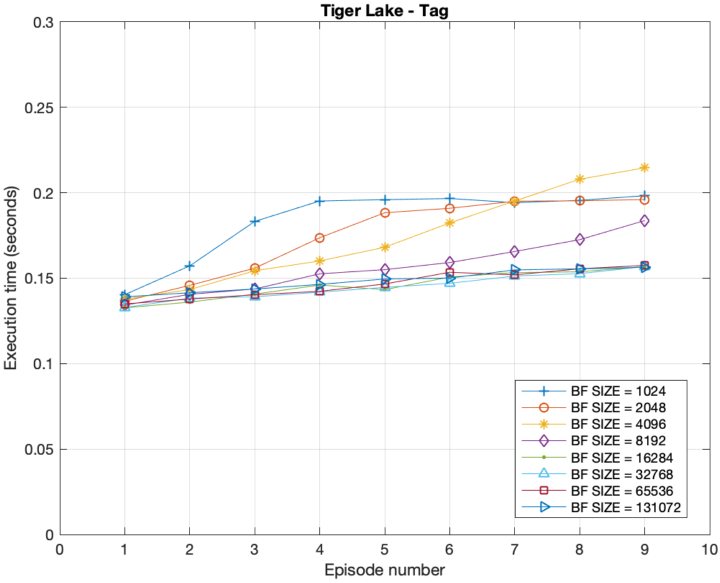


Figure 4.19: Tiger Lake - Tag: X-axis shows the episode number in the PM_x experiment and the Y-axis indicate the planning time for different BF sizes, ranging from 1024 to 131072. Lower values are better. Similar to Kaby Lake, the planning time improves as the BF size increases, but only up to a value of 16284.

for RockSample(7,8) it is 1024 bits, and for RockSample(11,11) it is 16284 bits. The optimal BF size for RockSample benchmarks is lower than predicted by the proposed heuristic, but it is spot-on for Tag and LaserTag, which have a lower number of states.

By looking at the evolution of the planning time for consecutive PM_x experiments, in Figs. 4.17, 4.18, 4.19, and 4.20, we notice similar patterns for all the benchmarks and BF size configurations on the two testbeds. Also, we see that most BF configurations lead to a stable, predictable planning time after only four episodes. For instance, the planning time settles at under 220 ms in all cases for Tag on Kaby Lake and at under 195 ms on Tiger Lake, with two exceptions: BF sizes of 4096 and 8192, which appear to find a stationary value after ten consecutive episodes. For RockSample benchmarks, the planning time results are indistinguishable for all BF sizes. In this case, the optimal BF setup is the smallest size that offers the shortest execution time for all episodes.

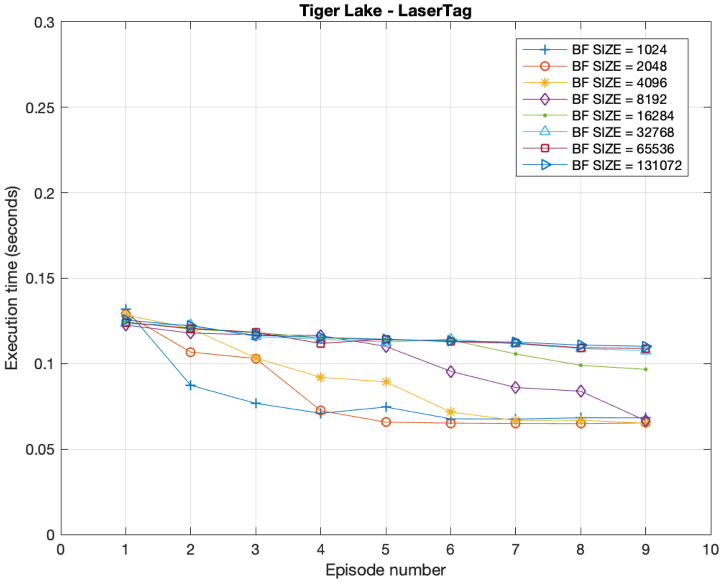


Figure 4.20: Tiger Lake - LaserTag: X-axis shows the episode number in the PM_x experiment and the Y-axis indicate the planning time for different BF sizes, ranging from 1024 to 131072. Lower values are better. Similar to Kaby Lake, for PM_1, the two smallest BF sizes, give a slightly worse planning time, but over all the PM_x experiments, BF sizes of 1024 and 2048 bring the best performance.

These results suggest that for fine-tuning the BF size for a new benchmark to obtain either the best reward, execution time, or a compromise of both, a wide range of scenarios should be explored with different BF size values, both on the left and right side of the POMDP number of states. In future work, a possible improvement of our method for usability is to find a smarter heuristic that considers more characteristics of the POMDP and reduces the work for tuning the BF size for a particular benchmark to obtain the best performance.

4.7.6 Summary of Performance Results

In the previous sections we have explored the optimal parameters of Recall-Planner—used in the experiments summarizing the results—and evaluated the limitations, gains and costs incurred from using a BF Memory in combination with a state-of-the-art online planner. From the experiments evaluated in Section 4.7.4, we can confirm there is a benefit from using a BF memory for on-policy planning, especially as a means to prune the decision tree search of an online planner and reduce the planning time per iteration. This benefit comes at a cost acceptable when planning for indoor navigation and autonomous decision making problems in general that have soft real-time constraints.

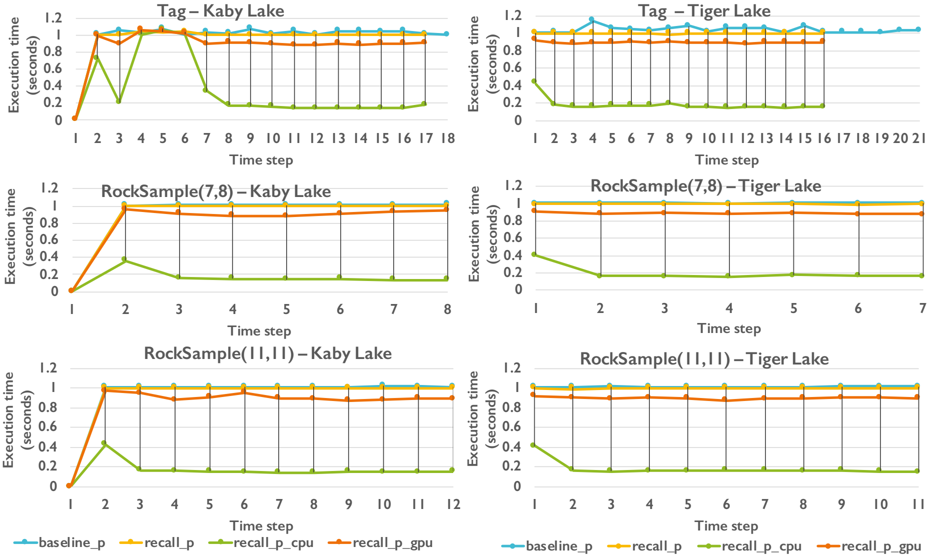


Figure 4.21: Planning time of parallel implementation. On the X-axis we use as a timescale the planning timestep. In the Y-axis we show the planning time of the Recall-Planner when parallelizing the kernel for similar experience search from Fig. 4.4. For Recall-Planner we evaluate three implementations: with a red line we show the GPU time, in green the multicore time, and in yellow the sequential time. The baseline time is shown in blue.

Here we compare the performance of a state-of-the-art planner to that of Recall-Planner for the case when we compute the similarity score kernel on the multicore versus on embedded GPU. We show the results for the case when the agent starts the execution with no prior experience in the BF memory (PM_0). In the graphs from Fig.4.21 we illustrate the overall speedup obtained by computing in parallel the kernel which finds the most similar experience entry to the current belief for three benchmarks and the two testbeds, Kaby Lake and Tiger Lake. In our parallel implementations, we use oneDPL—the oneAPI DPC++ Library [3].

---

[3] `https://spec.oneapi.com/versions/0.5.0/oneAPI/Elements/onedpl/onedpl_root.html`

Implementation \Time step	1	2	3	4	5	6	7
tag-baseline	1.0073	1.0511	**1.0297**	1.0740	**1.0097**	1.0360	1.0198
tag-bf-seq	1.0001	1.0001	1.0503	**1.0365**	1.0410	1.0001	1.0001
tag-bf-cpu	**0.7229**	**0.2056**	1.0064	1.0576	1.0204	**0.3406**	**0.1780**
tag-bf-gpu	0.9946	0.8940	1.0583	1.0477	1.0202	0.8965	0.9156
ltag-baseline	0.1001	0.1052	0.0725	0.0654	0.0555	0.0524	0.0369
ltag-bf-seq	1.0005	1.0003	1.0004	1.0003	1.0003	1.0003	1.0003
ltag-bf-cpu	**0.4787**	**0.3146**	**0.3262**	**0.2931**	**0.2866**	**0.2848**	**0.2817**
ltag-bf-gpu	1.0319	0.9176	0.8767	0.9041	0.8960	0.8963	0.8866
rs(7,8)-baseline	1.0069	1.0119	1.0093	1.0088	1.0119	1.0098	1.0164
rs(7,8)-bf-seq	1.0001	1.0001	1.0001	1.0001	1.0001	1.0001	1.0001
rs(7,8)-bf-cpu	**0.3620**	**0.1588**	**0.1477**	**0.1409**	**0.1479**	**0.1386**	**0.1388**
rs(7,8)-bf-gpu	0.9606	0.9137	0.8855	0.8789	0.9081	0.9341	0.9448
rs(11,11)-baseline	1.0083	1.0081	1.0073	1.0108	1.0079	1.0090	1.0079
rs(11,11)-bf-seq	1.0001	1.0001	1.0001	1.0001	1.0001	1.0001	1.0001
rs(11,11)-bf-cpu	**0.4287**	**0.1614**	**0.1556**	**0.1519**	**0.1481**	**0.1394**	**0.1384**
rs(11,11)-bf-gpu	0.9673	0.9480	0.8775	0.9086	0.9498	0.8930	0.8892

Table 4.5: Average execution time per time step (in seconds) for the sequential baseline online planner (*-baseline), sequential Recall-Planner (*-bf-seq), multicore (*-bf-cpu) and gpu (*-bf-gpu) implementation. The * symbol stands for the corresponding benchmarks the Tag (tag) Laser Tag (ltag), and RockSample (rs(7,8) and rs(11,11)). We have marked in bold the best execution time results for each benchmark and the default parameter values.

In Table 4.5 we outline the execution time to compute an action for each implementation for the first seven timesteps for Kaby Lake, which has the most interesting pattern of the two tesbeds. Just like in Fig. 4.21, we see that it takes a few time steps for the planning time of the Recall-Planner implementation to become stable. We observe that the Recall-Planner with multicore execution improves the search time significantly after only two time steps in almost all cases. The only exception being Tag benchmark, which takes five steps to gather enough useful experience entries to make the use of a Bloom filter memory significantly more advantageous than a memoryless online search strategy.

Summarizing our experimental results for two scenarios. First, when using the sequential implementation of the BF memory scheme (on top of the baseline) versus the memoryless baseline, the overhead is negligible, with the advantage that the agent running Recall-Planner converges faster by reaching the goal in less steps in some cases (see Fig. 4.7). Unfortunately, we were not able to find a pattern to systematically reproduce this result for all benchmarks. Second, when parallelizing the computation of the similarity search for the current belief and existing experience entries in the Bloom filter memory, we see a reduction of 3.5 to 7.5× in the overall execution time. Therefore, our parallel BF memory makes it possible for an online planner to take reasonably good actions in only a fraction of the time required by a memoryless planner. This is true only for our multicore implementation. Although, running the same kernel on the GPU, we only see marginal improvements in the planning time, of up to 10%, for the PM_0 test case.

The iGPU vs CPU performance results for PM_0 suggests that it is not justified using a CPU+iGPU heterogeneous implementation when starting the planning with an empty experience memory, given that our adaptive scheduler has approximately a 5% overhead, plus the complexity added to implementing it. But when the BF memory has a larger number of entries (hundreds of thousands), using a heterogeneous scheduler should be worthwhile exploring. For the largest benchmarks used, even in PM_9 the BF memory has only 3156 entries and the GPU only improves the planning time up to 13%. In a future work we will evaluate a heterogeneous CPU+iGPU implementation based on our oneAPI LogFit scheduler for larger POMDP benchmarks.

4.8 Conclusions

In this chapter, we propose a new design for online planning for POMDP agents. We introduce an online planner with Bloom filter memory, *Recall-Planner*, then implement and evaluate it on two low-power CPU+GPU SoCs suitable for running the online planning for navigation decision-making onboard a mobile robot.

Using multicore execution of the most computing-intensive kernel of our Bloom filter memory, we reduce the overall planning time from 3.5 to 7.5× for three representative benchmarks in the POMDP literature. This result promises new opportunities for using POMDP agents on low-power mobile platforms and in real-time use cases. With some modifications, our memory solution could enhance state-of-the-art online planning methods to reuse past decisions of similar experiences from history, compute better policies and reduce the overall cost of finding the best action to execute. In future work, we will explore the possible benefits in terms of energy consumption using a heterogeneous implementation of the Recall-Planner.

In conclusion, when the Bloom filter experience memory parameters are tuned accordingly, the computation time of planning an action is reduced from one second to a fraction of a second. Moreover, the overhead incurred from using the proposed Bloom filter memory data structure together with an online planner is negligible. This result could allow us to solve more variety of problems that need a real-time response on board low-power and consumer SoCs.

5 Conclusions

In this thesis, we pursue the goal of extending the use of intelligent (PO)MDP agents in applications for planning and decision-making for mobile robots—as described in Chapters 1 and 2. Throughout this quest, we propose new methods and ways to implement portable and efficient planners on low-power SoCs.

First, in the 3'rd chapter, we introduce a new data structure to represent the transition model of an MDP agent efficiently. Then, we explore a wide range of heterogeneous programming models and parallel implementations on CPU+iGPU SoCs, and evaluate their performance, energy efficiency and ease of programming. From our evaluation, it results that a heterogeneous planner based on DPC++ is the optimal approach to achieve both performance and productivity.

Next, we take the lessons learned from this experience and apply them in the 4'th chapter for solving problems more computationally complex than MDPs, while using the same type of low-power SoCs. Here, we propose a new method based on Bloom filter memory to enhance online planning under uncertainty for intelligent agents modeled as POMDPs. Our experimental evaluation suggests that our proposal improves the state-of-the-art in online point-based planning methods.

With this work, we show that it is feasible to implement real-time and energy-efficient solutions for decision-making under uncertainty on mobile platforms. As future work, we will first transfer the lessons learned and results from this thesis into a general-purpose ROS[1] robot navigation framework, then include a

---

[1] ROS stands for Robot Operating System, an open-source framework for robotics, used

new package for planning and decision-making in social environments based on Recall-Planner. We briefly present the follow-up plan in Chapter 6.

5.1 Summary of Contributions

The work in this thesis has been motivated by the need to find a way to efficiently implement planning and decision-making onboard service robots, such as Human Support Robot (HSR) [2]. This section summarizes the principal results, contributions, how we came about the motivation behind it, and the answer to the proposed research question.

5.1.1 Previous Work

During a brief stay at Northeastern University (Boston, US) in late 2018, we had access to an HSR mobile robot to experiment with state-of-the-art people tracking and path planning for social-aware navigation. As we struggled to make independent pieces of research for people detection and tracking, and robot localization and mapping work on top of the ROS navigation stack, we realized there was no way to run everything onboard the robot. The system would become unresponsive under the sheer amount of data and computation it had to process per second. As a result, we had to do part of the computation on a ROS master node connected by WiFi, which made everything slow and faulty when the robot was going far from the router or just moving behind the metal door of the lab.

This experience has inspired and shaped this thesis's end goal—to uncover recipes and guidelines to implement efficient planning for mobile robot navigation. We have worked towards this goal by combining methods for decision-making, reinforcement learning, and data-intensive parallel computing.

Prior to this, some work has been done to understand the fundamentals of decision-making algorithms, similarity search in multi-dimensional spaces, and optimizing implementations using parallel computing and heterogeneous programming models.

The results of this stage have been presented at three conferences:

1. Cáncer Gil, J., Escuín Blasco, C., Constantinescu, DA., Pérez Pavón, B.,

---

and backed up by a global open-source community that contributes to make robots better and accessible to everyone.

[2] https://robots.ieee.org/robots/hsr/

Canales Mayo M., Three is not a crowd: a CPU-GPU-FPGA K-means implementation. HiPEAC Computing Systems Week 2017, Zagreb, Croatia, April 2017 (International Conference)

2. Constantinescu, DA., Navarro, A., Fernández Madrigal, JA., Asenjo, R. Optimization of a decision-making algorithm for heterogeneous platforms. XXVIII Edición de las Jornadas de Paralelismo, Spain, September 2017. (National Conference)
3. Constantinescu DA., Rohra A., Padir T., Kaeli D. Path Planning for Socially-Aware Humanoid Robots. RISE Expo 2019, Northeastern University, Boston, USA, April 2019. (International Conference)

5.1.2 Stage 1: Solving Large MDPs Optimally on SoCs

In this stage, we provide a generalizable approach for computing an optimal policy with Value iteration (VI) through the use of TBB, OpenCL, and oneAPI [116, 58, 51] parallel programming models and advanced load balancing techniques for concurrent execution on the CPU and GPU embedded on chip [80].

For it, we implement and evaluate three heterogeneous CPU+GPU schedulers based on oneAPI programming model, using VI as a case study. Additionally, we compare oneAPI with the canonical framework, OpenCL, in terms of performance, energy efficiency, and programmability. Our evaluation discusses the impact of the abstraction penalties due to the programming model approach and the scheduling strategy. It provides guidelines to help programmers select the appropriate programming model and scheduling strategy for MDP-based solutions suitable for low-power platforms. Also, we demonstrate how to assess the Pareto-optimal implementation of a standard decision-making algorithm in terms of energy and time performance according to the problem size, platform resources, and optimization criteria for a diversity of platforms.

Overall, we explore fourteen parallel strategies for implementing the VI algorithm and analyze both their computational speed-ups and energy consumption. The implementations use open parallelization technologies to increase generality. Moreover, our evaluation has been extended to a representative set of heterogeneous computing platforms, varying from low-power embedded systems to medium-power systems, with thermal design power ranging from 4W to 65W. The selected testbeds feature memory ranging from 2GB to 32 GB and a wide range of computing capabilities that make them good candidates to be used for mobile robot navigation use-cases and other mobile decision-making applications.

The main result of this analysis is that the solution with the minimum execution time does not always achieve the minimum energy consumption. Also, both execution time and energy consumption are reduced on all studied platforms by exploiting heterogeneous scheduling strategies that allow the simultaneous execution of work in all devices (CPUs and GPU). We validate our results statistically through ANOVA.

Also, we optimize the MDP representation by leveraging the commonly sparse nature of the agent-world interaction model (MDP model) in real-world problems and propose the 3D-lite-CSR format. This enables us to execute large problems on low-to-medium power SoCs.

Finally, we build a set of MDP benchmarks for indoor mobile robot navigation, based on modeling the CRUMB robot-environment interaction (see Fig. 3.4a) in a physically realistic simulation [34].

The results in this stage have been published in two journal articles and presented at two international conferences:

1. Constantinescu DA., Navarro A., Corbera F., Fernández-Madrigal JA., Asenjo R. Solving Large-Scale Markov Decision Processes on Low-Power Heterogeneous Platforms, 19th International Conference on Computational and Mathematical Methods in Science and Engineering, Costa Ballena, Cádiz (España), Universidad de Cádiz, 2019. (International Conference)
2. Constantinescu DA., Navarro A, Fernández-Madrigal JA., Asenjo R. Performance evaluation of decision making under uncertainty for low power heterogeneous platforms. Journal of Parallel and Distributed Computing. November 2019. DOI:10.1016/j.jpdc.2019.11.009 (JCR T1/Q1 Journal)
3. Constantinescu, DA., Navarro, A., Corbera, F., Fernández-Madrigal, JA., Asenjo, R. Efficiency and productivity for decision making on low-power heterogeneous CPU+ GPU SoCs. The Journal of Supercomputing, 1-22. March 2020. DOI:10.1007/s11227-020-03257-3 (JCR T1/Q2 Journal)
4. Constantinescu, DA., Asenjo, R. Boosting Productivity of Decision-Making with oneAPI-based Heterogeneous Schedulers on SoCs. oneAPI Developer Summit 2020, November 2020. (International Conference)

5.1.3 Stage 2: Online Planning for POMDPs on SoCs

In this stage, we extend the state-of-the-art in online point-based planning for POMDP agents to support off-policy planning—the ability to use planning experience from prior execution scenarios. We propose and design a Bloom filter experience memory for this purpose. Then, we introduce the Recall-Planner algorithm to show how to use this experience memory to enhance any on-policy—

memoryless—online planner.

The proposed algorithm is generic enough to be transferable to a multitude of POMDP online planners, such as POMCP [100] and POMCPOW [107]. We exemplify its use with DESPOT [105, 37], a state-of-the-art point-based online planner that has no memory mechanism, which allows us to evaluate the benefits and limitations of our proposal experimentally.

We provide an efficient implementation of the Bloom filter memory and evaluate the performance of Recall-Planner (Bloom filter memory + baseline planner) for four POMDP benchmarks and two SoC platforms, and compare it to DESPOT (the baseline planner). Our results show that the planning time per timestep is sufficiently reduced to make Recall-Planner usable to make decisions in real-time applications while running all the code onboard a mobile platform. In contrast, the baseline alone could not achieve similar performance for the same test cases.

Additionally, we compile a taxonomy of POMDP literature to identify which methods could work best for online planning in real-time onboard a mobile robot. This result could help others navigate the principal axes of research in POMDP research more easily.

The results in this stage have been presented at two conferences, an IEEE magazine, and a journal article will be submitted for review by the end of June 2022:

1. Constantinescu, DA., Asenjo, R., Navarro, A., Fernández Madrigal, JA., Cruz-Martin, A., Enhancing Online Planning under Uncertainty via Bloom Filter Based Memory. Jornadas SARTECO 20/21, pp. 327-335. Spain, September 2021. (National Conference)
2. Constantinescu, DA., Asenjo, R., Enhancing Online Planning under Uncertainty via Bloom Filter Based Memory. oneAPI Developer Summit at SC21, November 2021. (International Conference)
3. Constantinescu, DA., Asenjo, R., Navarro, A., Fernández Madrigal, JA., Cruz-Martin, A., Enabling Easier Programming of Machine Learning Algorithms on Robots with oneAPI Toolkits. IEEE Computer Society, March 2022. (IEEE Magazine)
4. Constantinescu, DA., Navarro, A., Fernández Madrigal, JA., Asenjo, R., Cruz-Martin, A. Accelerating on-line POMDP-based decision making for low power platforms using Bloom filters. IEEE Transactions on Cybernetics, June 2022. (JCR T1/Q1 Journal, under review)

5.1.4 Answer to Research Question

Here we answer the Research Question introduced in Chapter 1:

Is it possible to solve large-scale decision-making problems on mobile consumer platforms, despite their inherent limited computing, memory, and energy resources?

Our experimental evaluations show that the answer is positive, at least for the proposed use-case scenarios. We are now much closer to a decisive yes than at the beginning of this Thesis. However, there is still work to be done to make these results usable in practice by roboticists, hobbyists, and researchers worldwide, that are not necessarily experts in parallel computing. Next, we outline the following steps in this direction.

6 Future Work: Decision Making for Real-World Scenarios

Realistic use-case scenarios for sequential decision-making with physical agents such as mobile robots are challenging to simulate. In this thesis, we have experimented mostly with simulated use-case scenarios because we did not have a modern robot at our disposal until recently. As a follow-up to this research, we will work on decision-making and planning onboard physical robots for social-aware indoor navigation. Our next target is to optimize the ROS navigation stack and enhance it with a new feature for social-aware navigation based on Recall-Planner. We aim to provide a power-efficient and real-time navigation decision-making solution for ROS. In the following two sections, we draft a modular architecture for this purpose and a practical use-case for Recall-Planner.

For any of these robotic applications to be feasible, we need a base navigation layer that is safe (for the robot, environment, and people), social-aware (can detect people, their intention, and make navigation decisions accordingly), energy-efficient (for increased autonomy), and responsive (real-time navigation and decision making).

6.1 Application Architecture Proposal

We already see mobile robots working in constrained and controlled environments, such as assembly lines and logistics. However, we are far from having mobile robots run freely in social settings, both indoors and in open spaces. Some appealing uses of social mobile robots include:

- Remote medical attention and care.
- Companion robot.
- Self-service and room service in museums, hotels, and restaurants.
- Helping with tasks of carrying and delivery.

In the next stage of this research, we will work on optimizing a generic social-aware navigation architecture based on Robot Operating System (ROS) navigation stack, and on integrating our online POMDP planner as a plugin for local navigation.

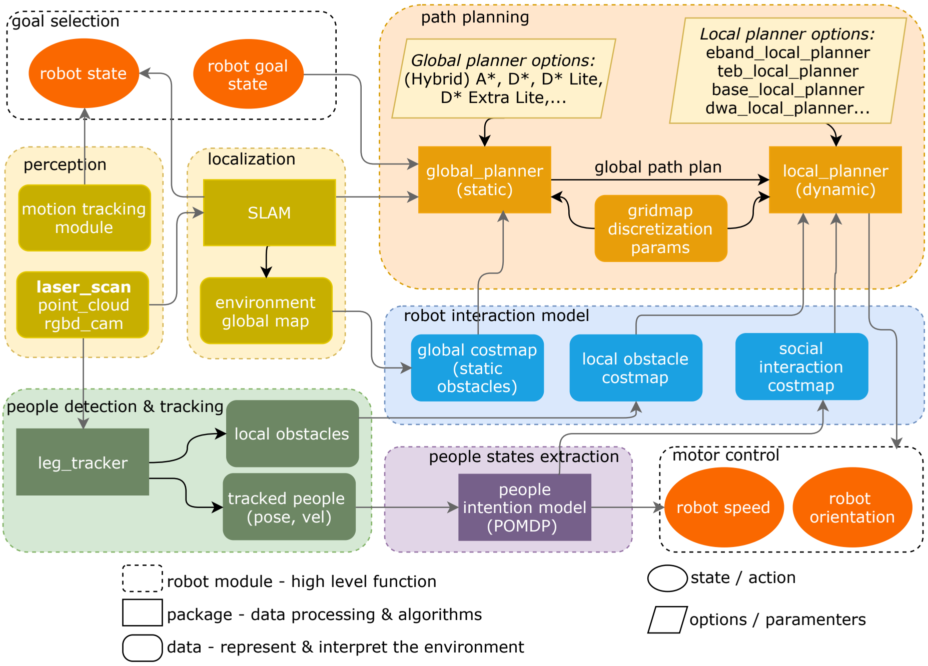


Figure 6.1: ROS navigation stack proposal.

As we can see in Fig. 6.1, the navigation stack has a modular architecture, composed of the following high-level modules:

- *Goal selection* – keeps track of the current state and goal state of the robot.
- *Perception & localization* – used to estimate the current state of the robot, that is, its location in the world and relative to its target.

- *Path planning* – used to make fine grain and coarse grain navigation decisions, given the current state, goal, and information from the environment.
- *Robot interaction model* – models the navigation constraints in the environment using costmaps. These costmaps (local, global, social), overlap on top of the global map generated with SLAM and ponder how costly it is to navigate in certain areas. Free spaces in the map have a cost close to zero, while regions where an obstacle (person, wall) or future path of a person may collide with the robot, have a very high cost.
- *People states extraction* – models the intentions of people nearby the robot.
- *Motion control* – executes motor control commands (orientation, speed), given the trajectory computed in the path planning module and intentions of nearby people.

A concerning fact is that in realistic scenarios, we could easily deplete all onboard computing resources of the robot with just one of the core modules (localization/people detection & tracking/path-planning/ tracking people states). Our task here is to identify optimization opportunities that would help make social-aware navigation planning possible onboard, while still leaving some room for the specific application that makes use of it.

The most obvious application of the lessons learned and methods proposed throughout this work are in optimizing the modules for *perception* and *localization* and *people detection & tracking* for real-time and energy efficiency with heterogeneous computing. Existing implementations of these modules in ROS are normally sequential and do not account for energy use.

The *perception and localization* module answers the “where am I?” based on sensory data from a laser scanner, point cloud sensor, or an RGB-D camera. This data is used by a Simultaneous Localization And Mapping library (SLAM) to build a map of the environment, and at the same time, localize the robot in it. The same data is used as an input in *people detection & tracking* module, which allows the robot to distinguish static and moving obstacles from people.

Other uses of the work methods presented in this thesis, especially on the modeling problems for decision making and planning under uncertainty, are in the *people states extraction* and *path planning* modules to enhance a robot agent to make socially-aware navigation decisions.

For navigation in social environments, it is essential to differentiate people from obstacles to program different behaviors for the robot while navigating in their proximity. Some fundamental behaviors a social-aware robot may incorporate are:

- Assuring a minimum safety distance that is comfortable for people, especially in the case when the robot task does not directly dictate close interaction with the person (e.g., feeding, delivering an object);
- Implement cultural and courtesy habits: passing on the right side "lane" when crossing paths with a person; make room for the person to pass or stay still on the side when crossing paths with a person in a narrow corridor;
- Driving slightly slower nearby people to make them feel safe.

Essentially, the perception and localization module estimates the state of the robot. Given the *current state* (where am I?), a *goal state* (where am I going?), and a map of the environment, the robot has the prerequisites to compute a navigation plan. The plan is then executed through motion commands. Every step, new observations about the environment are made, the robot belief state is actualized, the local plan is recomputed accordingly, then executed again, and so on, until the goal is reached.

We list the robot requirements for the proposed use-case:

- Scanning Laser Rangefinder (e.g., LIDAR, Hokuyo UST-20LX) or/and RGB-D camera sensor (e.g., Xtion PRO LIVE, Kinect) and/or point cloud sensor.
- Low-power and high performance on-board computing capabilities.
- IMU motion tracking module.
- ROS compatible.
- Microphone and speakers (optional).

We consider three robot platforms suitable for prototyping the social navigation stack and other human-robot interaction applications:

- Human Support Robot - a highly versatile mobile robot for medical applications, equipped with all the required sensors and more. This work proposal was initially planned with this robot in mind, but we no longer have access to it, and it is prohibitively costly.
- TurtleBot 2/3 - both versions comply with the bare minimum requirements for the use-case and are relatively affordable. We currently have access to a Turtlebot 3.
- AgileX Scout - makes an ideal compromise between price and features for our follow up work, and fortunately, we now have access to one (Fig. 6.2).

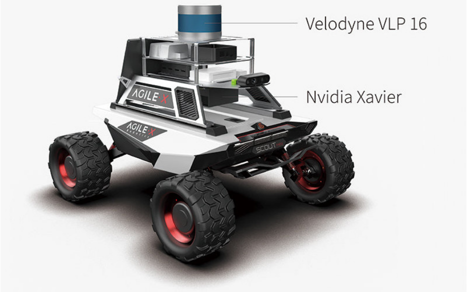


Figure 6.2: AgileX AI and mobile robot research kit.

6.2 Modeling People Intentions as a POMDP for Social-Aware Navigation

The desired behavior for social robot navigation has to comply with a simple set of rules:

1. Do not collide.
2. Do not bother people by making them stop, change direction, or speed. Have a set of courtesy responses.
3. Follow the least costly path to reach the target, which may be a place, or a moving person, while complying with points 1 and 2.

With careful configuration and an accurate global map, the navigation stack does the job of not colliding reasonably well and does a satisfactory job with autonomous navigation, too. However, the belief inferences from sensor readings are reliable only if the world is not too symmetrical. The tricky part is “guessing” what the people nearby will be doing in the immediate future and planning accordingly.

We define this problem as a POMDP that can be solved with *Recall-Planner*—our proposal from Chapter 4—to obtain a plan for navigation based on people’s intentions. The goal of the planner is to learn social behavior for safe, social-aware navigation.

States (continuous) – the POMDP keeps track of the robot inner state and the relative state of a number of people, NP that are closest to it. The robot state tracks its velocity, $\vec{V}_{ego}$, (i.e., V_x, V_y, and ω). The state of the tracked persons is stored in a list of objects with information about the tracked people, $trackedPeople$. A tracked person is represented in the robot state by an id, relative distance, and orientation with respect to the robot. If one of the tracked people is the target (goal), the state of the POMDP also includes the person id from the *trackedPeople* list.

Actions (discrete) – set a way-point in space (at a distance smaller than the distance between the robot and the target) that leads the robot to either stay still or move towards one of the cardinal directions: towards the target, given that the target person always indicates the North of the robot. The action lasts, t, a sufficiently long time, ensuring that the action has the desired effect. A local navigation planner controls the execution of the action and we have two options. The straightforward option is to use a ROS local planner that implements the Trajectory Rollout, and Dynamic Window approaches for (local) robot navigation on a plane. Other ROS packages for local navigation in ROS are *carrot_planner* and *dwa_local_planner*. Alternatively, we could adapt our MDP model for robot navigation in dynamic indoor scenarios—presented in Chapter 3—for local navigation together with a *Value Iteration* solver.

Observation (discrete) – the robot is equipped with an IMU which measures the robot instant velocity. We use a LIDAR to measure distances to obstacles as input for the leg_tracker package. Leg_tracker publishes a list with the positions, orientations, and ids of people close to the robot. This list is used as an observation in the POMDP. Every person tracked has associated a confidence parameter. A Kalman filter keeps track of each person from one observation frame to another.

Apéndice A
Resumen en español

El objetivo de este trabajo es hacer que sea factible y fácil de implementar soluciones para la toma de decisiones y la planificación autónoma bajo incertidumbre en plataformas móviles de bajo consumo. El fin es usar estas soluciones para aplicaciones prácticas, como la conducción autónoma y la robótica de servicios, que deben ejecutarse en plataformas móviles SoC. Estas plataformas, además de tener restricciones de ejecución en tiempo real, suelen tener pocos recursos computacionales y de memoria disponibles. El principal desafío es encontrar el modelo de programación heterogénea con la menor complejidad para codificar soluciones de problemas de toma de decisiones, que además cumplan con los requisitos de ejecución y eficiencia energética. Nuestra propuesta utiliza estrategias de computación heterogéneas de bajo consumo basadas en el modelo de programación oneAPI y estructuras de datos dispersas, las cuales nos permiten resolver problemas de toma de decisión con millones de estados en tiempo real a bordo de sitemas SoC de bajo consumo.

En la primera parte de la tesis, comparamos tres estrategias de programación heterogénea para ejecutar código paralelo en SoC CPU+iGPU. Evaluamos su rendimiento en un conjunto de benchmarks para planificar secuencias de acciones para el caso de uso de navegación de un robot móvil. Los benchmarks calculan un plan de navegación óptimo a través del algoritmo de Value Iteration—un método fundamental para encontrar políticas óptimas en la toma de decisiones bajo incertidumbre—lo que permite que un agente inteligente modelado como proceso de decisión de Markov (MDP) actúe de forma autónoma. Nuestros resultados experimentales muestran que las implementaciones basadas en el modelo de programación oneAPI son hasta 5× más fáciles de programar que las basadas en OpenCL, mientras que incurren en solo de 3 a 8% en el consumo de energía y en rendimiento.

En la segunda parte, transferimos las lecciones aprendidas de la optimización del algoritmo Value Iteration a un marco de toma de decisiones autónomo más complejo—los procesos de decisión de Markov parcialmente observables (POMDPs). A diferencia de los MDP, los POMDPs tienen en cuenta todas las fuentes de incertidumbre en la interacción del agente con el entorno. Proponemos un nuevo método para la planificación online bajo incertidumbre para POMDPs, Recall-Planner, que supera a los planificadores en línea de última generación para un conjunto conocido de benchmarks de navegación en la literatura de los POMDP.

Esta investigación demuestra que es factible resolver procesos de decisión de Markov (parcialmente observables) a gran escala y en tiempo real utilizando plataformas CPU+iGPU heterogéneas de bajo consumo. Podemos mejorar tanto el rendimiento como la productividad si seleccionamos cuidadosamente la estrategia y el modelo de programación. En concreto, destacamos que el modelo de programación oneAPI crea nuevas oportunidades para mejorar la productividad, el rendimiento y la eficiencia en sistemas heterogéneos de bajo consumo.

Bibliography

[1] Open Robotics. ROS: Open Source Robotics Foundation, Consulted on October 24, 2020.

[2] Giorgos Apostolikas and Spyros Tzafestas. Improved qmdp policy for partially observable markov decision processes in large domains: Embedding exploration dynamics. *Intelligent Automation & Soft Computing*, 10(3):209–220, 2004.

[3] Karl Johan Åström. Optimal control of markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10(1):174–205, 1965.

[4] Augusto Cesar Espíndola Baffa and Angelo EM Ciarlini. Modeling pomdps for generating and simulating stock investment policies. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2394–2399, 2010.

[5] Haoyu Bai, Shaojun Cai, Nan Ye, David Hsu, and Wee Sun Lee. Intention-aware online pomdp planning for autonomous driving in a crowd. In *2015 ieee international conference on robotics and automation (icra)*, pages 454–460. IEEE, 2015.

[6] Haoyu Bai, David Hsu, Wee Sun Lee, and Vien A Ngo. Monte carlo value iteration for continuous-state pomdps. In *Algorithmic foundations of robotics IX*, pages 175–191. Springer, 2010.

[7] R. Barber, J. Crespo, C. Gomez, A.C. Hernamdez, and M. Galli. *Mobile Robot Navigation in Indoor Environments: Geometric, Topological, and Semantic Navigation*, chapter 5, pages 393–640. Intech Open, 3 2019.

[8] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[9] Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 2. Athena Scientific, 3 edition, 2007.

[10] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms*, pages 684–695. Springer, 2006.

[11] Richard J Boucherie and Nico M van Dijk, editors. *Markov decision processes in practice.* Springer, 2017.

[12] Craig Boutilier and David Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1168–1175. Citeseer, 1996.

[13] Dumitru Brinza, Matthew Schultz, Glenn Tesler, and Vineet Bafna. Rapid detection of gene–gene interactions in genome-wide association studies. *Bioinformatics*, 26(22):2856–2862, 2010.

[14] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet mathematics.* Citeseer, 2002.

[15] Alex Brooks, Alexei Makarenko, Stefan Williams, and Hugh Durrant-Whyte. Parametric pomdps for planning in continuous state spaces. *Robotics and Autonomous Systems*, 54(11):887–897, 2006.

[16] Panpan Cai, Yuanfu Luo, David Hsu, and Wee Sun Lee. Hyp-despot: A hybrid parallel algorithm for online planning under uncertainty. *arXiv preprint arXiv:1802.06215*, 2018.

[17] Kenny AQ Caldas, Júnior AR Silva, Felix M Escalante, V Grassi, Marco H Terra, and AG Siqueira. A comparison between a simulated and a real mobile robot path tracking application using v-rep. *XIII Simpósio Brasileiro de Automação Inteligente*, 2017.

[18] Anthony Cassandra. pomdp-solve software, version 5.3, 2011.

[19] Anthony R Cassandra, Leslie Pack Kaelbling, and Michael L Littman. Acting optimally in partially observable stochastic domains. In *Aaai*, volume 94, pages 1023–1028, 1994.

[20] Anthony R Cassandra, Michael L Littman, and Nevin Lianwen Zhang. Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. *arXiv preprint arXiv:1302.1525*, 2013.

[21] Sanaa Chafik and Cherki Daoui. A modified policy iteration algorithm for discounted reward Markov decision processes. *International Journal of Computer Applications*, 133(10):28–33, January 2016.

[22] Krishnendu Chatterjee, Martin Chmelik, and Mathieu Tracol. What is decidable about partially observable markov decision processes with ω-regular objectives. *Journal of Computer and System Sciences*, 82(5):878–911, 2016.

[23] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39. Citeseer, 2004.

[24] Suho Cho, Seungho Cho, and Kin Choong Yow. A robust time series prediction model using pomdp and data analysis. *Journal of Advances in Information Technology (JAIT)*, 2017.

[25] Jack Collins, Shelvin Chand, Anthony Vanderkop, and David Howard. A review of physics simulators for robotic applications. *IEEE Access*, 9:51416–51431, 2021.

[26] Denisa-Andreea Constantinescu. Optimization of a decision making algorithm under uncertainty for heterogeneous platforms. Master's thesis, Universidad de Málaga, 2017.

[27] Denisa-Andreea Constantinescu, Angeles Navarro, Francisco Corbera, Juan-Antonio Fernández-Madrigal, and Rafael Asenjo. Efficiency and productivity for decision making on low-power heterogeneous cpu+ gpu socs. *The Journal of Supercomputing*, pages 1–22, 2020.

[28] Silvia Coradeschi, Amadeo Cesta, Gabriella Cortellessa, Luca Coraci, Cipriano Galindo, Javier Gonzalez-Jimenez, Lars Karlsson, Anette Forsberg, Susanne Frennert, Francesco Furfari, Amy Loufti, Andrea Orlandini, Filippo Palumbo, Federico Pecora, Ales Stimec, Stephen Von Rump, Jonas Ulberg, and B. Otslund. GiraffPlus: a system for monitoring activities and physiological parameters and promoting social interaction for elderly. In *Human-Computer Systems Interaction: Backgrounds and Applications 3*, pages 261–271. Springer, 2014.

[29] Francisco Corbera, Andrés Rodríguez, Rafael Asenjo, Angeles Navarro, Antonio Vilches, and María J Garzarán. Reducing overheads of dynamic scheduling on heterogeneous chips. *arXiv preprint arXiv:1501.03336*, 2015.

[30] Biplob Debnath, Sudipta Sengupta, Jin Li, David J Lilja, and David HC Du. Bloomflash: Bloom filter on flash-based storage. In *2011 31st International Conference on Distributed Computing Systems*, pages 635–644. IEEE, 2011.

[31] Tarek A El-Moselhy, Ibrahim M Elfadel, and Luca Daniel. A hierarchical floating random walk algorithm for fabric-aware 3d capacitance extraction. In *2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers*, pages 752–758. IEEE, 2009.

[32] Keren Erez, Jacob Goldberger, Ronen Sosnik, Moshe Shemesh, Susan Rothstein, and Moshe Abeles. Analyzing movement trajectories using a markov bi-clustering method. *Journal of computational neuroscience*, 27(3):543–552, 2009.

[33] Juan-Antonio Fernández-Madrigal. *Simultaneous Localization and Mapping for Mobile Robots: Introduction and Methods: Introduction and Methods.* IGI global, 2012.

[34] Juan-Antonio Fernández-Madrigal, Ana Maria Cruz-Martin, Marina Aguilar-Moreno, and Iván Fernández Vega. CRUMB: Cognitive-robotics-supporting mobile base, Consulted 1st of August, 2019.

[35] Johannes Fischer and Ömer Sahin Tas. Information particle filter tree: An online algorithm for pomdps with belief-based rewards on continuous domains. In *International Conference on Machine Learning*, pages 3177–3187. PMLR, 2020.

[36] Amalia Foka and Panos Trahanias. Real-time hierarchical pomdps for autonomous robot navigation. *Robotics and Autonomous Systems*, 55(7):561–571, 2007.

[37] Neha P Garg, David Hsu, and Wee Sun Lee. Despot-α: Online pomdp planning with large state and observation spaces. In *Robotics: Science and Systems*, 2019.

[38] Shahabeddin Geravand and Mahmood Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 57(18):4047–4064, 2013.

[39] Carlos H González and Basilio B Fraguela. A generic algorithm template for divide-and-conquer in multicore systems. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 79–88. IEEE, 2010.

[40] Geoffrey J Gordon. *Approximate Solutions to Markov Decision Processes.* PhD thesis, Carnegie Mellon University Pittsburgh, PA, 1999.

[41] Joseph L Greathouse, Kent Knox, Jakub Poła, Kiran Varaganti, and Mayank Daga. clSPARSE: A vendor-optimized open-source sparse BLAS library. In *Proceedings of the 4th International Workshop on OpenCL*. ACM, April 2016.

[42] Khronos Group. *SYCL Specification: SYCL integrates OpenCL devices with modern C++, v1.2.1*, 1 2019.

[43] Sérgio Guerreiro. Decision-making in partially observable environments. In *2014 IEEE 16th Conference on Business Informatics*, volume 1, pages 159–166. IEEE, 2014.

[44] Milos Hauskrecht. Value-function approximations for partially observable markov decision processes. *Journal of artificial intelligence research*, 13:33–94, 2000.

[45] Milos Hauskrecht and Hamish Fraser. Planning treatment of ischemic heart disease with partially observable markov decision processes. *Artificial Intelligence in Medicine*, 18(3):221–244, 2000.

[46] Benjamin Hernandez, Hugo Pérez, Isaac Rudomin, Sergio Ruiz, Oriam de Gyves, and Leonel Toledo. Simulating and visualizing real-time crowds on GPU clusters. *Computación y Sistemas*, 18(4):651–664, 2014.

[47] Jesse Hoey, Pascal Poupart, Axel von Bertoldi, Tammy Craig, Craig Boutilier, and Alex Mihailidis. Automated handwashing assistance for persons with dementia using video and a partially observable markov decision process. *Computer Vision and Image Understanding*, 114(5):503–519, 2010.

[48] Kaijen Hsiao, Leslie Pack Kaelbling, and Tomas Lozano-Perez. Grasping pomdps. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 4685–4692. IEEE, 2007.

[49] Stefano Iannucci, Qian Chen, and Sherif Abdelwahed. High-performance intrusion response planning on many-core architectures. In *International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6. IEEE, 2016.

[50] Tsutomu Inamoto, Yoshinobu Higami, and Shin-ya Kobayashi. Optimal periods for probing convergence of infinite-stage dynamic programmings on GPUs. *International Journal of Networking and Computing*, 4(2):321–335, 2014.

[51] Intel. *Intel oneAPI Programming Guide (Beta)*, 10 2019.

[52] Intel. Threading building blocks (Intel(r) TBB), Consulted 1st of August, 2019.

[53] Ekaterini Ioannou, Odysseas Papapetrou, Dimitrios Skoutas, and Wolfgang Nejdl. Efficient semantic-aware detection of near duplicate resources. In *Extended Semantic Web Conference*, pages 136–150. Springer, 2010.

[54] Wojciech Jaśkowski. Mastering 2048 with delayed temporal coherence learning, multistage weight promotion, redundant encoding, and carousel shaping. *IEEE Transactions on Games*, 10(1):3–14, 2017.

[55] Arsæll Þór Jóhannsson et al. GPU-based Markov decision process solver. Master's thesis, School of Computer Science, Reykjavík University, 2009.

[56] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134, 1998.

[57] Sherry L Kausch, Jennifer M Lobo, Michael C Spaeder, Brynne Sullivan, and Jessica Keim-Malpass. Dynamic transitions of pediatric sepsis: a markov chain analysis. *Frontiers in pediatrics*, page 1060, 2021.

[58] Khronos OpenCL Working Group and others. The OpenCL specification, 2018.

[59] Mykel J Kochenderfer, Christopher Amato, Girish Chowdhary, Jonathan P How, Hayley J Davison Reynolds, Jason R Thornton, Pedro A Torres-Carrasquillo, N Kemal Ure, and John Vian. Optimized airborne collision avoidance. *Decision Making under Uncertainty*, page 251, 2015.

[60] MJ Kochendorfer. Decision making under uncertainty, 2015.

[61] Vikram Krishnamurthy. *Partially Observed Markov Decision Processes*. Cambridge University Press, 2016.

[62] Hanna Kurniawati, David Hsu, and Wee Sun Lee. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *Robotics: Science and systems*, volume 2008. Citeseer, 2008.

[63] Cody Kwok, Dieter Fox, and Marina Meila. Adaptive real-time particle filters for robot localization. In *2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422)*, volume 2, pages 2836–2841. IEEE, 2003.

[64] Marcello La Rocca. *Advanced Algorithms and Data Structures.* Simon and Schuster, 2021.

[65] Zhan Wei Lim, David Hsu, and Wee Sun Lee. Monte carlo value iteration with macro-actions. In *NIPS*, pages 1287–1295, 2011.

[66] Yunlong Liu and Jianyang Zheng. Combining offline models and online monte-carlo tree search for planning from scratch. *arXiv preprint arXiv:1904.03008*, 2019.

[67] William S Lovejoy. Computationally feasible bounds for partially observed markov decision processes. *Operations research*, 39(1):162–175, 1991.

[68] Rosalind B Marimont and Marvin B Shapiro. Nearest neighbour searches and the curse of dimensionality. *IMA Journal of Applied Mathematics*, 24(1):59–70, 1979.

[69] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM (JACM)*, 45(2):288–323, 1998.

[70] Ángel Martínez-Tenor, Juan Antonio Fernández-Madrigal, and Ana Maria Cruz-Martin. Lego© Mindstorms NXT and Q-Learning: a teaching approach for robotics in engineering. In *7th International Conference of Education, Research and Innovation (ICERI)*, pages 4836–4845, 2014.

[71] Gonçalo S Martins, Hend Al Tair, Luís Santos, and Jorge Dias. αpomdp: Pomdp-based user-adaptive decision-making for social robots. *Pattern Recognition Letters*, 118:94–103, 2019.

[72] Scott E Maxwell, Harold D Delaney, and Ken Kelley. *Designing experiments and analyzing data: A model comparison perspective.* Routledge, 3 edition, 2017.

[73] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 2:308–320, 1976.

[74] Nicolas Meuleau, Kee-Eung Kim, Leslie Pack Kaelbling, and Anthony R Cassandra. Solving pomdps by searching the space of finite policies. *arXiv preprint arXiv:1301.6720*, 2013.

[75] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[76] Jérôme Morio. Non-parametric adaptive importance sampling for the probability estimation of a launcher impact position. *Reliability engineering & system safety*, 96(1):178–183, 2011.

[77] Arslan Munir, Ann Gordon-Ross, and Sanjay Ranka. *Modeling and optimization of parallel and distributed embedded systems*. John Wiley & Sons, 2015.

[78] Kyle Wray Mykel Kochenderfer, Tim Wheeler. *Algorithms for Decision Making*. Massachusetts Institute of Technology, 2021.

[79] Angeles Navarro, Francisco Corbera, Andres Rodriguez, Antonio Vilches, and Rafael Asenjo. Heterogeneous parallel_for template for CPU–GPU chips. *International Journal of Parallel Programming*, pages 1–21, 2018.

[80] Angeles Navarro, Francisco Corbera, Andres Rodriguez, Antonio Vilches, and Rafael Asenjo. Heterogeneous parallel_for template for CPU–GPU chips. *International Journal of Parallel Programming*, 47(2):213–233, Apr 2019.

[81] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287, 1999.

[82] Konstantinos Nikas, Matthew Horsnell, and Jim Garside. An adaptive bloom filter cache partitioning scheme for multicore architectures. In *2008 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 25–32. IEEE, 2008.

[83] NVIDIA. The API reference guide for cuSPARSE, Consulted on October 24, 2019.

[84] Open Source Robotics Foundation. Gazebo, Consulted on October 24, 2019.

[85] OpenMP, ARB. The OpenMP API specification for parallel programming, Consulted on October 24, 2019.

[86] Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. Point-based value iteration: An anytime algorithm for pomdps. In *IJCAI*, volume 3, pages 1025–1032. Citeseer, 2003.

[87] Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. Anytime point-based approximations for large pomdps. *Journal of Artificial Intelligence Research*, 27:335–380, 2006.

[88] Joelle Pineau and Geoffrey J Gordon. Pomdp planning for robust robot control. In *Robotics Research*, pages 69–82. Springer, 2007.

[89] Pascal Poupart, Kee-Eung Kim, and Dongho Kim. Closing the gap: Improved bounds on optimal pomdp solutions. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 21, 2011.

[90] Warren B Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality.* John Wiley & Sons, second edition, 2011.

[91] Martin L Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming (Wiley Series in Probability and Statistics).* John Wiley & Sons, 2005.

[92] Coppelia Robotics. V-REP: virtual robot experimentation platform, Consulted 21st of June, 2022.

[93] Andrés Rodríguez, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Rubén Gran, Darío Suárez, and Jose Nunez-Yanez. Parallel multiprocessing and scheduling on the heterogeneous xeon+fpga platform. *The Journal of Supercomputing*, Jun 2019.

[94] Stéphane Ross, Brahim Chaib-Draa, et al. Aems: An anytime online search algorithm for approximate policy refinement in large pomdps. In *IJCAI*, pages 2592–2598, 2007.

[95] Nicholas Roy, Geoffrey Gordon, and Sebastian Thrun. Finding approximate pomdp solutions through belief compression. *Journal of artificial intelligence research*, 23:1–40, 2005.

[96] Sergio Ruiz and Benjamín Hernández. A parallel solver for Markov decision process in crowd simulations. In *Artificial Intelligence (MICAI), 2015 Fourteenth Mexican International Conference on*, pages 107–116. IEEE, 2015.

[97] Thomas B Schön. *Estimation of nonlinear dynamic systems: Theory and applications.* PhD thesis, Institutionen för systemteknik, 2006.

[98] Guy Shani, Joelle Pineau, and Robert Kaplow. A survey of point-based pomdp solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51, 2013.

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

[100] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, pages 2164–2172, 2010.

[101] Amit Kumar Singh, Charles Leech, Basireddy Karunakar Reddy, Bashir M Al-Hashimi, and Geoff V Merrett. Learning-based run-time power and energy management of multi/many-core systems: current and future trends. *Journal of Low Power Electronics*, 13(3):310–325, 2017.

[102] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal processing magazine*, 25(2):128, 2008.

[103] T Smith and R Simmons. Heuristic search value iteration for pomdps: Detailed theory and results. Technical report, Technical report, Robotics Institute, Carnegie Mellon University., 2004.

[104] Trey Smith and Reid Simmons. Point-based pomdp algorithms: Improved analysis and implementation. *arXiv preprint arXiv:1207.1412*, 2012.

[105] Adhiraj Somani, Nan Ye, David Hsu, and Wee Sun Lee. Despot: Online pomdp planning with regularization. In *Advances in neural information processing systems*, pages 1772–1780, 2013.

[106] Matthijs TJ Spaan and Nikos Vlassis. Perseus: Randomized point-based value iteration for pomdps. *Journal of artificial intelligence research*, 24:195–220, 2005.

[107] Zachary N Sunberg and Mykel J Kochenderfer. Online algorithms for pomdps with continuous state, action, and observation spaces. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*, 2018.

[108] S Joshua Swamidass and Pierre Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling*, 47(3):952–964, 2007.

[109] Lei Tai and Ming Liu. Mobile robots exploration through CNN-based reinforcement learning. *Robotics and biomimetics*, 3(1):3–24, December 2016.

[110] Atul Thakur, Petr Svec, and Satyandra K Gupta. GPU based generation of state transition models using simulations for unmanned surface vehicle trajectory planning. *Robotics and Autonomous Systems*, 60(12):1457–1471, 2012.

[111] Sebastian Thrun. Monte carlo pomdps. In *Advances in neural information processing systems*, pages 1064–1070, 2000.

[112] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics.* MIT press, 2005.

[113] Nguyen Mau Toan and Inoguchi Yasushi. Audio fingerprint hierarchy searching on massively parallel with multi-gpgpus using k-modes and lsh. In *2016 Eighth International Conference on Knowledge and Systems Engineering (KSE)*, pages 49–54. IEEE, 2016.

[114] Iván Fernández Vega. Development of a programming environment for a simulated TurtleBot-2 robot with a WindowsX manipulator arm through the connection of V-REP and MATLAB. B.Sc. Thesis, University of Málaga, 2016.

[115] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ parallel programming with threading building blocks.* Apress, 2019.

[116] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks.* Apress, 2019.

[117] DJ White. *Markov decision processes.* John Wiley & Sons, 1993.

[118] Marco Wiering and Martijn van Otterlo, editors. *Reinforcement Learning. State-of-the-Art.* Springer Verlag, 2012.

[119] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Performance Counter Monitor (PCM), Consulted 21st of January, 2020.

[120] Kyle Hollins Wray and Shlomo Zilberstein. Generalized controllers in pomdp decision-making. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7166–7172. IEEE, 2019.

[121] Kyle Hollins Wray, Shlomo Zilberstein, and Abdel-Illah Mouaddib. Multi-objective MDPs with conditional lexicographic reward preferences. *Proceedings of the Twenty-Ninth Conference on Artificial Intelligence (AAAI), Austin, Texas.*, pages 3418–3424, 2015.

[122] Zhimin Wu. *Parallelizing Model Checking Algorithms Using Multi-core and Many-core Architectures.* PhD thesis, Nanyang Technological University, Singapore, 2017.

[123] Ui Yamaguchi, Fuminori Saito, Koichi Ikeda, and Takashi Yamamoto. HSR, human support robot as research and development platform. In *Intl. Conf. on Advanced Mechatronics: toward evolutionary fusion of IT and mechatronics*, pages 39–40, 2015.

[124] Nevin L Zhang and Wenju Liu. Planning in stochastic domains: Problem characteristics and approximation. Technical report, Technical Report HKUST-CS96-31, Hong Kong University of Science and Technology, 1996.

[125] Cen Zhiwang, Xu Jungang, and Sun Jian. A multi-layer bloom filter for duplicated url detection. In *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, volume 1, pages V1–586. IEEE, 2010.

[126] He Zhou, Sunil P Khatri, Jiang Hu, Frank Liu, and Cliff Sze. Fast and highly scalable bayesian MDP on a GPU platform. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 158–167. ACM, 2017.

[127] Xin-Zhong Zhu and Jian-Min Zhao. Spoken dialogue management as planning and acting under uncertainty. In *Proceedings. International Conference on Machine Learning and Cybernetics*, volume 2, pages 669–673. IEEE, 2002.