UNIVERSITY OF CALIFORNIA,
IRVINE

Computation-communication co-optimization in the era of networked embedded devices

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Seyyed Ahmad Razavi

Dissertation Committee:
Professor Eli Bozorgzadeh, Chair
Professor Ian Harris
Professor Solmaz Kia

2022

DEDICATION

To my family, and to my father, who never saw this day.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, my colleagues, and my friends for supporting me in this journey.

I thank Institute of Electrical and Electronics Engineers (IEEE), Association for Computing Machinery (ACM) for giving me permissions to include my previously published papers in this dissertation.

Portions of Chapter 2 were previously published as "Resource-Aware Decentralization of a UKF-Based Cooperative Localization for Networked Mobile Robots, Euromicro Conference on Digital System Design, 2018, S. A. Razavi, E. Bozorgzadeh, K. Kim and S. Kia". Permissions to reuse the text were granted by IEEE. The co-author listed in this publication directed and supervised research which forms the basis for the thesis/dissertation.

Portions of Chapter 3 were previously published as "Communication-Computation co-Design of Decentralized Task Chain in CPS Applications, Design, Automation Test in Europe Conference Exhibition (DATE), 2019, S. A. Razavi, E. Bozorgzadeh and S. S. Kia". Permissions to reuse the text were granted by IEEE. The co-author listed in this publication directed and supervised research which forms the basis for the thesis/dissertation.

Portions of Chapter 4 were previously published as "On Exploiting Patterns For Robust FPGA-based Multi-accelerator Edge Computing Systems, Design, Automation Test in Europe Conference Exhibition (DATE), 2022, S. A. Razavi, H. -Y. Ting, T. Giyahchi and E. Bozorgzadeh". Permissions to reuse the text were granted by IEEE and co-authors.

VITA

Seyyed Ahmad Razavi

EDUCATION

Doctor of Philosophy in Computer Science **2022**
University of California, Irvine *Irvine, California*

Master in Computer Engineering **2010**
Amirkabir university of technology *Tehran, Iran*

Bachelor in Computer Engineering **2007**
Isfahan university of technology *Isfahan, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant **2014–2022**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2015–2022**
University of California, Irvine *Irvine, California*

Publications

- S. A. Razavi, H.-Y. Ting, T. Giyahchi, and E. Bozorgzadeh. On exploiting patterns for robust fpga-based multi-accelerator edge computing systems. In *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022
- M. Sadeghi, S. A. Razavi, and M. S. Zamani. Reducing reconfiguration time in fpgas. In *2019 27th Iranian Conference on Electrical Engineering (ICEE)*, pages 1844–1848. IEEE, 2019
- S. A. Razavi, E. Bozorgzadeh, and S. S. Kia. Communication-computation co-design of decentralized task chain in cps applications. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1082–1087. IEEE, 2019
- S. A. Razavi, E. Bozorgzadeh, K. Kim, and S. S. Kia. Resource-aware decentralization of a ukf-based cooperative localization for networked mobile robots. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 296–303. IEEE, 2018
- S. A. Razavi and M. Saheb Zamani. Improving bitstream compression by modifying fpga architecture. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 167–170, 2013
- S. H. Moallempour, S. A. Razavi, and M. S. Zamani. Tsv reduction in homogeneous 3d fpgas by logic resource and input pad replication. In *2011 IEEE International 3D Systems Integration Conference (3DIC), 2011 IEEE International*, pages 1–5. IEEE, 2011
- H. Ebrahimi, M. S. Zamani, and S. A. Razavi. A switch box architecture to mitigate bridging and short faults in sram-based fpgas. In *2010 IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 218–224. IEEE, 2010
- D. Aghamirzaie, S. A. Razavi, M. S. Zamani, and M. Nabiyouni. Reduction of process variation effect on fpgas using multiple configurations. In *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, pages 85–90. IEEE, 2010
- S. A. Razavi, M. S. Zamani, and K. Bazargan. A tileable switch module architecture for homogeneous 3d fpgas. In *2009 IEEE International Conference on 3D System Integration*, pages 1–4. IEEE, 2009
- A. Neekabadi, S. Samavi, S. Razavi, N. Karimi, and S. Shirani. Lossless microarray image compression using region based predictors. In *2007 IEEE International Conference on Image Processing*, volume 2, pages II–349. IEEE, 2007
- A. Neekabadi, S. Samavi, N. Karimi, E. Nasr-Esfahani, S. Razavi, and S. Shirani. Lossless compression of mammographic images by chronological sifting of prediction errors. In *2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 58–61. IEEE, 2007

ABSTRACT OF THE DISSERTATION

Computation-communication co-optimization in the era of networked embedded devices

By

Seyyed Ahmad Razavi

Doctor of Philosophy in Computer Science

University of California, Irvine, 2022

Professor Eli Bozorgzadeh, Chair

Nowadays, networked embedded systems are widely being used. These systems are consisted of multiple embedded devices connected through wired or wireless network to accomplish a mission, such as networked robots for rescue missions and surveillance systems. Networked embedded systems are driven by the tight coordination between computational components, sensors and the interaction with each other, more specifically, in cooperative distributed tasks that each device has to share its data with others to be processed. These devices have a limited computation and computation capacity, therefore, solely considering just the computation or just the communication in designing applications, may result in enormous communication or computation overhead. Therefore, computation-communication aware design is unavoidable in order to achieve a reliable and yet fast execution.

In the first part of the dissertation, we explore a methodology to decentralize an application among a team of cooperative agents (networked embedded systems). As a target application, we focused on Cooperative Localization, where there is a tightly coupled data dependency between agents. Cooperative localization (CL) is a popular method for localizing a team of communicating robots in GPS-denied environments. In this iterative application, the data is generated locally on each agent. The data is then shared among all agents because the application output depends on the data generated by the whole team. After computing the

location, the output has to be sent back to the agent. Fast and accurate localization is critical because a delayed estimation will mislead navigation and other applications and may lead to a mission failure. Therefore, we first distribute the UKF based CL among robots, then we provide a replication technique to reduce the communication overhead, which improves the run-time of the algorithm. However, computation replication increases the computation overhead on the agents. Therefore, we provide a method to choose between the replication amount that minimizes the overall CPU overhead considering a user defined application latency.

In some applications, because of limited computation capacity of embedded devices, the data has to be sent to a node with a higher computation capacity, such as a local server, edge, or cloud for processing acceleration. In addition, in some systems, agents are not willing to share their information with each other, other than a trusted node (e.g. edge). Edge computing has been recently the focus of many research projects. In this part of the dissertation, we provide a methodology for agents that are coordinated with the edge to reduce the task wait time on the edge. In an edge with a time slotted scheme, task arrival time impacts the system's performance. Since the on-device computation/network delay varies, the tasks might arrive after the allocated time on the edge, so they need to wait on the edge until the next time the accelerator becomes available for processing. In a cooperative system, the edge can guide the end device to shift its sampling time. This will ensure that the tasks arrive at the right moment to reduce the task wait time on the edge. Therefore, we proposed a method to find the end device sampling time adjustment during runtime to decrease the overall task wait time on the edge based on the arrival time distribution. However, the tasks arrive in a streaming mode and the entire arrival time distribution is unknown. In addition, the distribution changes over time. Therefore, we also developed a method to predict the task arrival time, and enforce the adjustment delay during runtime using a feedback loop. The experimental result shows that by sending a limited number of commands to the end devices (Raspberry Pi boards), the wait time decreased by up to 74%.

In conclusion, during this dissertation, we illustrated how communication-computation co-optimization can improve the performance of networked embedded systems. We provided solutions for application decentralization while considering the communication-computation tradeoff. Furthermore, we show that the coordination between networked embedded systems and the edge can improve the responsiveness despite the network noise by just utilizing task arrival time and a feedback loop.

Chapter 1

Introduction

In recent years the number of connected embedded systems has increased significantly. Statista [4] has estimated that the number of IoT devices will increase to 75.4B devices by 2025. These systems might work individually or as part of a team to accomplish a goal, such as monitoring the health signals of a patient [80, 118, 117], providing safety for houses [11], controlling the city traffic [113], etc. Networked embedded devices have sensing, computing, and communicating capabilities. They might even have an actuator such as an arm or a wheel. They varies a lot in terms of technologies, and the designers, use them based on their needs. There are a wide range of sensors that measure a parameter, such as temperature, pressure, distance, image, etc. They might have a basic microcontroller [1], multi core microprocessor [7], or an on-board FPGA or GPU [5]. In terms of communication technology, they might take advantage of wired or wireless communication such as Zigbee, Bluetooth, WiFi, or 4G/5G/6G cellular systems. Due to various parameters such as power consumption and cost per unit, the system designers choose the devices with just enough computation and communication capacity to fulfill the system requirements. These limitations impose many challenges in designing software [79], such as lowering the computation load, and minimizing the communication overhead.

There is a tight connection between sensing, computation, and communication in networked embedded systems. The processing starts after receiving data from a sensor, and the raw data or processed data might be shared with other connected embedded nodes, or an edge, for further processing. The size of data that the embedded device transfer to the other nodes affects the communication delay and power consumption significantly. Furthermore, the computation time has correlation with the size of data. For example, the image resolution taken by a camera drastically has a strong correlation with the jpeg encoding latency. Based on the computation allocation, applications can be categorized into centralized, decentralized and distributed.

1.1 Centralized, decentralized and distributed

Cooperative agents share their locally generated data with each other, and jointly process them [44], and the result might be sent to any of the team members. In [23], a social learning approach has been presented where a team of robots process their sensory data and actions using distributed reinforcement algorithm. The data processing can be done on a server, or distributed among the team. Running the application on a single node, a.k.a server, makes the system vulnerable as the server becomes the single point of failure. Figure 1.1 shows a centralized method where agents send their data to a server for processing, and server sends back the result to agents. To avoid a single point of failure and to increase scalability, application decentralization has been studied in the literature, for example in robotics community [27]. Decentralization can be considered as a subset of distribution. In decentralization, the application is distributed in a way that each node can make decision, while in a centralized method, all the decision is made in a single node. On the other hand, in application distribution, data processing is distributed among the agents, and it may or may not involve decision making. Decentralization requires a deep knowledge of the

application, and an expert has to set the aspects of the system. In this dissertation, we used UKF cooperative localization as a sample application, and we considered it decentralized if each agent can compute its own location. An agents can receive the required data, including the partial computation, from other agents. Application distribution enables near data processing, since the computation load of the application distribute among all the agents, which usually have a limited computational resources. Near data processing usually decreases the application latency, which is crucial for time critical applications. Figure 1.2 shows a distributed system where data processing happens in the agents, and the result has to be transferred to the corresponding agents. Since the application is not decentralized, if an agent fail, it will lead to system failure. The decentralized application is shown in Figure 1.3 where each agent computes its own partial result and an agent failure will not interrupt the system. Note that agents are still depend on the data that they receive form other team members. Computation distribution might not be feasible for all applications due to their heavy computational load. Therefore offloading the computation to a server with a stronger processing unit might be necessary.

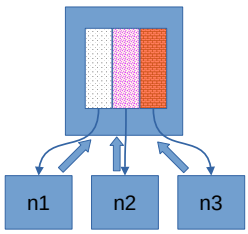

Figure 1.1: centralized

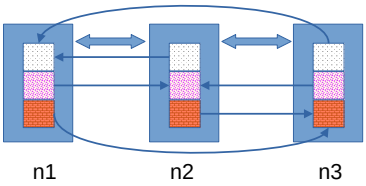

Figure 1.2: distributed but not decentralized

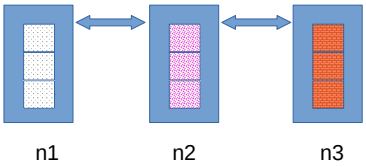

Figure 1.3: distributed and decentralized

1.1.1 Distributed/decentralized method

In harsh environments, such as natural disasters, the devices might not have access to a server, therefore, the data processing has to be done by the team. Running the entire application on a device might not be possible due to the limited computation power. Therefore, application distribution is necessary. In addition, relying on one device means having a single point of failure. Therefore, decentralization is essential. For example, in [66], the Simultaneous Localization and Mapping (SLAM) is distributed among the entire team, where each agent computes a part of SLAM and share the information with others for further processing. In [112], a distributed algorithm is proposed for autonomous cars where each vehicle receives data from other vehicles to improve their Kalman based target tracking accuracy. The tight coupling of sensing, computation, and communication among the agents makes the application distribution challenging.

Although the computation load of each node might decrease due to the computation distribution, the communication overhead increases. The data transfer overhead might lead to a high and unaffordable latency, specifically, for devices with wireless connection. On the other hand, the sensory data is generated and consumed locally, which imposes restrictions on the allocation of application components on the devices. Therefore, in distributing an application on networked embedded devices, the computation allocation has to be done considering the locality of data and the data transfer due to the computation distribution.

Due to the wide diversity of devices, there is no unique solution for distributing an application,

and the application has to be optimized considering the device computational and Communicational resources. For example, if the communication link between the devices is through Zigbee, transferring data will take a long time compared to WiFi, and the computation allocation has to be in a way that the size of data transfer get minimized. Without a computation-communication aware method, the decentralized algorithm may suffer from long End-to-End delay and high computation and communication overhead, which makes it impractical. Therefore, an automated method is required to optimize the application. To this end, we proposed a algorithmic way to distribute an application on the networked embedded systems.

1.1.2 Centralized method

In some systems, the embedded devices has access to a node with a stronger processing unit. This node can be a local server, an edge, or a cloud. Cloud provides virtually an unlimited processing power for users, and due to their enormous storage, they can be used for big data applications [88]. However, they suffer from high network latency, which might not be suitable for real time applications. In addition, due to privacy concerns, users might not want to share their data with the cloud. Therefore, edge computing is getting more popular. Offloading computation to edge is a promising method for networked embedded to process their data with low latency. The computation offloading strategy can be optimized for various parameters such as lowering the latency, or reducing the energy consumption [71]. Edges can exploit strong processing units, such as FPGAs and GPUs. [28, 124, 22] studied the benefits of using FPGA in the edge such as lower power consumption and higher throughput for DNN applications. [33, 57] evaluated the performance of GPU on the edge, which are easier to develop application for.

There are extensive number of research papers for leveraging the power of edge for IoT systems, which some of them surveyed in [26, 76, 115]. Edge can be used to accelerate

robotic applications such as image classification using deep learning and SLAM. In recent years, due to the advent of low latency network technologies, such as 5G and 6G, controlling robots using edge and cloud is getting more viable [15]. The robot can use the storage and processing power of the cloud with low latency, for various applications such as image classification using DNN. In [133], a real-time SLAM has been proposed, where the agent computes a small portion of the map, and the server, processes the whole map and sends information to the agent. [109, 30, 50] used edge for computing SLAM. SLAM can take more than 80% of the CPU time on a commodity robot [50]. [20] partitioned a visual SLAM computation between the robot and the edge, and moved all the computation except the tracking to the edge. Authors in [51] proposed ColaSLAM that uses multiple edges to process the data from multiple robots for collaborative SLAM, which is 40% faster than running it on the cloud. [16] used edge computing to take advantage of the local context information, such as the wireless network delay, to smoothly drive a robot. [36] studied offloading the deep learning based image classification from aerial robots to the edge.

In application such as remote surgery, generating cloud point from the vehicle camera, and controlling robots, the freshness of the data has a direct impact on the performance of the application. The freshness of data can be measured by Age of Information (AoI) metric, which is the average delay between the sensor sampling time on the end device and the beginning of the processing of the data on the edge [128]. This metric can be linear, which increases linearly by time, or nonlinear, to penalty the late tasks. [61] has studied the effect of sampling rate on the AoI. If the sampling rate is high, the tasks has to stay in queues on both end device and edge, which will leads to high AoI. On the other hand, if the sampling rate is low, the data will be outdated. In [54, 53, 55] authors investigate the optimal policy for minimizing the average age of information in a multiple sensing setup. They analytically derive the average AoI in specific setups and show that Last Come First Serve (LCFS) queuing protocol results in a smaller average AoI compared to First Come First Serve (FCFS). [19] has proposed a message selective encoding method for optimizing

the AoI. [129] proposed an online compression method to jointly compress the data from multiple sources, since for compressing the data from on source, multiple measurement have to collected, which increase the AoI.

1.2 Case study: Robot Localization

Localization is a crucial application for mobile agents such as Unmanned Aerial Vehicles (UAV), self-driving cars, and mobile ground robots. Localization should be fast and accurate because a delayed or inaccurate estimation mislead the agent and might even cause a mission failure. The localization has to run frequently, and its end-to-end delay should be small enough to be able to capture the motion of the robot and provide the higher-level applications with accurate and almost real-time data. A robot can find its location using Global Navigation Satellite System (GNSS). BeiDou, Galileo, GLONASS and the well-known GPS (Global Position System) are subsets of GNSS [2]. Users receive signals that is broadcasted by GPS satellites, and calculate their location based on it. The accuracy of the localization depends on the receiver hardware, the environment, and the satellite location, and it can be around 5 meters for smart phones, to a few centimeters for high end devices [3]. However, in some environments, such as inside the buildings and tunnels, there is no GPS signal. In addition, equipping devices with GPS sensor increases the unit cost. Therefore, other GPS-free methods, such as Dead-Reckoning [81] and Simultaneous Localization And Mapping (SLAM) [120], are being used for such environments.

Dead Reckoning is based on tracking. where the new location can be calculated by knowing the previous location and the driven distance. This distance can be estimated by sensors such as Inertial Measurement Unit (IMU) and gyroscope, measured by wheel encoder [68], or simply by monitoring the control signals. The drawback of this method is that sensors are not ideal and have error. This error will lead to accumulated localization error, and

after a while, the robot estimated location will drift from the actual location. To reduce the effect of noise, estimation methods such as Kalman filter has been applied to combine sensor measurements with the predicted state of the system. Kalman filter is a popular method to denoise the data during runtime for applications such as target tracking, and localization. Basic kalman filter is designed for linear systems, and the noise is modeled by Gaussian distribution. Kalman filter, at each time step k, use the control signals (U_k), sensor measurements, and the previous state of the system (X_{k-1}) to estimate the new state of the system (X_k), here the location of agent and its uncertainty (Figure 1.4). The estimated location (X_k) and its uncertainty (P_k) will be used as the input for higher-level applications such as navigation. There are many research papers on the Kalman filter, some of them surveyed in [114, 9, 91]. Extended Kalman filter (EKF) and Unscented Kalman filter (UKF) are two extensions of Kalman filter for non linear systems. EKF linearizes the state transition and observation models, which leads to lower accuracy compared to UKF. In addition, EKF is inconsistent for highly nonlinear systems or when the filter has high initialization errors.. On the other hand, UKF is more accurate and stable [65, 40, 12].

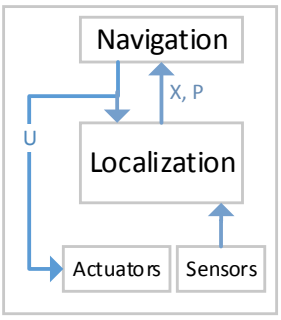

Figure 1.4: Single robot localization

A recent paradigm in localization for a multi-robot system is Cooperative localization (CL) (Figure 1.5). Each robot share its locally generated data with other robots, and more accurate location can be estimated using those information [122]. The estimated location has to be sent to the team members to be used for other applications running on the robot. Distance based CL uses the measured relative distance between the agents, and increases

the accuracy of the localization of the entire team. The distance can be measured using low cost sensors such as sonar, camera, and radar. [41] equipped Unmanned Aerial Vehicles with Inertial, and Ultra-Wide Band sensor.

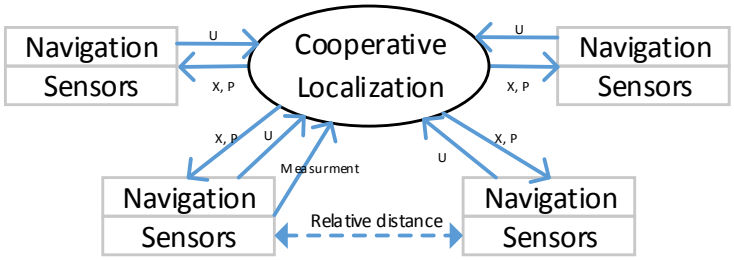

Figure 1.5: Cooperative Localization for a team of robots

The measured distance is timestamped, and can be fused with the predicted state of the system. In CL, the combined location of all the agents is considered as the state of the system [102]. Therefore, a measurement between two agents not only affect the localization of the involved robots, but the entire team. Various methods have been used for CL such as Extended Kalman filters (EKF) [106, 41, 67], Unscented Kalman filter (UKF) [31], particle filters [37, 48, 94, 73], maximum likelihood [49], and maximum a posteriori (MAP) [86]. [135] has proposed a method to reduce the communication overhead of EKF based CL by running a decentralized filter. CL method has been presented in [69, 123] for under water vehicles, where agents can communicate with each other using acoustic waves. A CL method based on the visible light positioning and odometer is presented in [125]. This method requires image processing to detect the distance to the light. [90] presented a Deep Reinforcement Learning model for measurement sharing schedule among vehicles to minimize the communication overhead of Kalman filter based CL.

To avoid a single point of failure and to increase scalability, CL decentralization has been studied by the robotics community. [32] proposed a distributed UKF based CL where the computation is distributed among the agents, but they rely on a server to store and transfer the data. In this method, each agent computes the columns of Cholesky decomposition,

which is used to get the moments of the covariance matrix. In CL, because robots share their local estimations to jointly calculate their location using relative distance measurement, the computations of robots are highly correlated. This correlation creates a challenge in the decentralization of CL algorithms because it induces significant processing and communication requirements [107, 89, 60, 86, 64, 75]. Various decentralization schemes using EKF formulations have been proposed in the literature [107, 64, 75, 63]. On the other hand, UKF is proven to work more consistently than the EKF for systems with nonlinear state and measurement models [58]. A simulation study over ground robots in [31] shows significant estimation performance improvement for a UKF based cooperative localization in comparison to an EKF based cooperative localization. Therefore, in this paper, we focused on UKF based CL. UKF is composed of tightly-coupled computationally-intensive tasks that may lead to poor performance during a deployment. The wireless communication delay overhead has been a major barrier in adopting UKF despite their more accurate solution compared to conventional EKF approach [85].

The robot's control signals and sensor data are generated locally, and also the estimated location will be used locally. Therefore, the data processing has to be on or near the robots to avoid a large communication delay. On the other hand, applications such as localization have to be executed frequently, therefore, they impose a consistent overhead on the limited robot's resources entire the mission.

Cooperative Localization is an essential application in networked robots. In addition, it requires data from all the robots, and the result will be consumed locally. On the other hand, UKF, despite having tight data dependency, is more accurate than the EKF. Therefore, we chose UKF based Cooperative Localization as the target application for our research, and provided a methodology to distribute it among the robots considering the limited computation and communication capacity of robots as networked embedded systems.

1.3 Overview and contributions of this dissertation

In this dissertation, we focus on computation-communication co-optimization for networked embedded systems. At first, we propose a serverless method [98] for decentralizing an iterative cooperative application on a team of agents to minimize the communication among them. We take the UKF based CL as the target application and provide an automated method to further reduce the data transfer between agents by computation replication technique, which reduces the application end-to-end delay. However, computation replication increases the CPU utilization of the agents. Hence, there is a trade-off between computation and communication. For networked embedded systems with limited computational and communicational resources, the overhead of computation replication is not negligible. On the other hand, by increasing the number of replicated computations, the communication overhead reduces. Therefore, finding the right amount of replication is not trivial. Thus, We proposed a method [97] to automatically find the replication that minimize the overall CPU utilization, while meeting the user defined end-to-end delay.

Computation offloading to Edge is a promising way to accomplish computational heavy applications, such as image classification using DNN. For a time slotted edge, if the task arrival time does not match with the accelerator timing, the task has to wait on the edge for the next slot. In this dissertation, we show that the edge can reduce the wait time, and hence the AoI, significantly by guiding the end device to adjust its sampling time. Due to the network delay variation, finding the adjustment delay is challenging. First, using statistical analysis, the optimal task arrival time is calculated on the edge. Then, using a feedback loop, the edge adjusts the end device timing. We proposed a sliding window averaging and a Kalman filter based staggering method. Experimental results shows that the Kalman filter based methods requires less communication with the end node compared to sliding window averaging. Our method does not need complicated estimation methods, and impose minimum engineering cost and computation overhead to the end-device. In addition, we

extended our method to support the cooperative applications that need synchronized sensing among the team, such as UKF based CL. The proposed method show how collaboration between edge and end devices can improve the responsiveness of the system.

Chapter 2

Communication aware decentralization

2.1 Introduction

Localization is a crucial application for mobile agents such as Unmanned Aerial Vehicles (UAV), self-driving cars, and mobile robots. Localization should be fast and accurate because a delayed or inaccurate estimation mislead the agent and might even cause a mission failure. The localization has to run frequently, and its End-to-End delay should be small enough to be able to capture the motion of the robot and provide the higher-level applications with accurate and almost real-time data. A popular method for localization is using Global Positioning System (GPS). However, GPS is not available in environments like inside of the buildings and tunnels. Therefore, other GPS-free methods, such as dead-reckoning [81] and Simultaneous Localization And Mapping (SLAM) [120], are being used for such environments. In a GPS-denied environment, due to accumulated sensor error, the estimated location drifts from the actual location after a while. Therefore, a filter, usually

Kalman Filter, is applied to reduce the effects of noise and also to integrate the estimation gained from various sensors in order to have a more accurate estimation. Filters, at each time step k, use the control signals (U_k), sensor measurements, and the previous state of the system (S_{k-1}) to estimate the new state of the system (S_k), here the location of agent and its uncertainty (Figure 1.4). The estimated location (X_k) and its uncertainty (P_k) will be used as the input for higher-level applications such as navigation. EKF and UKF are two well-known filters for estimating the state of non-linear systems. EKF linearizes the state transition and observation models, which leads to lower accuracy compared to UKF. In addition, EKF is inconsistent for highly nonlinear systems or when the filter has high initialization errors.

A recent paradigm in localization for a multi-robot system is Cooperative localization (CL), where agents share data with the team to increase the accuracy of localization (Figure 1.5). CL uses relative distance measurements among the robots, using low cost local sensors such as sonar, camera, and radar as feedback to increase the accuracy of the estimated location. Various filters have been used for CL algorithms such as Extended Kalman filters (EKF) [106], Unscented Kalman filter (UKF) [31], particle filters [37, 48, 94, 73], maximum likelihood [49], and maximum a posteriori (MAP) [86].

To avoid a single point of failure and to increase scalability, CL decentralization has been studied by the robotics community. In addition, decentralization enables near data processing, and hence, the data fusion can be partly processed in each node with respect to the local sensor data. However, in CL, because robots share their local estimations to jointly calculate their location using relative distance measurement, the computations of robots are highly correlated. This correlation creates a great challenge in the decentralization of CL algorithms because it induces significant processing and communication requirements [107, 89, 60, 86, 64, 75].

Various decentralization schemes using EKF formulations have been proposed in the literature

[107, 64, 75, 63]. On the other hand, UKF is proven to work more consistently than the EKF for systems with nonlinear state and measurement models [58]. A simulation study over ground robots in [31] shows significant estimation performance improvement for a UKF based cooperative localization in comparison to an EKF based cooperative localization. Therefore, in this paper, we focused on UKF based CL. UKF is composed of tightly-coupled computationally-intensive tasks that may lead to poor performance during a deployment. The wireless communication delay overhead has been a major barrier in adopting UKF despite their more accurate solution compared to conventional EKF approach [85].

Since the robot's control signals and sensor data are generated locally, and also the estimated location will be used locally, the data processing has to be on or near the robots to avoid a large communication delay. On the other hand, applications such as localization have to be executed frequently, therefore, they impose a consistent overhead on the limited robot's resources entire the mission.

Based on the locality of computation and data, various schemes have been proposed for CL: centralized, partially decentralized, and fully decentralized. In the centralized method (Figure 2.1.a), CL runs on a central machine, which computes the state of the system (X and P), given the measurement and control signals that have been sent by each robot. The central machine can be a cloud, an edge, a local server, or any of the robots. Because of the long delay of sending data between cloud and robots, the cloud is not suitable for running time-critical applications like CL. In the case of a local server, the delay of communication is less than the delay of sending data to a cloud, however, a local server might not be available in some missions such as search and rescue. The central machine can be one of the robots, however, the connection between the central robot and other robots might be lost. Moreover, this central robot will be the single point of failure of the system. Therefore, the CL has to be decentralized among the robots to be reliable and fast enough.

The tight coupling of sensing, computation, and communication among the agents makes

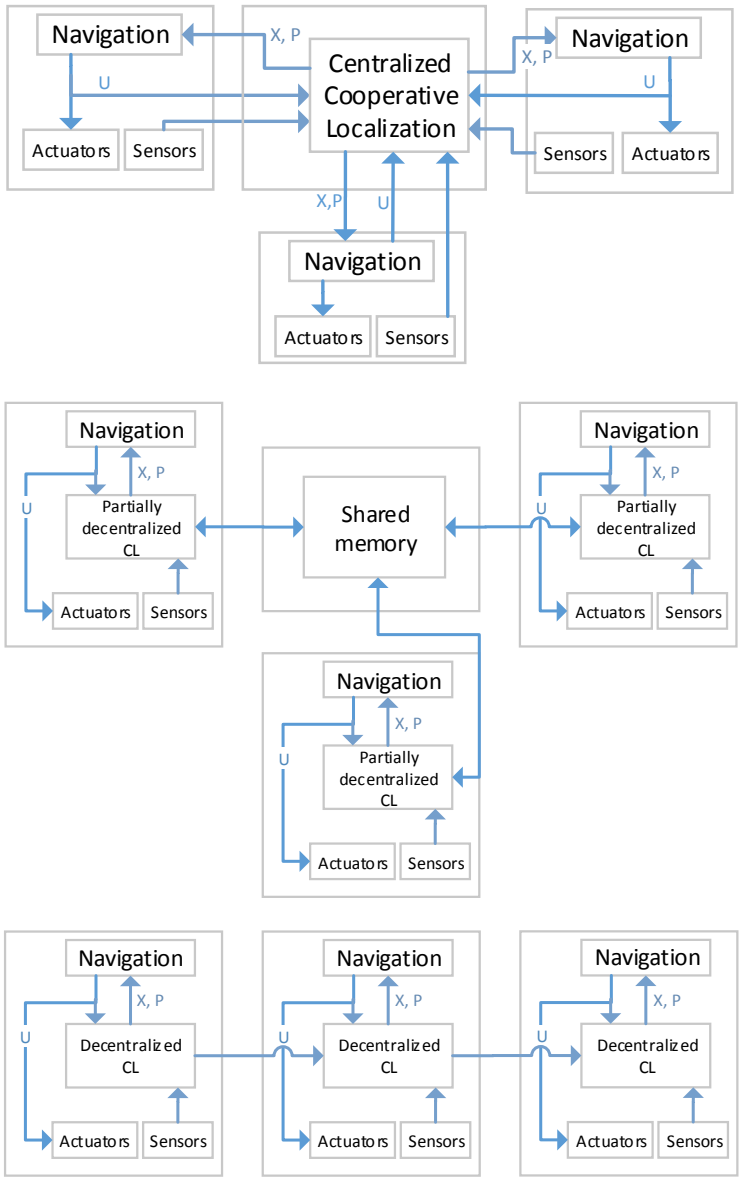

Figure 2.1: Various schemes for CL

the decentralization of UKF-based CL challenging. Without a computation-communication aware method, the decentralized algorithm may suffer from long End-to-End delay and high computation and communication overhead, which makes it impractical. For example, in [31], a partially decentralized UKF-based CL is proposed in which the UKF equations are decoupled in a way that UKF computations is distributed among all the robots, and the system's state and intermediate results are stored in a server (Figure 2.1.b). The robots read data from the server, and after processing, write the result back to the server. In this method, the server is a single point of failure, and the size of transferred data is large. This method is inefficient because decentralization results in a large size of data transfer between agents and server which significantly increases the delay.

In [98], we proposed a fully decentralized method where the computation and data are distributed among robots. Each robot computes its own location information using the information received from other agents (Figure 2.1.c).

Unlike other methods, if the connection between robots lost, the robot can still localize itself because it has the required information, including the location information from last time step, locally. In addition, we developed a framework to overcome the complexity of decentralization of UKF-based CL without loss of accuracy and computationally-identical to the centralized method. At first, UKF is fully decentralized using K-way partitioning to minimize the size of data transfer between agents considering the locality of data. In this method, the shared memory server in [31] is removed and both computation and data are shared among the agents. Therefore, the size of data transfer and also delay are reduced compared to partially decentralized method [31]. In addition, we proposed a replication method in [98] to further reduce the size of data, where some computations repeated in the destination agents, therefore, fewer data need to be transferred. Using the replication technique, only control signals need to be transferred between agents which can fit in one network packet. Due to the reduced data transfer, the End-to-End delay is less than the

fully decentralized method without replication.

In this paper, the UKF-based CL's decoupling and decentralization are studied in detail, and a method to replicate the computation for stateful applications has been proposed. By replicating computations, as in [98], the overall communication between agents can be reduced significantly while the overall computation overhead will increase. The rest of the paper is organized as follows. In section 2, UKF based CL and system configuration will be explained in detail. Section 3 describes the proposed framework, which includes decentralizing the application, communication link refinement, and computation replication. Finally, experimental results including End-to-End delay and computation overhead of the proposed method on multiple single-board computers have illustrated in Section 4.

2.2 UKF-based Cooperative Localization

EKF and UKF are variants of the Kalman filter for estimating the state of a non-linear system denoted by $\mathbf{x}^+$. These filters are composed of *prediction* and *update* steps. In the prediction step, which runs periodically, the motion model of the system is used to predict the state estimate of the system based on the control signals and the prior state estimate and covariance matrix. The Update step runs whenever the system receives a measurement.

In CL, a measurement between any two robots can improve the accuracy of the estimated location of all robots because of their correlation through covariance matrix. In Kalman based CL, as shown in Figure 2.2, robots at each time step k share their control signals ($\mathbf{u}(k)$), state vectors ($\mathbf{x}^+(k)$), their uncertainty ($\mathbf{P}^+(k)$), and sensors measurements to calculate the location ($\mathbf{x}^+(k+1)$) and uncertainty of estimated location ($\mathbf{P}^+(k+1)$) of the next time step $(k+1)$. In the context of CL, the state vector ($\mathbf{x}^+(k+1)$) includes the global pose (position and orientation of the robots) in addition to possibly other states required for modeling the

dynamics of the robots (e.g. steering angle). In a system of N robots, the state vector of each robot i is represented by n_i state variables. These states adds up to n_x states of the system ($\mathbf{x}^+(k)$).

In a UKF-based CL, the prediction step for the collective system with n_x states starts with computing the square root matrix, as a triangular matrix, of matrix $\mathbf{P}^+(k)$ using the Cholesky Decomposition method. After that, a set of $2n + 1$ sample points, called Sigma Points (χ), is generated by eq. (3.3b). In the equations, (c) denotes the c^{th} column of the matrix. The state transition model (f) will use Sigma Points and $\mathbf{u}(k)$ to generate Transformed Sigma Points ((3.3c)). The predicted UKF state ($\mathbf{x}^-$) is the weighted arithmetic mean of transformed sigma points (χ^-) ((3.3d)). Finally, the predicted covariance matrix ($\mathbf{P}^-(k+1)$) will be obtained by equation (2.1f) using prediction error $\mathbf{e}_x$ in (3.3e). In the equations, $l \in \{0, \cdots, 2n\}$, $c \in \{1, \cdots, n_x\}$, and w are based on a system-defined constant κ ($w_{(l)} = \frac{1}{2(n_x+\kappa)}$ and $w_{(0)} = \frac{\kappa}{(n_x+\kappa)}$).

$$\mathbf{P}^+(k) = \mathbf{L}(k)\mathbf{L}(k)^\top, \tag{2.1a}$$

$$\chi_{(0)} = \mathbf{x}^+(k), \chi_{(c,c+n_x)} = \mathbf{x}^+(k) \pm \sqrt{(n_x+\kappa)}[\mathbf{L}(k)]_c, \tag{2.1b}$$

$$\chi^-_{(l)} = \mathbf{f}(\chi_{(l)}, \mathbf{u}(k)), \quad l \in \{0, \cdots, 2n_x\}, \tag{2.1c}$$

$$\mathbf{x}^-(k+1) = \sum_{l=0}^{2n_x} w_{(l)}\, \chi^-_{(l)}, \tag{2.1d}$$

$$\mathbf{e}_{x,(l)} = \chi^-_{(l)} - \mathbf{x}^-(k+1), \tag{2.1e}$$

$$\mathbf{P}^-(k+1) = \sum_{l=0}^{2n_x} w_{(l)} \mathbf{e}_{x,(l)} \mathbf{e}^\top_{x,(l)} + \mathbf{B}(k)\mathbf{Q}(k)\mathbf{B}(k)^\top. \tag{2.1f}$$

Whenever there is a measurement ($\mathbf{z}_{ab}$) in the system, the Update step runs. UKF update step increases the accuracy of the predicted state, computed by prediction step, using the measurement and Kalman gain. In update step, innovation covariance $\mathbf{S}_{ab}$, cross-covariance

$\mathbf{P}_{\text{XZ}}$, $\mathbf{e}_{z,(l)}$, and innovation $\mathbf{r}^a$ will be computed by (2.2) using the Sigma Points. In relative measurement models, we only need the sigma points corresponding to robot a and b. Then, the predicted relative measurement, the measurement residual and the innovation covariance are, respectively,

$$\zeta_{ab,(l)} = \mathbf{h}_{ab}(\chi^{a^-}_{(l)}, \chi^{b^-}_{(l)}) \tag{2.2a}$$

$$\hat{\mathbf{z}}_{ab} = \sum\nolimits_{l=0}^{2n_x} w_{(l)} \zeta_{ab,(l)}, \tag{2.2b}$$

$$\mathbf{r}^a = \mathbf{z}_{ab} - \hat{\mathbf{z}}_{ab} \tag{2.2c}$$

$$\mathbf{e}_{z,(l)} = \zeta_{ab,(l)} - \hat{\mathbf{z}}_{ab} \tag{2.2d}$$

$$\mathbf{S}_{ab} = \sum\nolimits_{l=0}^{2n_x} w_{(l)}\, \mathbf{e}_{z,(l)} \mathbf{e}_{z,(l)}^\top + \mathbf{r}^a. \tag{2.2e}$$

After that, Kalman gain $\mathbf{K}$ is computed using $\mathbf{S}_{ab}$ and $\mathbf{P}_{\text{XZ}}$ as follows,

$$\mathbf{P}_{\text{XZ}} = \sum\nolimits_{l=0}^{2n} w_{(l)}\, \mathbf{e}_{x,(l)} \mathbf{e}_{z,(l)}^\top, \tag{2.3a}$$

$$\mathbf{K}(k+1) = \mathbf{P}_{\text{XZ}}\, {\mathbf{S}_{ab}}^{-1}. \tag{2.3b}$$

Finally, the corrected collective team estimations are

$$\mathbf{x}^+(k+1) = \mathbf{x}^-(k+1) + \mathbf{K}(k+1)\mathbf{r}^a, \tag{2.4a}$$

$$\mathbf{P}^+(k+1) = \mathbf{P}^-(k+1) - \mathbf{K}(k+1)\mathbf{S}_{ab}\mathbf{K}(k+1)^\top. \tag{2.4b}$$

If there is no measurement, then $\mathbf{x}^-(k+1)$ and $\mathbf{P}^-(k+1)$ (the output of prediction step) will be used as $\mathbf{x}^+(k+1)$ and $\mathbf{P}^+(k+1)$ for the next iteration of UKF, i.e.:

$$\mathbf{x}^+(k+1) = \mathbf{x}^-(k+1), \quad \mathbf{P}^+(k+1) = \mathbf{P}^-(k+1). \tag{2.5}$$

In this paper, we denote the components of the aggregated state vector X of the team

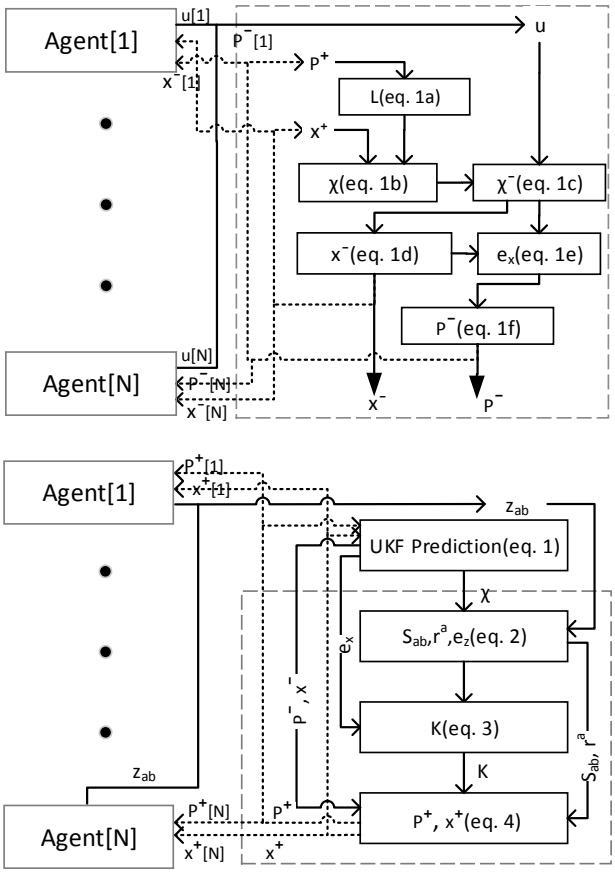

Figure 2.2: centralized UKF (top) Prediction step (bottom) Update step

corresponding to robot i by $\mathbf{x}^{-}[i]$. In a similar way, $\mathbf{P}^{-}[i]$ and $e_x[i]$ denote the corresponding portion of $\mathbf{P}^{-}$ and e_x matrices for robot i. Each sub matrices is consisted of n_i rows, where n_i is the number of states of robot i. Figure 2.3 illustrates the shape and size of the main matrices and their sub matrices[1].

In the next section, at first, we discuss decentralizing the UKF among N agents using min-cut partitioning and also decentralizing the Cholesky decomposition and covariance

[1]the size of $\mathbf{x}^{-}$ and $\mathbf{x}^{+}$ is the same. Similarly, $|\mathbf{P}^{-}| = |\mathbf{P}^{+}|$, and $|\chi| = |\chi^{-}|$. Figure 2.2 shows the task graphs for centralized UKF prediction and update steps, tagged with the equation numbers, where each robot i sends its control signals ($\mathbf{u}(k)[i]$) to the server to compute the χ^{-} using (3.3). At the end, if there is no measurement, the $\mathbf{x}^{-}(k+1)$ and $\mathbf{P}^{-}(k+1)$ is used as the inputs for the next iteration of UKF, i.e. $\mathbf{x}^{+}(k+1)$ and $\mathbf{P}^{+}(k+1)$. The navigation and other applications that is running on robot i will use $\mathbf{x}^{+}[i]$ and $\mathbf{P}^{+}[i]$. In the update step, the measured relative distance will be used to compute $\mathbf{x}^{+}(k+1)$ and $\mathbf{P}^{+}(k+1)$ through (2.2), (2.3), and (2.4), which will be used for the next iteration of UKF, and also will be sent to robots.

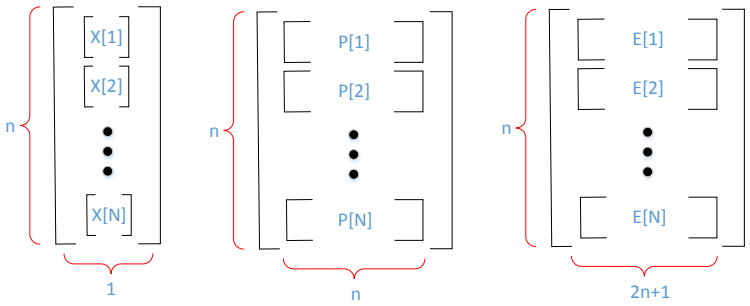

Figure 2.3: left: Vector X, middle: matrix P, right: matrix E and their sub matrices

matrix computation which is the source of coupling between the agents. Then, we optimize the number of communication links by aggregating the data that has to be sent to the same destination. Using the computation replication technique, we minimize the size of data transfer, and due to changes in the task graph, we optimize the links between agents considering the reduced data dependency between agents.

2.2.1 system configuration

In this paper, we assumed that agents have communication and computation modules. There is a wide range of network technologies each with specific characteristics such as range and speed. Two wireless standards are popular for robotics: IEEE 802.11 and IEEE 802.15. The former has a higher speed, which makes it more proper for mid-size cooperative agents. Based on the existence of infrastructure, there are several network topologies, such as star and mesh. The range of computation modules is very wide, from modules with low-end single-core microcontrollers, such as Arduino, to mid-end multi-core single computer boards, such as Raspberry Pi, to high-end processing units with GPU and FPGA.

For regular distributed applications, usually, the communication delay is several times larger than the computation delay. Therefore, there are several years of researches on reducing the amount of data transfer between agents. In this paper, we propose a method to reduce the communication for an iterative application to reduce the latency without modifying

the centralized equivalent method. We evaluate the proposed method on typical mid-end robotics hardware, i.e. multi-core single computer boards with built-in WiFi module.

2.3 Proposed Framework for Decentralizing CL

In order to decentralize UKF-based CL, we need to decouple UKF equations considering the locality of generating inputs,e.g. $\mathbf{u}(k)$, and consuming outputs, e.g. $\mathbf{x}^{+}(k+1)$ and $\mathbf{P}^{+}(k+1)$. Each robot i needs to compute $\mathbf{x}^{+}(k+1)[i]$ and $\mathbf{P}^{+}(k+1)[i]$, which are its own location parameters, and will be used locally by navigation application running on it. In UKF-based CL, all the computations of the prediction step are decoupled, except for the computation of the Choleskey decomposition ($\mathbf{L}(k)$) and $\mathbf{P}^{-}(k+1)$ (see (2.1a) and (2.1f)), where all robots need to partially compute and share them among the team (see [18]). we partition the UKF task graph into a set of subgraphs, each of which, is computed locally at the designated robot. The edge between subgraphs represents enter-robot data dependency. The communication between robots are through a wireless network which can result in a large delay depends on the size of transferred data.

Therefore, after reducing the number of communication links between agents, we minimize the size of data transfer using computation replication. In our method, the computation of decentralized UKF is identical to the computation of centralized UKF, so the accuracy of the filter will not be affected.

2.3.1 *Row-based UKF Partitioning (R-UKF):*

Each robot i at time step k generates the control signals ($\mathbf{u}(k)[i]$) locally, which will affect the location of robot ($\mathbf{x}^{+}(k+1)[i]$ and $\mathbf{P}^{+}(k+1)[i]$[2]) and has to be computed for further processing by navigation and some other higher level applications. The UKF computations can be decoupled in a way that each robot just computes what it needs [31]. However, because of data dependency between computations of some sub-matrices, such as $\mathbf{L}(k)$ and $\mathbf{P}^{-}(k+1)$ in the prediction step (3.3) and [18], each robot has to receive data from other robots.

In prediction step, the computation of $\mathbf{L}(k)[i]$ depends on the locally available $\mathbf{P}^{-}(k)[i]$, and also remotely computed $\mathbf{L}(k)[1:i-1]$. Computing $\mathbf{P}^{-}(k+1)[i]$ relies on locally generated $\mathbf{e}_x[i]$ and $\mathbf{e}_x[1:i-1]$, which has to be acquired from other robots. If each robot i has $\mathbf{L}(k)[i]$, then all other equations can be completely decoupled up to the calculation of $\mathbf{P}^{-}(k+1)$, i.e.:

$$\chi_{(0)}[i] = \mathbf{x}^{+}(k)[i], \chi_{(c,c+n_x)}[i] = \mathbf{x}^{+}(k)[i] \pm \sqrt{(n_x+\kappa)}[\mathbf{L}(k)]_c[i] \tag{2.6a}$$

$$\chi^{-}_{(l)}[i] = \mathbf{f}(\chi_{(l)}[i], \mathbf{u}(k)[i]), \quad l \in \{0, \cdots, 2n_x\} \tag{2.6b}$$

$$\mathbf{x}^{-}(k+1)[i] = \sum_{l=0}^{2n_x} w_{(l)}[i]\, \chi^{-}_{(l)}[i] \tag{2.6c}$$

$$\mathbf{e}_{x,(l)}[i] = \chi^{-}_{(l)}[i] - \mathbf{x}^{-}(k+1)[i] \tag{2.6d}$$

Note that in the context of localization, generating the transformed Sigma points of robot i, $\chi^{-}_{(l)}[i]$, only depends on the $\chi_{(l)}[i]$ and $\mathbf{u}(k)[i]$. In the next subsection, the distribution of $\mathbf{P}^{-}(k+1)$ and $\mathbf{L}(k)[i]$, which is the source of coupling between agents, is studied.

[2]We represent the corresponding n_i number of rows in $\mathbf{x}^{+}$, i.e. from row n_i*(i-1) to n_i*i, with $\mathbf{x}^{+}[i]$. To show the corresponding rows of $\mathbf{x}^{+}$ from i^{th} robot to j^{th} robot, we will use $\mathbf{x}^{+}(k+1)[i:j]$. The same representation is used for other matrices as well.

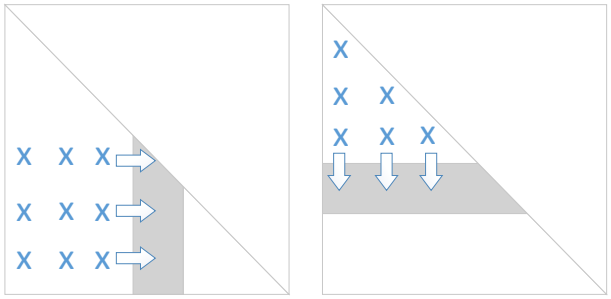

Figure 2.4: Data dependency of the Cholesky decomposition techniques: (A) column-wise approach (Cholesky-Crout Cholesky): the elements shown by $\times$ are needed to compute the grey column, (B) row-wise approach (Cholesky-Banachiewicz): to compute the gray row the elements shown by $\times$ is needed

Distributing Choleskey Decomposition, and covariance matrix

There are two major methods to traverse and compute the Choleskey decomposition of a matrix, here $\mathbf{P}^{+}(k)$): column-wise (or the Cholesky-Crout algorithm), and row-wise (or the Cholesky-Banachiewicz algorithm). Both methods use the same equations and start from the top left element of the matrix. These two methods just differ in how they traverse the matrix to compute the $\mathbf{L}(k)$, which is column by column, and row by row respectively [18]. Figure 2.4 shows the data dependency for both of these methods. To compute the $\mathbf{L}(k)$ of gray part of $\mathbf{P}^{+}(k)$, the already computed components $\mathbf{L}(k)$ (shown by $\times$), and the current value of corresponding part of $\mathbf{P}^{+}(k)$ is required. In the CL context, each robot will compute the gray part of $\mathbf{L}(k)$, and the $\times$ part has to transfer to it. After that, each robot i computes $\chi_{(l)}[i]$ based on the $\mathbf{L}(k)[i]$ that they computed or received from other robots. In the case of the column-wise method, which is used in [31], each robot i will receive the already computed columns of $\mathbf{L}(k)$, i.e. $[\mathbf{L}(k)]_{1:i-1}$, and calculate $[\mathbf{L}(k)]_i$ using $[\mathbf{P}^{+}]_i(k)$ (the gray part in the figure).

$$[\mathbf{L}(k)][i] = Chol([\mathbf{L}(k)][1:i-1], [\mathbf{P}^{+}(k)][i]) \tag{2.7}$$

$[\mathbf{L}(k)][i]$ in (2.7) contains the location information of robot i to robot N. Using this method, robot i to robot N rely on robot i, likewise on robot 1 to $i-1$, to get information that they need to calculate their χ^{-}. Therefore this method increases the data dependency between robots and decreases the reliability. If one of the robots fail, the other agents cannot calculate their location.

In the case of the row-wise method each robot i will receive $\mathbf{L}(k)[1:i-1]$ and calculate $\mathbf{L}(k)[i]$ using $\mathbf{P}^{+}(k)[i]$ as follows,

$$\mathbf{L}(k)[i] = Chol(\mathbf{L}(k)[1:i-1], \mathbf{P}^{+}(k)[i]). \tag{2.8}$$

$\mathbf{P}^{+}(k)[i]$ contains the location information of robot i and will be calculated locally, therefore there is no need to transfer it from other robots. This increases the reliability of the system because the data is generated where it will be used, and each robot can continue localizing itself even if the network between robots get disconnected. Therefore, we use the row-wise method for the Cholesky decomposition, which has the same computational demand as of column-wise method for computing UKF, and we will call it *Row-based* UKF. Using the row-wise method, each robot i can calculate $\mathbf{L}(k)$ locally, and hence it can compute the complete set of Sigma points of robot 1 to i, which is essential for replicating the computation of UKF (section 3.3). we used computation replication to further decrease the data transfer between robots in next section.

We used the up-looking variant of the Cholesky decomposition (the dual of left-looking in column-wise method) where at each iteration, one block-row of the L will be computed and all the previous block-rows is transferred at the same time. By this variant, the data can be sent in burst between robots, eliminating the transition of many packets.

Based on (2.1f), to calculate $\mathbf{P}^{-}(k+1)[i]$, $\mathbf{e}_x[1:i-1]$ has to transfer to robot i. $\mathbf{P}^{-}(k+1)[i]$

can be computed by (2.9) for the next iteration of UKF.

$$\mathbf{P}^{\text{-}}(k+1)[i] = \sum_{l=0}^{2n_x} w_{(l)} \mathbf{e}_{x,(l)}[i] \mathbf{e}_{x,(l)}[1:i]^\top + (\mathbf{B}(k)\mathbf{Q}(k)\mathbf{B}(k)^\top)[i] \tag{2.9}$$

Hence, each robot i after receiving $\mathbf{L}(k)[1:i-1]$ and $\mathbf{e}_x[1:i-1]$, computes the $\mathbf{L}(k)[i]$, $\chi[i]$, $\chi^{\text{-}}[i]$, $\mathbf{x}^{\text{-}}(k+1)[i]$, $\mathbf{e}_x[i]$, and $\mathbf{P}^{\text{-}}(k+1)[i]$ using (2.6), (2.8), and (2.9).

This partitioning will reduce the enter-robot data dependency. Note that the overall computation result will be identical to that of the centralized UKF. The only difference is the location of computation, which will affect the delay of transferring data between computations. This partitioning does not need a server for computing or sharing data, unlike the centralized and partially decentralized CL[31].

The prediction step runs frequently, and the update steps runs whenever there is a measurement in the system. Considering the partitioning of prediction step of R-UKF, we partition the update step as follows. Assuming that robot A measured its relative distance to Robot B, at first, robot B will send $\chi[B]$ to robot A. Then robot A, using $\chi[A]$, $\chi[B]$, and measurement data, will compute $\mathbf{S}_{ab}$, $\mathbf{r}^a$, and $\mathbf{e}_z$ using (2.2). After that, starting from robot 1, each robot i will compute $K[i]$ using $e_x[i]$, which is generated locally, and $\mathbf{S}_{ab}$, $\mathbf{r}^a$, and $\mathbf{e}_z$ that is computed in Robot A ((2.10)).

$$\mathbf{P}_{\text{XZ}} = \sum_{l=0}^{2n_x} w_{(l)}\, \mathbf{e}_{x,(l)}[i] \mathbf{e}_{z,(l)}^\top \tag{2.10a}$$

$$\mathbf{K}(k+1)[i] = \mathbf{P}_{\text{XZ}}\, {\mathbf{S}_{ab}}^{-1} \tag{2.10b}$$

Finally, each robot i receives $K[1:i-1]$ from robot $i-1$ to compute $\mathbf{P}^{+}[i]$ and $\mathbf{x}^{+}[i]$

((2.11)). $\mathbf{K}[1:i]$ will be sent to robot $i+1$ for computing the relative UKF update step. Note that R-UKF has exactly the same computation of a centralized UKF for CL; it is just distributed among robots.

$$\mathbf{x}^{+}(k+1)[i] = \mathbf{x}^{-}(k+1)[i] + \mathbf{K}(k+1)[i]\mathbf{r}^{a}, \tag{2.11a}$$

$$\mathbf{P}^{+}(k+1)[i] = \mathbf{P}^{-}(k+1)[i] - \mathbf{K}(k+1)\mathbf{S}_{ab}\mathbf{K}(k+1)[1:i]^{\top}. \tag{2.11b}$$

2.3.2 *R-UKF Communication Graph Refinement*

Based on (2.8) and (2.9), $\mathbf{L}(k)[i]$ and $e_x[i]$ submatrices that are computed in robot i are required to compute $\mathbf{L}(k)[i+1:N]$ and $\mathbf{P}(k)[i+1:N]$, therefore robot i have to be transferred them to all robots $i+1$ to N. We represent the N-way partitioned UKF task graph by $G=(V,E)$ where $V = V_1, V_2, \cdots, V_N$. V_i is a set of UKF computation nodes in V that belongs to robot i, and E represents the data dependency between UKF computation nodes. Figure 2.5.a represents the data dependency between the nodes. To send $\mathbf{L}(k)[i]$ and $e_x[i]$ from V_i *directly* to robots $i+1$ to N, a complete communication graph is required. The complete communication graph is not scalable, and in wide spread areas and ad-hoc networks, it imposes large communication traffic and delay. Note that broadcasting protocols such as UDP, which can be used to send the same message to many nodes, are not reliable, therefore they cannnot be used for applications such as CL. On the other hand, establishing a TCP/IP connection includes handshaking and several packet transmission.

Each node V_i $i \in \{2, \cdots, N\}$, in order to compute $\mathbf{L}(k)[i]$, requires $\mathbf{L}(k)[1:i-1]$, therefore, it has to wait till node V_1,..., V_{i-1} finish the computation of $\mathbf{L}(k)[1]$,...,$\mathbf{L}(k)[i-1]$. The computation of $\mathbf{L}(k)[N]$ is sequential, and its critical path starts from V_1 and ends in V_N. V_i has $\mathbf{L}(k)[1:i-2]$ when it computes $\mathbf{L}(k)[i-1]$, hence, $\mathbf{L}(k)[1:i-1]$ can be sent from V_{i-1} to

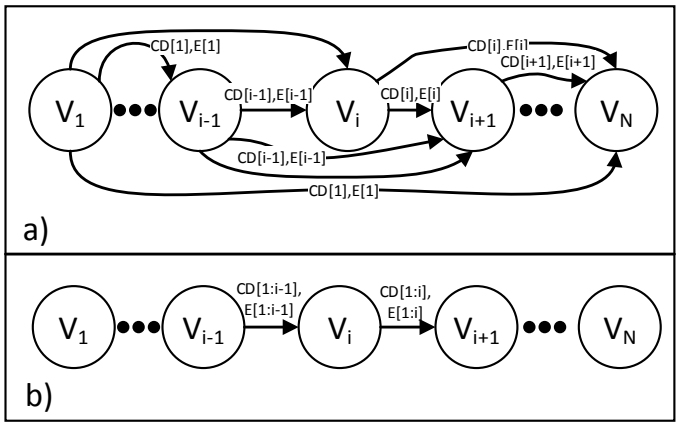

Figure 2.5: a) Decentralized R-UKF data dependency, b) refined communication graph

V_i, eliminating the need to send them directly from other nodes. Therefore, communication links between V_1 to V_{i-2} and V_i will be removed (see Figure 2.5.b). This communication model is a linear tree graph with $N-1$ links which is the minimum possible number of communication links for a connected network. Note that the underlying network can be ad-hoc or infrastructure network. This discussion is out of the scope of this paper. Reducing communication links can reduce the communication overhead. In the case of R-UKF, the computation delay is small compared to communication delay and the time overhead of establishing and maintaining a TCP/IP link.

Figure 3.5 shows the proposed UKF prediction task sub-graphs associated with robot i and $i+1$ after communication graph refinement. It shows that the UKF-based CL task graph is highly correlated and sequential. Therefore parallelism is limited. Likewise, Figure 2.7 shows the UKF update task graph.

The proper partitioning reduces the size of data transfer between agents, yet large matrices such as L and e_x have to be transferred, which leads to a large End-to-End delay. In the next section, we apply a computation replication technique called min-cut replication to further reduce the size of transferred data.

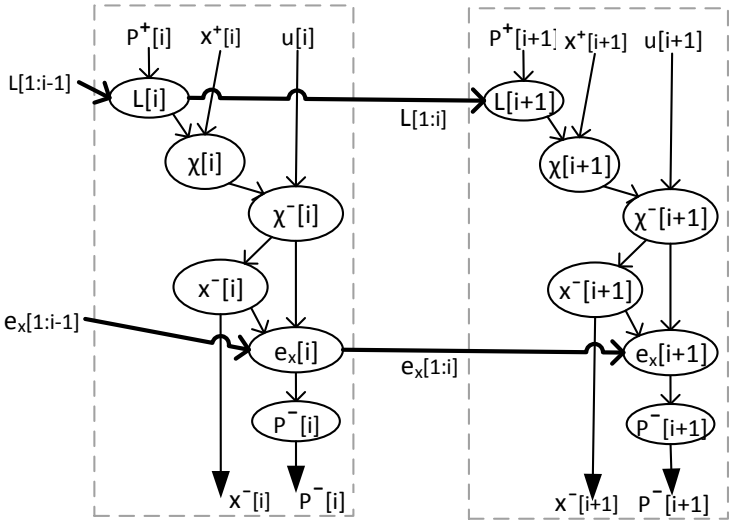

Figure 2.6: Row-based decentralized (R-UKF) prediction step

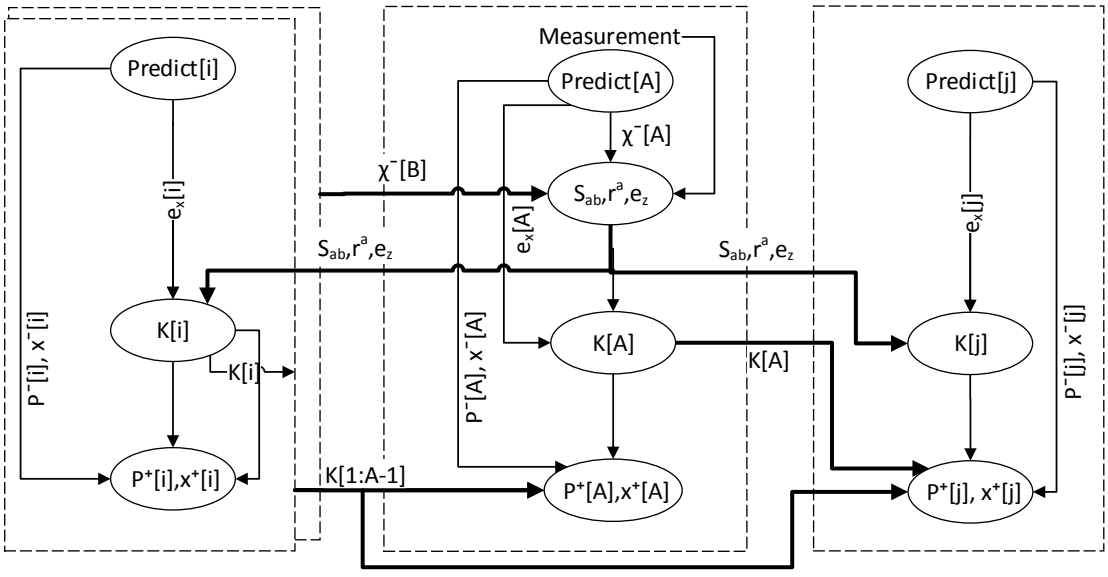

Figure 2.7: Row-based decentralized UKF (R-UKF) update step

2.3.3 *Communication Minimization by Computation Replication on R-UKF (RR-UKF):*

Replication has been used extensively in VLSI CAD partitioning to reduce the number of pins (e.g., [52, 130, 78]), and also distributed system task scheduling to reduce the communication overhead at the cost of higher computation. The replication increases the computation overhead because the same computation has to be repeated multiple times in various agents. In [70], they have introduced a method for multi-core systems to minimize the latency under power and performance constraints for a pipelined workflow using clustering and replication. In [134], authors have proposed a replication-based task scheduler for computer clusters by assigning the task paths with the highest dependency degrees to one processor. [95] has presented a method for scheduling a Directed Acyclic Graph (DAG) onto a heterogeneous system. In [136], two energy-aware scheduling methods have proposed for parallel tasks on homogeneous clusters by judiciously replicating tasks if it can improve performance without increasing energy consumption.

In this paper, after partitioning, we replicate some of the computations from a partition to another to reduce the size of data transfer between them. Figure 2.8(a) shows an arbitrary 2-way partitioned graph with two partitions of V_1 and V_2. The sum of the weight of links that represent the size of data transfer between V_1 and V_2 is 10. In this example, by replicating node b from partition V_1 in partition V_2, the size of data transfer will shrink to 4 (2+2) (Figure 2.8(c)). Therefore by more computation overhead, it is possible to reduce the communication between partitions. We propose a two-phase method to minimize the size of data transfer between agents for UKF-based CL. Since UKF-based CL is a periodic algorithm, we unroll few iterations to minimize the communication using the replication technique. Then first, we apply Min-cut Max-flow replication to the unrolled prediction step of UKF. Second, we prune the UKF update task graph given the changes in the UKF prediction task graph after replication. Then Min-cut Max-flow replication will be applied to the pruned UKF

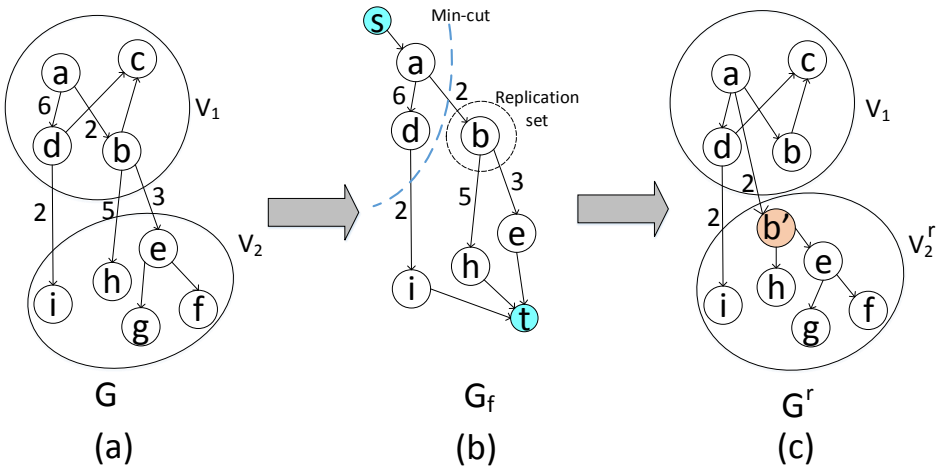

Figure 2.8: An example of Min-cut Replication a) Original graph, b) Min-cut and replication set, c) Replicated graph

update task graph. The result of the first and second step is a fully decentralized UKF with minimum data transfer.

Replication technique: We apply the min-cut replication technique on the decentralized prediction and update steps task graph of UKF-based CL to reduce the data transfer between agents. We are given a directed graph $G = (V, E)$ with N partitions represented by $V_1, V_2, \cdots, V_N$. Each partition V_i contains the nodes in V that belong to partition i, and E represents the data dependency between nodes. The set of cut edges is the set of enter-partition edges $C \in E$ that connects the nodes belonging to the different partitions. The cut edges indicate the data transfer between the robots. As an example, let's consider two partitions V_1 and V_2 as shown in Figure 2.8. We will call the set of cut edges between these two partitions I. The objective is to find a subset of nodes in V_1 such that if it is replicated in V_2, the cutsize between V_1 and V_2 is minimized. We will call this subset as *replication set* R_1. Replicating R_1 in V_2 removes the cut edges between R_1 and V_2 from the cut set, and adds the input edges to R_1 to the cut set. R_1 is a minimum replication set with respect to V_2 if the cutsize after replication is minimum among all other sets of nodes.

Figure 2.8 shows min-cut and replication set (b), and the replicated graph (c) for the example graph of G (a). To find the minimum replication set, we construct a graph $G_f = (V_f, E_f)$

based on graph G, where V_f includes nodes in V_1 that are reachable from I (node a, b, and d) and the nodes in V_2 that are adjacent to I (node i, h, and e). Two dummy nodes are added to G_f as source (node s) and sink (node t) to apply max-flow theorem. The min-cut max-flow theorem will indicate the replication set from the rest of the nodes. The replication set R_1 is the subset of nodes in V_1 starting from min-cut edges to the nodes adjacent to I (node b). After replicating R_1 in V_2, the cut edges between R_1 and V_2 (edge $b \rightarrow h$, and $b \rightarrow e$), which are enter-partition edges, can be removed from the cut set, and new edges from the inputs to the R_1 to the replicated nodes in V_2 will be added to cut set (edge $a \rightarrow b'$). If the replication set is empty, the graph will remain intact. This method reduces enter-partition communication at the cost of more computation overhead. To get the exact result from both the original and replicated task graph, the computation nodes in the replication set should be deterministic, i.e. always generate the same output for a particular input. In the case of UKF, all the computations are deterministic.

The pseudo-code for min-cut max-flow replication is presented in Algorithm 1. The input to this algorithm is two partitions of V_i and V_j. $|C|$ is the cutsize after applying the min-cut max-flow theorem on G_f. If the replication set R_j is empty, no node will be replicated and V_j will remain intact.

Algorithm 1: Min-cut Max-flow Replication

input : V_i, V_j
output: V_j^r

initialization:$V_j^r = V_j$;
Construct Graph $G_f(V_i, V_j)$;
$|C|, R_j \leftarrow$ min-cut max-flow on G_f;
if $R_j \neq \emptyset$ **then**
  $V_j^r = V_j^r \cup R_j$;
end
Return V_j^r;

Replication technique for iterative applications: For iterative stateful applications, such as UKF, the state of the system in iteration i depends on the state of the system in

the previous iterations, which can be interpreted as data dependency between iterations of the application. Applying replication technique on one iteration of the application will only address the data dependency between the agents in the current iteration, and will result in local optima. In order to fully take advantage of the replication technique, the iterative application has to be *unrolled*, similar to the concept of loop optimization in compiling programming languages.

For the iterative applications, agents at each iteration k has to acquire information from the previous m iterations ($S_{[k-m:k-1]}$), compute the current state ($S_{[k]}$), and provide it to the future iterations ($S_{[k+1:k+m]}$). Therefore, in the stationary condition, in each iteration, agents have to provide the same states that they received from the previous iterations for the next iterations. Figure 2.9 shows an iterative application and the equivalent version of it with m unrolled iterations. f function computes the new state based on the input and previous m states of the system. In the case of UKF-based CL, the state of the system just depends on the last iteration, i.e. m is 1.

Algorithm 2 shows the proposed replication method for an iterative application. To find the optimal solution, we *unroll* the application graph V by m iterations, i.e iteration k to $k+m$. Then we apply the replication technique to the *unrolled* graph V_{umi} with initialized link weights, which generates task graph V^r_{umi} with replicated computation in possibly all unrolled m iterations. The replicated sub-graph in iteration k, V^r, will be the computation that can be used with the minimum communication overhead.

Since the initial state is known to all the agents, there is no communication overhead for transferring it between agents and the link's weight is 0. In addition, the overhead of transferring states between iterations of an identical agent is zero because the states will be generated and consumed locally.

Apply Replication method on UKF based CL: We apply the replication technique to

```
1 initialize (S_{-m+1:0});
2 k = 1;
3 while True do
4 |   S_k = f(S_{k-m:k-1}, input_k);
5 |   k = k + 1;
6 end
```

```
1 initialize (S_{-m+1:0});
2 k = 1;
3 while True do
4 |   S_k = f(S_{k-m:k-1}, input_k);
5 |   S_{k+1} = f(S_{k-m+1:k}, input_{k+1});
6 |   ...;
7 |   S_{k+m} = f(S_{k:k+m-1}, input_{k+m});
8 |   k = k + m + 1;
9 end
```

Figure 2.9: unrolling an iterative application

Algorithm 2: Iterative Application Replication

input : N-way partition of one iteration of an application Task Graph V, and m
output: V^r

```
1 //Initialization
2 V_um = Unroll(V, m)
3 V_umi = initialize_weights(V_um)
4 //Replication
5 V^r_umi =Min-cut Max-flow replication(V_umi, m)
6 //Post processing
7 V^r = Group_nodes_edges(V^r_umi, k)
8 Return V^r
```

every two adjacent partitions starting from the robot 1. Min-cut max-flow replication uses weights of the edges to calculate the cutsize. In the UKF task graph, the weight of edges indicates the size of data and the wireless communication cost (e.g., the number of hops multiplied by the size of data). UKF composed of prediction and update steps. Next, we present a replication algorithm for both steps of UKF.

We proposed an algorithm based on the aforementioned min-cut max-flow replication 1 for N-way partitions of UKF which drastically reduces the cutsize for both prediction and update steps. Algorithm 5 represents the pseudo-code of Replicated R-UKF (RR-UKF). The input to the algorithm is R-UKF which is a N-way partitioned graph. We represent the R-UKF prediction(update) step by V_p (V_u). First, starting from the first partition, Min-cut Max-flow replication is applied on all adjacent partitions in decentralized UKF prediction (line [3-6]). The first partition V_p1 remains intact because it does not have data dependency

to other partitions. Then R-UKF update step with respect to the measurement between V_a and V_b is considered. To refine the R-UKF update, at first, we prune the cut edges considering the replicated R-UKF prediction step (line [7]). The reason is that the R-UKF graph has dependencies to the replicated computation in R-UKF prediction step, which moves some edges from cutsets between the partitions inside the partitions. After pruning, line [8-11] applies Mmin-cut max-flowreplication between the N-way partitions of R-UKF update similar to UKF prediction.

Algorithm 3: RR-UKF-Replication

input : N-way partition of UKF prediction Task Graph $V_p = V_{p1}, V_{p2}, V_{p3}, \cdots, V_{pN}$ and Update(a,b) Task Graph $V_u(a,b) = V_{u1}, V_{u2}, \cdots, V_{uN}$

output: V_p^r and V_u^r

1 //Initialization;
2 $V_p^r = V_p$, $V_p^r = V_p$, $V_u^r = V_u$
3 //========== UKF prediction;
4 **for** *i from* 1 *to* $N-1$ **do**
5 | $V_{pi+1}^r \leftarrow$ Replication-iterative-application ($\{V_{pi}^r, V_{pi+1}\}$,$m = 1$)
6 **end**
7 $V_u \leftarrow$ Prune-cut-edges (V_p^r, V_u)
8 //========== UKF update;
9 **for** *each* V_{ui} *and* V_{uj} *in* V_u **do**
10 | $V_{uj}^r \leftarrow$ Min-cut Max-flow replication (V_{ui}, V_{uj})
11 **end**
12 Return V_p^r, V_u^r

Figure 2.10 demonstrates the decentralized UKF prediction with min-cut replication. By replicating $\chi[1]$, $\chi^{-}[1]$, $\mathbf{x}^{-}(k+1)[1]$, $\mathbf{e}_x[1]$, $\mathbf{P}^{-}(k+1)[1], \cdots, \chi[i-1]$, $\chi^{-}[i-1]$, $\mathbf{x}^{-}(k+1)[i-1]$, $\mathbf{e}_x[i-1]$,$\mathbf{P}^{-}(k+1)[i-1]$, in V_i, the cut size will be minimized. Therefore, for RR-UKF, each partition v_i, after receiving $\mathbf{u}(k)[1:i-1]$ from v_{i-1}, will compute $\mathbf{L}(k)[1:i]$. Because of replicated computations, v_i can calculate $e_x[1:i-1]$ and $\mathbf{L}(k)[1:i-1]$ using $\mathbf{u}(k)[1:i-1]$ (new cut edge) and the $\mathbf{P}^{+}(k-1)[1:i]$ and $\mathbf{x}^{+}(k-1)[1:i]$ which are located in the same partition (calculated in the previous iteration of UKF). The size of $\mathbf{u}(k)[1:i-1]$ is much

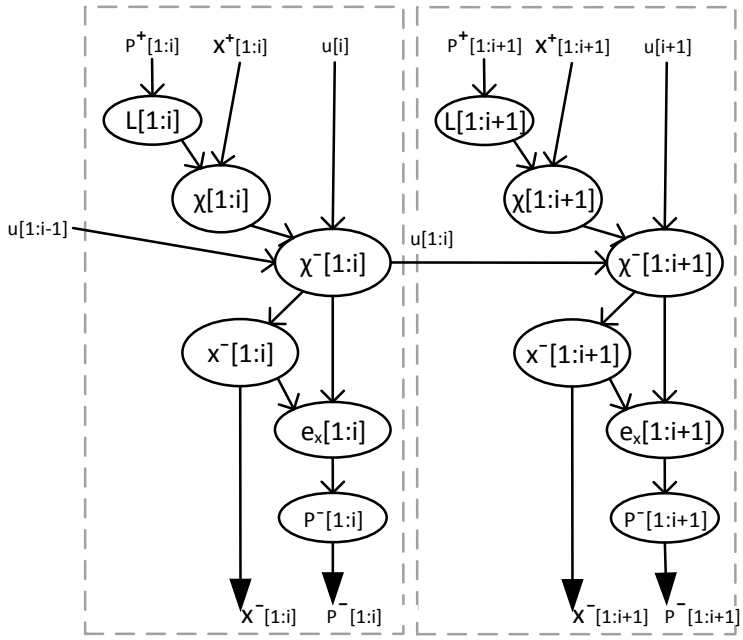

Figure 2.10: UKF prediction step after min-cut replication (RR-UKF)

smaller compared to $\mathbf{e}_x[1:i-1]$ and $\mathbf{L}(k)[1:i-1]$ ($O(n_x)$ vs $O(n_x^2)$).

$$\mathbf{P}^{+}(k)[1:i] = \mathbf{L}(k)[1:i]\mathbf{L}(k)^{\top}[1:i] \tag{2.12a}$$

$$\chi_{(0)}[1:i] = \mathbf{x}^{+}(k)[1:i], \chi_{(c,c+n_x)}[1:i] = \mathbf{x}^{+}(k)[1:i] \pm \sqrt{(n_x+\kappa)}[\mathbf{L}(k)]_c[1:i] \tag{2.12b}$$

$$\chi^{-}_{(l)}[1:i] = \mathbf{f}(\chi_{(l)}(k)[1:i], \mathbf{u}(k)[1:i]), \quad l \in \{0, \cdots, 2n_x\} \tag{2.12c}$$

$$\mathbf{x}^{-}(k+1)[1:i] = \sum_{l=0}^{2n_x} w_{(l)}[1:i]\, \chi^{-}_{(l)}[1:i] \tag{2.12d}$$

$$\mathbf{e}_{x,(l)}[1:i] = \chi^{-}_{(l)}[1:i] - \mathbf{x}^{-}(k+1)[1:i] \tag{2.12e}$$

$$\mathbf{P}^{-}(k+1)[1:i] = \sum_{l=0}^{2n_x} w_{[l]}[i]\mathbf{e}_{x,(l)}[1:i]\mathbf{e}_{x,(l)}[1:i]^{\top} + \mathbf{B}(k)\mathbf{Q}(k)\mathbf{B}(k)^{\top}[1:i] \tag{2.12f}$$

Figure 2.11 shows the update step after replication. Partition V_A has the $\chi[B]$ and $\mathbf{e}_x[1:A]$ as a result of replication in UKF prediction (V_p^r). Hence, $\mathbf{S}_{ab}$,$\mathbf{r}^a$, and also $\mathbf{K}[1:A]$ can be generated in robot A (refer to (2.2), and (2.3)). Partition V_A will send $\mathbf{K}[1:A-1]$ to partitions V_{ui} where $i < A$. Partition V_A will send only relative distance measurement to

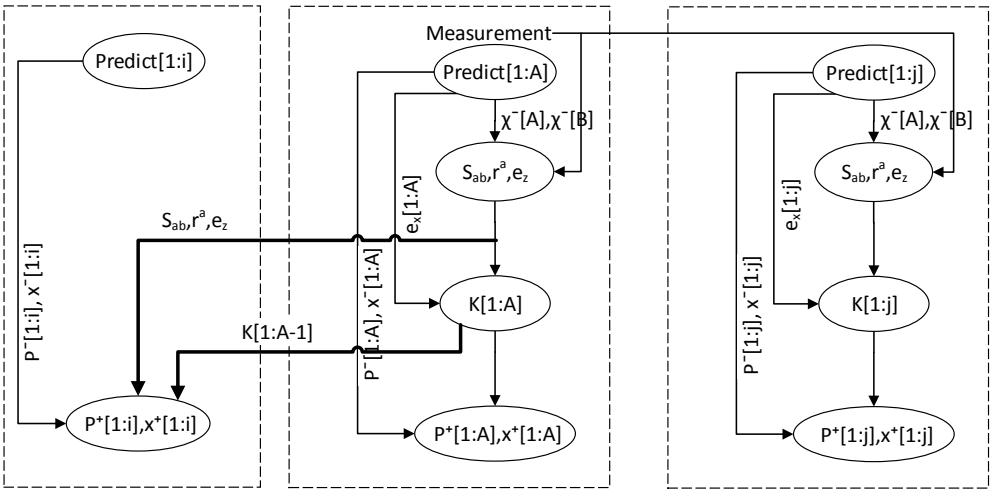

Figure 2.11: UKF update step after min-cut replication (RR-UKF)

partitions V_{ui} when $i > A$. Since $\chi[A]$ and $\chi[B]$ already exist in V_p^r, all other parameters are locally computed and hence those edges from cutset can be removed. After that, agent i, $i < A$, will just compute equation (2.13).

$$\mathbf{x}^{+}(k+1)[1:i] = \mathbf{x}^{-}(k+1)[1:i] + \mathbf{K}(k+1)[1:i]\mathbf{r}^{a} \tag{2.13a}$$

$$\mathbf{P}^{+}(k+1)[1:i] = \mathbf{P}^{-}(k+1)[1:i] - \mathbf{K}(k+1)[1:i]\mathbf{S}_{ab}\mathbf{K}(k+1)[1:i]^{\top} \tag{2.13b}$$

For the agents i, where $i > A$, after receiving measurement and computing equation (2.2), they compute $\mathbf{K}[1:i]$ using equation (2.14), and then using equation (2.13), they will compute $P^{+}(k+1)[1:i]$ and $x^{+}(k+1)[1:i]$, which will be used as input for the next RR-UKF prediction step.

$$\mathbf{P}_{\mathrm{XZ}} = \sum_{l=0}^{2n_x} w_{(l)}\, \mathbf{e}_{x,(l)}[1:i]\mathbf{e}_{z,(l)}^{\top} \tag{2.14a}$$

$$\mathbf{K}(k+1)[1:i] = \mathbf{P}_{\mathrm{XZ}}\, {\mathbf{S}_{ab}}^{-1} \tag{2.14b}$$

2.3.4 *RR-UKF Communication Graph Refinement*

Computation replication changes the task graph; some edges will be removed and some edges/nodes will be added. Therefore, the same communication graph that is proposed for the R-UKF might not be efficient for RR-UKF. By removing edges, the data dependency between partitions will reduce. In the prediction step of RR-UKF, only the control signal $\mathbf{u}$ has to be transferred between agents, and no pre-processing is required. Therefore, agents can start computing after receiving the control signal $\mathbf{u}$ from the other agents, unlike R-UKF that they have to wait till receiving the processed data from them. For the star shape networks, every agent is one hop away from the other agents. Therefore, the delay of RR-UKF with a complete communication graph will be close to that of the centralized method. However, the number of communication links and transferred packets will be greater than that of the linear chain model ($O(N)$ vs $O(N^2)$).

2.4 Experiments

2.4.1 Experimental setup

To evaluate the proposed method, we implemented centralized and decentralized UKF-based CL in C++ with an open-source linear algebra library EIGEN [42], and measured the delay and CPU utilization on a network of Raspberry Pi 3 B, with quad-core 1.2GHz 64bit CPU, 1 GB main memory, and a built-in 802.11 b/g/n WiFi module. Each board i runs Linux kernel v4.9. For our experiments, the CPU frequency is fixed to 1.2GHz. We used Linux Perf [6] to measure the CPU cycles spent on running UKF-based CL. All boards are connected to an isolated Netgear N300 wireless router in an office environment to create a realistic network environment. In the following, we report the median value for the End-to-End delay which

Table 2.1: End-to-end Delay Comparison between Partially Decentralized UKF [31] and RR-UKF

UKF End-to-End delay measured at application level (msec)						
	UKF Prediction			UKF Update		
N	[31]	RR-UKF	ratio	[31]	RR-UKF	ratio
3	20.3	12.2	1.7	17.4	9.8	1.76
5	56.1	27.0	2.1	36.8	20.3	1.82
7	130.5	40.8	3.2	59.6	22.9	2.61
9	282.4	71.9	3.9	85.7	24.8	3.45
11	529.2	91.2	5.8	115.4	32.3	3.57
13	847.3	111.9	7.6	145.3	43.2	3.37
15	1322.9	122.9	10.8	186.2	54.4	3.42

is measured from 1000 rounds of UKF. In the tables, we just reported the entries for the odd number of robots because of lack of space. In our experiments, we considered a team of robots each with 6 local states (n_s) and with measurement ($\mathbf{z}_{ab}$) and control signals($\mathbf{u}$) size of 3. We measured the End-to-End delay from robot 1's initiation to robot N's completion of each iteration of UKF-based CL, which includes the delay of both computing and data transfer on all the robots (or boards).

2.4.2 Evaluation of UKF-based CL End-to-End Delay

To demonstrate the effectiveness of our method, We compare the performance of various proposed decentralized UKF, Partially Decentralized UKF [31], and centralized UKF. Table 2.1 illustrates the End-to-End delay of RR-UKF, with aggregated communication links, and Partially Decentralized UKF for a various number of agents (N) between 3 and 15. The Partially Decentralized UKF [31] uses a server to store and share the data of UKF, however, the computation is distributed among agents. The End-to-End delay measured for RR-UKF-min-link is reduced by a factor of up to 10.8 for the prediction step and by a factor of 3.57 for the update step, compared to the Partially Decentralized UKF method in [31]. The reason is that Partially Decentralized UKF is not communication efficient.

Table 2.2: Data Communication Comparison between R-UKF and RR-UKF (min-link)

	UKF Prediction			UKF Update		
N	R-UKF	RR-UKF	ratio	R-UKF	RR-UKF	ratio
Total data bytes transmitted by all robots at application level						
3	6120	216	28.33	4174	575	7.26
5	33840	720	47.00	10606	1727	6.14
7	99288	1512	65.67	19918	3455	5.76
9	218592	2592	84.33	32110	5759	5.58
11	407880	3960	103.00	47182	8639	5.46
13	683280	5616	121.67	65134	12095	5.39
15	1060920	7560	140.33	85966	16127	5.33
Total packet frames transmitted by all robots at kernel level						
3	6	3	2.00	5	3	1.67
5	26	5	5.20	10	5	2.00
7	70	7	10.00	16	7	2.29
9	151	9	16.78	26	9	2.89
11	278	11	25.27	36	12	3.00
13	462	13	35.54	51	16	3.19
15	715	15	47.67	66	20	3.30
End-to-End delay measured at application level (in millisecond)						
3	19.2	12.2	1.57	10.6	9.8	1.09
5	50.0	27.0	1.85	23.5	20.2	1.16
7	93.7	40.8	2.30	32.7	22.8	1.43
9	167.8	71.9	2.33	45.7	24.8	1.84
11	310.5	91.2	3.40	61.5	32.2	1.91
13	440.1	111.9	3.93	74.4	43.1	1.73
15	604.4	122.9	4.92	92.0	54.4	1.69

Replication technique can significantly reduce the size of transferred data in decentralized UKF. Computation replication eliminates transmission of $\mathbf{L}$, $\mathbf{e}_x$ and $\mathbf{e}_z$ between robots and replaces with $\mathbf{u}$ and $\mathbf{z}_{ab}$ which are significantly smaller.

Table 2.2 shows the total network traffic and the median End-to-End delay measured for one iteration of UKF for both R-UKF (without replication) and RR-UKF (with replication) while performing with minimum communication links (min-link). Compared to R-UKF, the total transmitted data bytes at the application level have been drastically reduced up to 140 times for the prediction step and 7.26 times for the update step in RR-UKF.

The reduction ratio for the number of transferred packets is up to 47.67 for the prediction

step and up to 3.3 for the update step which is smaller than that of the total transmitted data bytes. The reason is that every packet frame, whose maximum size is 1500 bytes in our WiFi network, does not convey the same number of data bytes depending on the network I/O behavior of the application. For 15 agents, the End-to-End delay is reduced from 604 ms (R-UKF) to 122 ms (RR-UKF) for the prediction step, while for the update step, it is reduced from 92 ms (R-UKF) to 54 ms (RR-UKF). This significant reduction in the End-to-End delay allows each robot to react much faster in response to the resulting localization data of other robots.

To measure the overhead of decentralizing UKF, we compared the results of decentralized UKF, and centralized method in Figure 2.12. In the centralized method, agents send $\mathbf{u}$ to the server, i.e. one of the agents, and after computing UKF, it sends back the result. In the centralized method, the agent with the greatest label has selected as the server because the size of its location data is more than those of other agents. Therefore, by selecting it as the server, it is not required to send it over the wireless links. We have also implemented R-UKF and RR-UKF without applying the link reduction technique (R-UKF-max-link and RR-UKF-max-link respectively). As shown in Figure 2.12, the fastest method is centralized method, and the slowest one is R-UKF-min-link. For 15 agents, the delay of RR-UKF-max-link is higher than the centralized method by a negligible amount of 7 ms, and it is lower than RR-UKF-min-link by 40 ms. The reason is that although the delay of computation on the critical path is the same for both RR-UKF-min-link and RR-UKF-max-link, the delay of communication is higher for RR-UKF-min-link. For harsh environments, where the communication between agents is limited, the RR-UKF-max-link method is more desirable because it requires a minimum number of communication links and the size of transferred data. Both R-UKF methods (R-UKF-min-link and R-UKF-max-link) are slower than the Centralized and RR-UKF methods, which indicates the large overhead of communication.

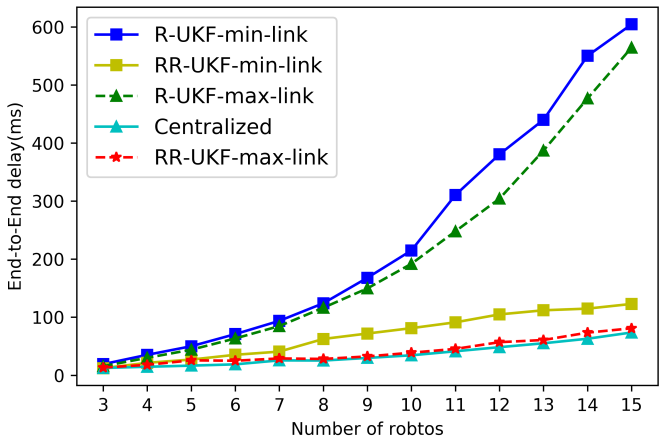

Figure 2.12: The End-to-End delay of UKF prediction step for various number of robots

2.4.3 Cost-benefit analysis

To analyze the effect of the replication technique on the CPU load, we have measured the number of the CPU cycles spent on running Centralized, R-UKF, and RR-UKF. Figure 2.13 shows the number of CPU cycles of one iteration of the prediction step for each of the agents in a system with 15 nodes. For the centralized method, the CPU utilization for all agents is almost the same except for the last node (server). The average number of CPU cycles of this method is the minimum, because no computation is replicated, and also the size of transferred data is small. For both RR-UKF-min-link and RR-UKF-max-link, the number of the CPU cycles increases quadratically, and for the last agent, it is almost the same as that in the last agent in the centralized method (server). Based on our detailed measurements, a large portion of CPU utilization for RR-UKF and centralized method is for UKF computation. However, in R-UKF a large portion of CPU cycles is spent on data serialization and transferring it. In the case of R-UKF, the number of cycles for agent 14 is higher than the other nodes. The reason is that it has to send and receive more data compared to others. The number of cycles for R-UKF is higher than that of RR-UKF on average because the size of data transfer is larger, and more CPU time has to be spent on serialization and transferring data compared to UKF computation.

Table 2.3: The CPU utilization for various time intervals for a system of 15 agents (the crossed numbers are not feasible (End-to-End Delay > Threshold delay))

method	100ms	200ms	400ms	600ms	800ms	1000ms
Maximum CPU utilization over 15 agents (%)						
R-UKF-min-link	~~12.2~~	~~6.1~~	~~3.1~~	~~2.0~~	1.5	1.2
R-UKF-max-link	~~9.1~~	~~4.5~~	~~2.3~~	~~1.5~~	1.1	0.9
RR-UKF-min-link	~~10.3~~	5.1	2.6	1.7	1.3	1.0
RR-UKF-max-link	11.8	5.9	3.0	2.0	1.5	1.2
Centralized	12.2	6.1	3.1	2.0	1.5	1.2
Average CPU utilization over 15 agents (%)						
R-UKF-min-link	~~7.4~~	~~3.7~~	~~1.8~~	~~1.2~~	0.9	0.7
R-UKF-max-link	~~7.0~~	~~3.5~~	~~1.8~~	~~1.2~~	0.9	0.7
RR-UKF-min-link	~~4.9~~	2.4	1.2	0.8	0.6	0.5
RR-UKF-max-link	5.0	2.5	1.2	0.8	0.6	0.5
Centralized	1.5	0.7	0.4	0.2	0.2	0.1

Since the CL has to run periodically at certain time intervals (T), we need to consider the CPU utilization of CL to prevent starving other applications running on the robot. We calculated CPU utilization as the percentage of CPU cycles that spent on CL in each second if CL runs each T seconds, i.e (CL_Cycles*1/T)/(Cycles_Per_Sec * num_of_cores) * 100, where num_of_cores is 4 for Raspberry Pi 3 B. Table 2.3 shows the CPU utilization. Note that the End-to-End delay of R-UKF for 15 nodes is 604ms, hence, smaller time intervals cannot be meet by R-UKF, which are crossed out in the table. For a time interval of 100ms, the RR-UKF-max-link takes 11.8% of all cores on the last node (the node with maximum CPU utilization), and 5.0% on average over all nodes. Hence, although the replication technique increases the average application level CPU utilization, it significantly decreases the CPU time for transferring data. In addition, less data transfer translates to smaller End-to-End delays.

Table 2.4 compares the characteristics of various methods. R-UKF-max-link and UKF-Centralized methods have the lowest delay, while the Partially Decentralized UKF has the maximum delay. The delay of R-UKF-min-link and R-UKF-max-link is in between which is still non-desirable for a large number of agents. For these two methods, although the

Table 2.4: comparison between various methods

Method	Comp.	Comm.	Links	Delay	Decentralized
R-UKF-min-link [98]	lowest, distributed	high	lowest	high	yes
R-UKF-max-link	lowest, distributed	high	highest	high	yes
RR-UKF-min-link [98]	highest	lowest	lowest	low	yes
RR-UKF-max-link	highest	low	highest	lowest	yes
UKF-centralized	lowest, not distributed	low	lowest	lowest	no
Partially Decentralized UKF [31]	lowest, distributed	highest	lowest	highest	no

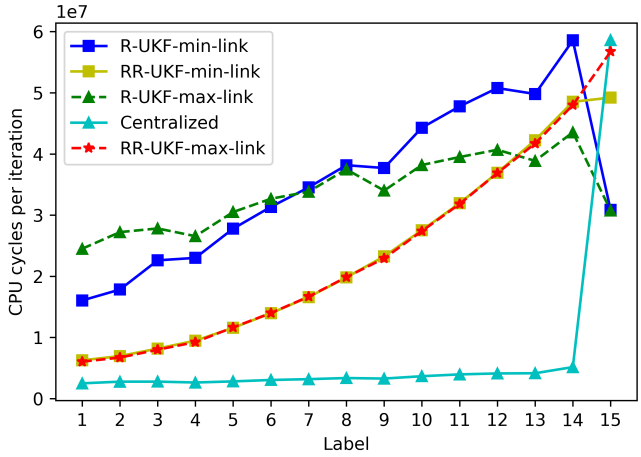

Figure 2.13: The computation overhead of different implementation of prediction step

application level computation overhead is lowest and distributed among the agents, because of the large size of data transfer, the End-to-End delay is high. In addition, due to large data transfers, the CPU overhead is high. The delay of RR-UKF (min-link and max-link) is comparable to the UKF-Centralized, and they are decentralized, which makes them more reliable. Their total computation overhead is high due to the replicated computation, however, because the size of data transfer is minimal, and the agents compute UKF in parallel, their end-to-end delay is minimum. The other important factor is the number of communication links. For the harsh environment and the situations where agents spread in large areas, and also for systems with a large number of agents, the fewer number of communication links is preferred which makes min-link methods more desirable. Therefore based on the target End-to-End delay and the scenario, the user can choose between these methods.

2.5 Conclusions

Cooperative localization is a method to increase the accuracy of localization within a network of cooperative robots. UKF is a variant of the Kalman filter, which is more accurate than other variants of the Kalman filter for localization but has more computation overhead. UKF decentralization requires a large data transfer between agents due to tight correlation among UKF computation tasks. In this paper, we proposed a framework to decentralize UKF considering computation and communication overhead. We applied computation replication to decrease the size of data transfer among robots. Our experimental results showed that the End-to-End execution time of the decentralized UKF prediction and update steps with replication is faster by up to 10.8 and 3.57 times compared to the Partially Decentralized UKF algorithm of [31]. Our measurements showed that by computation replication the delay of decentralized CL is comparable to the delay of the centralized method. In addition, we demonstrated that in the case of UKF-based CL, computation replication leads to less overall CPU overhead due to the reduced size of transferred data. As replication technique increases the computation overhead, in the next chapter, we propose a method to minimize the overall computation overhead by selectively replicating the computation.

Chapter 3

Computation-communication co-optimization

3.1 Introduction

In this chapter, we present a method to find an optimal trade-off between computation and communication of decentralized linear task chain running on a network of mobile agents. Task replication has been deployed to reduce the data links among highly correlated nodes in communication networks. The primary goal is to reduce or remove the data links at the cost of increase in computational load at each node. However, with increase in complexity of applications and computation load on end devices with limited resources, the computational load is not negligible. Our proposed selective task replication enables communication-computation trade-off in decentralized task chains and minimizes the overall local computation overhead while keeping the critical path delay under a threshold delay. We applied our approach to decentralized Unscented Kalman Filter (UKF) for state estimation in cooperative localization of mobile multi-robot systems. We demonstrate and evaluate our proposed

method on a network of 15 Raspberry Pi3B connected via WiFi. Our experimental results show that, using the proposed method, the prediction step of decentralized UKF is faster by 15%, and for the same threshold delay, the overall computation overhead is reduced by 2.41 times, compared to task replication without resource constraint.

With recent paradigm shift from cloud computing (high centralized computation) to edge and on-device computing, distributed and decentralized architectures have brought computation closer to sensor data on end devices [17][14]. Applications such as deep learning, sensor data fusion, and other compute-intensive algorithms are being decentralized and processed locally on end devices such as mobile robots and drones [43] [25]. Such decentralized algorithms, due to their distributed nature, enhance sensor fusion, system fault tolerance, and data privacy for cyber physical system (CPS) applications. However, with increasing number of compute-intensive applications running on end devices, the computation load can get beyond what low power embedded processing systems can tolerate.

There has been a great effort on reducing the wireless data transfer between the nodes during decentralization. The ideal case is when each allocated task can be fully decoupled from the tasks on other agents. However, in practice, some applications are composed of highly correlated tasks. Once decentralized, the delay/energy overhead of data transfer among the nodes cannot be neglected. This paper addresses the balance between communication and computation load during decentralization of an application running on a network of multi-agent systems.

Task replication has been deployed to reduce the data links among highly correlated nodes in communication networks [21][96]. The primary goal is to reduce or remove the data links at the cost of increase in computational load at each node. In [136] and [134], the proposed scheduling algorithms use duplication to increase the performance of parallel tasks on computer clusters connected via Ethernet while reducing the energy overhead on processors, network card, and routers. The tasks can be executed on any arbitrary

processor. In [136], a fast task scheduling based on replication is presented for heterogeneous systems. In their method, they deployed the idle time of processors for task replication without considering any computational resource constraints. The computational overhead has mostly been considered negligible compared to wireless communication delay. However, with increase in complexity of applications and computation load on end devices with limited resources, the computation load may lead to performance degradation. As a result, the communication overhead along with decentralized computation load contributes to total critical path delay of the target application.

Our proposed approach is a systematic CPS framework using *selective* task replication in order to balance communication and computation overhead in a decentralized task chain running on a network of mobile agents. We first generate multiple task replica sets for each agent, which provide trade-off between data communication and on-device computation cost. Among the replica sets for each agent, we select a configuration for each agent such that the total execution time of the decentralized task chain is met while minimizing the average computational overhead. We applied our approach to decentralized Unscented Kalman Filter (UKF) for state estimation in cooperative localization of mobile multi-robot systems [31]. Our results show that with selective task replication, the critical path delay constraint of 150 msec is met with $2.41x$ less CPU load, on average, compared to task replication without any resource constraints. To the best of our knowledge, this is the first effort on fully automated framework that orchestrates the communication and computation loads on decentralized correlated task chains to run efficiently on a network of mobile agents.

3.2 linear chain

Distributed applications can be modeled as directed acyclic graphs (DAG) where nodes represent the computations and edges represent the data dependency between the nodes.

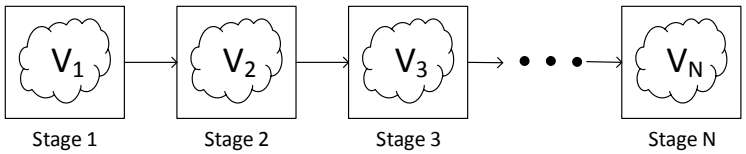

Figure 3.1: Linear chain model

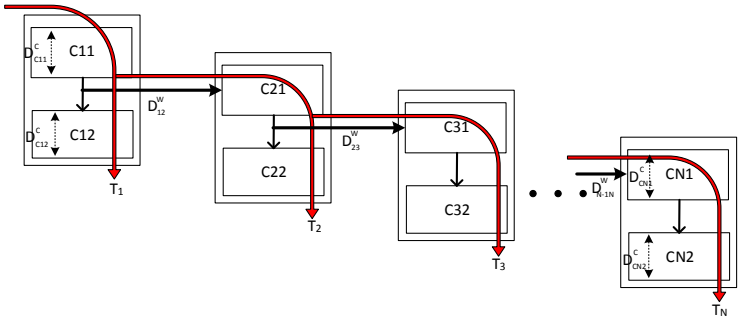

Figure 3.2: The timing of a linear chain task model

These DAGs may have special forms such as linear chain, fork, and tree [21]. In multi-robot systems and distributed embedded systems, some applications can be distributed as linear chain, for example Deep Neural Network[43], solving system of linear equations ($Ax = B$), and some families of filters such as Kalman filter [98]. In linear chain model, each stage receives data from the previous stage, and after processing, it sends data to the next stage. Figure 3.1 shows a linear chain with N stages. It can be modeled as a DAG $G = (V, E)$ where $V = V_1, V_2, \cdots, V_N$ are the partitions that are assigned to the agents (e.g. robots). Each partition is a subgraph of G containing the tasks and their local edges. E is a set of edges, representing the data dependency between partitions. The edges are directional and only connect the two consecutive agents in the chain. In this paper, the communication link between mobile agents is wireless (e.g., WiFi). Some inputs of a partition are generated locally (e.g. sensor data), and some have to be received from the previous partition. Each partition may generate local output as well as required input data for the next partition in the chain.

Figure 3.2 shows the linear chain task model and the application flow. For example, stage 2 computes $C12$ after receiving data from stage 1, and sends data to stage 3. In standard TCP/IP, the data will be copied to OS kernel. Therefore, agent 2 computes $C22$ without

waiting to send the data over the air. After receiving data, stage 3 starts computing $C31$. Each stage has a path generating the corresponding local output, and also partially contributes in the delay of the path of its successors. For example, in the same figure, $C21$ is on the critical path of stage 3($T3$), and all successor stages. The delay of path ending in stage 2 ($T2$), is calculated as the sum of the delay of computation on stage 1 (D^c_{C11}), the delay of transferring data from stage 1 to stage 2 (D^w_{12}), and the delay of computation on agent 2 (D^c_{C21}, D^c_{C22}). T^a_i represents the time by when stage i receives all input data. Hence, considering linear chain model, T_2 is $T^a_2 + D^c_{C21} + D^c_{C22}$. D^{CP}_i refers to computation time of stage i that contributes to critical path of successor stages and D^{NCP}_i refers to computation time that only contributes in the critical path ending in stage i. Hence, T^a_3 will be $T^a_2 + D^{CP}_3 + D^c_{C21}$, and the T_3 will be $T^a_2 + D^{CP}_3 + D^{NCP}_3$ ($D^{CP}_2 = D^c_{C21}$ and $D^{NCP}_2 = D^c_{C22}$). In general,

$$T^a_i = T^a_{i-1} + D^{CP}_{i-1} + D^w_{i-1,i} \tag{3.1a}$$

$$T_i = T^a_i + D^{CP}_i + D^{NCP}_i \tag{3.1b}$$

T^a_1 is 0 since the first stage does not need to receive data from any agents. The critical path delay of the task chain is the maximum of delay of all the paths (Equation 3.2).

$$T_{critical} = \max_{1 \le i \le N} (T_i) \tag{3.2}$$

Using Equations 3.1 and 3.2, it is possible to calculate the delay of critical path incrementally, starting from the first stage. Most CPS applications have to run periodically in less than a threshold delay ($T_{threshold}$). Our goal is to achieve a decentralized task chain such that $T_{critical}$ stays under $T_{threshold}$ with minimum computation overhead due to task replication.

3.3 computation-communication trade-off for two partitions

Replication is one of the well known methods to reduce the communication. It reduces the size of data transfer at the cost of increase in computational load at each agent. In this section, we present our method to generate various replicated graphs for a two-way partitioned graph. Generated task graphs provide a trade off between computation and communication. Let's assume a decentralized application divided into two partitions V_1, V_2. There may exists a subgraph in V_1, $R_1 \in V_1$ such that if it is replicated in V_2, it may reduce the cutsize between the two partitions. Task replication algorithm searches for such a replication set in V_1. After replication of R_1 in V_2, the cut edges between R_1 and V_2 are eliminated from the cut set and the input edges to R_1 are added to cutsize. Figure 3.3.a shows an example of G. Let's assume that I is the set of incoming edges to set V_2 from V_1. To find the minimum replication set, we construct a network graph $G_f = (V_f, L_f)$, where V_f includes nodes in V_1 that are reachable from I (nodes a, b, c, and d) and the nodes in V_2 that are adjacent to I (nodes f, and g) (Figure 3.3.b). To apply Min-cut max-flow theorem [38], two dummy nodes are added to G_f as a source (node s) and a sink (node t). The Min-cut max-flow theorem on this graph will separate the replication set from the rest of the network. The replication set R_1 is the subset of nodes in V_1 starting from min-cut edges to the nodes adjacent to I. As shown in Figure 3.3.c, after replicating R_1 (node c and d) in V_2, the cut edges between R_1 and V_2 (edge $c \rightarrow f$, and $d \rightarrow g$) are eliminated from the cut set and the input edges to R_1 are added to cutsize (edge $a \rightarrow c'$ and $b \rightarrow d'$). The replicated partition is called V_2^r. This method gives a replicated graph with minimum cutsize between V_1 and V_2^r.

Task replication comes with the cost of increase in the computation load in V_2. If there is no limitation on the weight of replicated nodes, the added computation load may not be efficiently manageable in the systems with limited computational resources. To control the computation overhead of the replication set, the weight of replication set should be

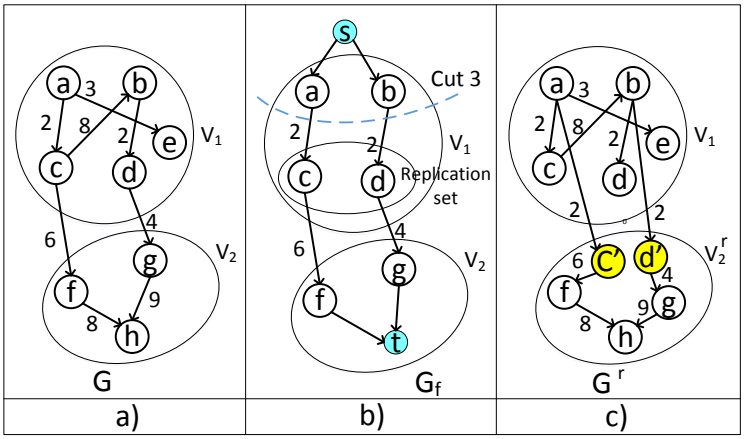

Figure 3.3: An example of Replication a) Original graph(G), b) G_f and min cut, c) replicated graph G^r

limited. i.e. $W_{R_1} \leq M$. In [126], the authors presented a method called Hyper-MAMC to incrementally find the replication set with a bounded size by starting from the initial result of Min-cut max-flow. We adopt Hyper-MAMC algorithm to replicate the task chains. However, the algorithm works between two partitions only and it does not bound the critical path delay. Using Hyper-MAMC algorithm, we generate multiple replicated sets under various CPU load constraints (M). Among those, the Pareto points will be selected as a set of configurations for each V_1 in the global search for optimum solution on the task linear chain.

The proposed algorithm (Replication-2way) is shown in Alg. 4. The weight associated with each node is the CPU cycle counts of its computation. The M varies between 0 and the sum of the weights of the nodes in V_1 to get various replicated graph (Alg. 4,line 2). Each V_2^r in $CList$ indicates a trade-off between computation load and the size of data transfer. Hence, each V_2^r in $CList$ is a point in computation-communication space. By walking through the sorted points based on the cutsize (Alg.4, line 6), and eliminating the points with higher computation load than the minimum observed thus far, we find the Pareto point V_2^r graphs.

For example, lets assume that the weight of all the nodes is 1 in the graph in Figure 3.3.a. Replication-2way will generate multiple V_2^r, as shown in Figure 3.4. For $M = 0$, no node

Algorithm 4: Replication-2way

input : 2-way partition directional Graph $G = V_1, V_2$ **output:** $CList$:list of Pareto point V_2^r

```
CList = []
for M in (0, max_inst., l) do
    V_2^r = HyperMAMC(G, M)
    CList.push((V_2^r, W(V_2^r), cutsize(V_1, V_2^r)))
end
for each (V_2^r, Comp, cut) in Sorted(CList, cutsize) do
    if Comp < Min then
        Min = Comp
    end
    else
        CList = CList − (V_2^r, Comp, cut)
    end
end
Return (CList)
```

will be replicated (Cut 1), and the cutsize will remain intact. For $M = 1$, node c will be replicated (Cut 2) which eliminates edge $c \rightarrow f$ and adds edge $a \rightarrow c'$, which reduce the

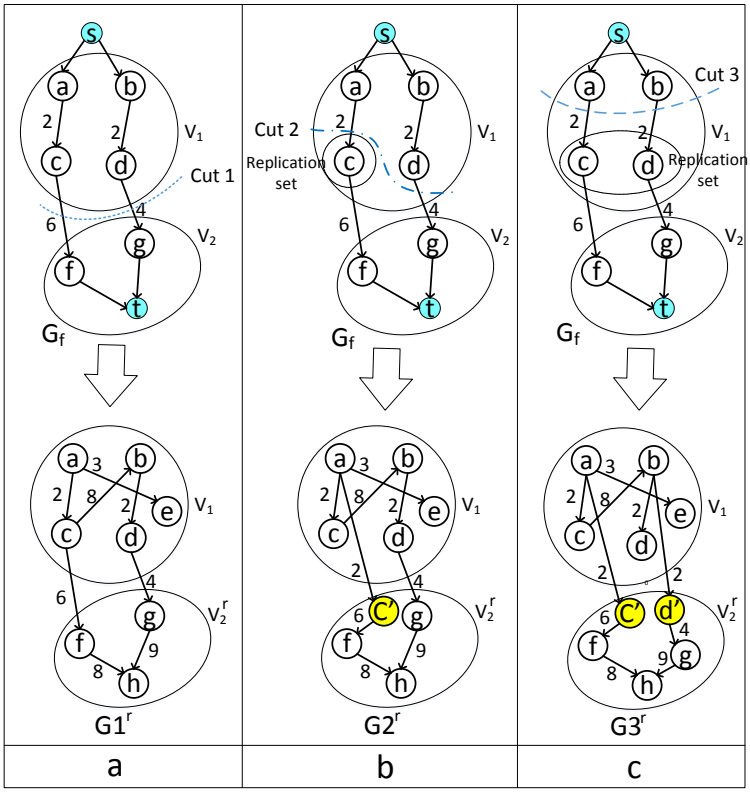

Figure 3.4: An example of constrained Replication a) $M = 0$, b) $M = 1$, c) $M = 2$

cutsize to 6. Finally, for $M = 3$, node a and b will be replicated (Cut 3) which reduce the cutsize to 4. For this example, $CList$ is $[(V1_2^r, 3, 10), (V2_2^r, 4, 6), (V3_2^r, 5, 4)]$.

3.4 Selective Replication on linear chain

Using replication, it is possible to minimize the size of transferred data. Although it reduces the communication delay, the replicated computations might increase the critical path delay. As shown earlier, we generate a group of replication sets among which there is a trade-off between their cutsizes (communication cost) and size of their replicated sets (computation cost). Next, we present an algorithm to select the optimum configuration.

Problem Formulation- we are given a N-way partition of $G = (V, E)$ represented by $V = V_1, V_2, \cdots, V_N$, and a threshold delay $T_{threshold}$. The V_i is a set of nodes in V that belongs to partition i, and E represents the data dependency between the nodes. The edges are directional and only occur between each two consecutive partitions, i.e. for all $e(u_i, u_j) \in L$, if $u_i \in v_i$, then $u_j \in v_i | u_j \in v_{i+1}$. The objective is to find a $G^r = (V^r, E^r)$ which $V^r = V_1^r, V_2^r, \cdots, V_N^r$ where V_i^r is the replicated set in V_i, such that the sum of the weight of all nodes in G^r is minimum while the delay of total critical path of G^r is less than $T_{threshold}$.

We use a dynamic programming approach to select one replicated set for each partition among the Pareto points such that the total critical path delay does not exceed a given threshold and the total computation overhead is minimized. Algorithm 5 shows the proposed method. If there are more than two stages in linear chain, applying Replication-2way on each two adjacent partitions will generate local optima. The subgraphs in previous partitions may belong to a reachable set of the current partition and hence, are potential candidates for replication sets. Therefore, to find the list of Pareto point replicated graphs of each V_i ($temp$), we search for R_i in all predecessors of V_i ($\cup_{j=1}^{i-1} V_j$) using Replication-2way (Algo. 5,

line 7).

Because the graph is partitioned as a linear chain, any feasible configuration of $V_1, V_2, \cdots, V_{i-1}$ only differs in the weight of the edges, and hence, the T^a_{i-1} and T_i will be computed incrementally based on LV^r_{i-1} and V^r_i in lines 10 and 11. Therefore, for generating V^r_i, it is not required to apply Replication-2way on all feasible configurations (LV^r_{i-1}) generated so far (PS). If the T_i of V^r_i is less than the $T_{threshold}$, it will be added to LV^r_i as a feasible replicated set considering LV^r_{i-1} (line 13). If no replication set is found to keep the critical path delay under the $T_{threshold}$, it will be removed from PS (Line 17). In line 14, the $([LV^r_i], T^a_i, Cost + W(V^r_i))$, which is a tuple of ([the list of valid configurations for agent 1

Algorithm 5: Selective_Replication

input : N-way partition of Linear Task Chain Graph $V = V_1, V_2, V_3, \cdots, V_N$ and $T_{threshold}$

output: PS

```
1  //Initialization;
2  T^a_1 = 0
3  LV^r_1 = [V_1]
4  Cost = W(V_1)
5  PS=[(LV^r_1, T^a_1, Cost)];
6  for i in [2:N] do
7      temp = Replication_2way(⋃_{j=1}^{i-1} (V_j), V_i)
8      for each (LV^r_{i-1}, T^a_{i-1}, Cost) in PS do
9          for each V^r_i in temp do
10             T^a_i = T^a_{i-1} + D^{CP}_{i-1} + D^w_{i-1,i}
11             T_i = T^a_i + D^{CP}_i + D^{NCP}_i
12             if T_i < T_threshold then
13                 LV^r_i = LV^r_{i-1} + V^r_i
14                 PS.push(([LV^r_i], T^a_i, Cost + W(V^r_i)))
15             end
16         end
17         PS = PS − (LV^r_{i-1}, T^a_{i-1}, Cost)
18     end
19 end
20 Return ((LV^r_N, T^a_N, Cost) in PS with minimum Cost)
```

to i], T_i^a, and the total weight of nodes), will be added to PS to be used in selecting the replication set for stage $i+1$. After this step, the $T_{threshold}$ of all the remaining configurations in PS will be less than $T_{threshold}$. At the end, the configuration with the lowest computation overhead ($Cost$) will be selected.

3.5 Case Study: Decentralized UKF in Cooperative Localization

The fast and accurate localization of mobile agents (or robots) is a crucial task since a delayed estimation will mislead the robots and might lead to mission failure. The time interval between executions of localization ($T_{threshold}$) should be small enough to be able to capture the motion of all the robots and provide applications with accurate and almost real time location [98]. Cooperative localization is a promising localization method in GPS-denied environment. In Cooperative Localization, robots estimate their location based on local sensor data, such as accelerometer, and correct their estimated location by the measured relative distance, using sensors such as Kinect or WiFi signal strength. Since sensors are noisy, various variants of Kalman filter is used for state estimation so as to improve the accuracy of the localization [62]. Our target application is decentralized Unscented Kalman Filter (UKF) in Cooperative Localization for multi-robot systems [31]. UKF is a recursive filter for estimating the state of a system (here location) referred to as $\mathbf{x}^+$. UKF is composed of prediction and update steps. Prediction step runs periodically, and update step runs whenever there is a measurement in the system.

Equation 3.3 shows the prediction step of UKF. K represents the time. The prediction step for the collective system with n states starts with computing the square root matrix, as a triangular matrix, of matrix $\mathbf{P}^+(k)$ using Cholesky Decomposition (CD) method. After

that, a set of $2n+1$ sample points, called Sigma Points ($\boldsymbol{\chi}$), is generated by eq. 3.3b. In the equations, $_{(c)}$ denotes the c^{th} column of the matrix. The system model function will use Sigma points and $\mathbf{u}(k)$ to generate Transformed Sigma Points (eq. 3.3c). The predicted state is the weighted arithmetic mean of $\boldsymbol{\chi}^{-}$s (eq.3.3d). Finally, the predicted covariance matrix ($\mathbf{P}^{-}(k+1)$) will be obtained by equation 3.3f using prediction error $\mathbf{e}_x$ (eq.3.3e). In the equations, $c \in \{0, \cdots, 2n\}$, $l \in \{1, \cdots, n\}$, and w are system defined constant.

$$\mathbf{L}(k) = CD(\mathbf{P}^{+}(k)) \tag{3.3a}$$

$$\boldsymbol{\chi}_{(0)} = \mathbf{x}^{+}(k), \boldsymbol{\chi}_{(l,l+n_x)} = \mathbf{x}^{+}(k) \pm \sqrt{(n_x+\kappa)}[\mathbf{L}(k)]_l \tag{3.3b}$$

$$\boldsymbol{\chi}^{-}_{(c)} = \mathbf{f}(\boldsymbol{\chi}_{(c)}(k), \mathbf{u}(k)), \quad c \in \{0, \cdots, 2n_x\} \tag{3.3c}$$

$$\mathbf{x}^{-}(k+1) = \sum_{c=0}^{2n_x} w_{(c)}\, \boldsymbol{\chi}^{-}_{(c)} \tag{3.3d}$$

$$\mathbf{e}_{x,(c)} = \boldsymbol{\chi}^{-}_{(c)} - \mathbf{x}^{-}(k+1) \tag{3.3e}$$

$$\mathbf{P}^{-}(k+1) = \sum_{c=0}^{2n_x} w_{(c)} \mathbf{e}_{x,(c)} \mathbf{e}^{\top}_{x,(c)} + \mathbf{B}(k)\mathbf{Q}(k)\mathbf{B}(k)^{\top} \tag{3.3f}$$

$\mathbf{P}^{-}(k+1)$ and $\mathbf{x}^{-}(k+1)$ might be used as $\mathbf{P}^{+}(k+1)$ and $\mathbf{x}^{+}(k+1)$ for the next UKF iteration.

In the decentralized UKF, each partition i is assigned to robot i. Each robot i has to compute its location, i.e. $\mathbf{P}^{-}[i]$, and $\mathbf{x}^{-}[i]$ using its locally generated control signals($\mathbf{u}[i]$), and $\mathbf{L}[1:i-1]$ and $\mathbf{e}_x[1:i-1]$ that receive from the last robot (robot $i-1$) [98][31]. Figure 3.5 shows the task graph of robot i and $i+1$ for decentralized UKF. Therefore, the task graph of decentralized UKF can be modeled as a *linear chain* (Figure 3.1), and its critical path delay can be estimated by equation 3.1. There are two edges, $\mathbf{L}[1:i]$ and $\mathbf{e}_x[1:i]$, between each robot i and robot $i+1$. By applying *Replication_2way* (Algo. 4), various edges between the partitions can be eliminated and multiple replication sets are generated. We refer to them based on their replication level in ascending order, such as UKF-M1 and

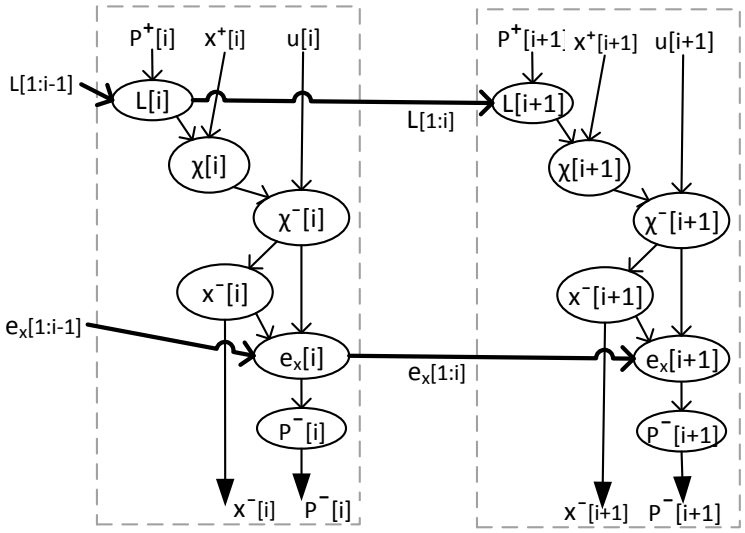

Figure 3.5: Decentralized UKF without replication

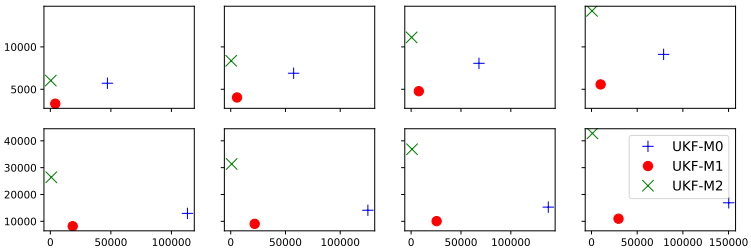

Figure 3.6: The UKF-M0, UKF-M1, and UKF-M2 computation-communication points, N=15, X-axis: communication, Y-axis: computation

UKF-M2. In UKF-M1, $\mathbf{L}[1:i]$, and in UKF-M2, $\mathbf{u}[1:i]$ will be transferred between agent i and agent $i+1$. We call the original task graph of decentralized UKF as UKF-M0, where no task is replicated. The UKF-M0 is inferior to UKF-M1 because it has more communication and computation (Figure 3.6). Although in UKF-M1, some of the UKF tasks are replicated, the CPU overhead of UKF-M0 can still be higher. Due to high volume of data transfer in UKF-M0, more CPU cycles are consumed for data serialization and TCP/IP stack. The output of *Selective_Replication* for UKF (called SR-UKF) is a list of UKF task graph for all agents, including UKF-M1 and UKF-M2.

Table 3.1: The number of CPU cycles for various time intervals for a system of 15 agents

Proposed method (SR-UKF)			
T_{thre}(ms)	Configuration	T_{crit}(ms)	CPU cycles(K)
135	[1,1,1,1,1,1,2,1,1,1,1,1,1,2,1]	134	125,427
140	[1,1,1,1,1,1,1,1,1,1,1,1,2,1,1]	138	114,074
150	[1,1,1,1,1,1,1,1,1,1,1,1,2,1,1]	138	114,074
160	[1,1,1,1,1,1,1,1,1,1,1,2,1,1,1]	151	109,593
170	[1,1,1,1,1,1,1,1,1,2,1,1,1,1,1]	169	101,964
180	[1,1,1,1,1,1,1,1,2,1,1,1,1,1,1]	175	98,751
Reference configurations (not SR-UKF)			
—	FR-UKF	145	275,191
—	NR-UKF	827	136,376

3.6 Experiments

We implemented out proposed framework on a network of single-board embedded devices, i.e., Raspberry Pi 3 B, with quad-core 64bit CPU, 1 GB main memory, and a built-in WiFi module of 802.11 b/g/n. During our experiments, the CPU frequency is set to 600 MHz, and we used all four cores of the CPU. Each device i runs Linux kernel v4.9 and an application process that implements robot i's task for decentralized UKF in C++ with an open-source linear algebra library EIGEN [42]. To create a realistic network environment, we set up an isolated WiFi network with one access point, Netgear N300 wireless router and N boards. In the following, for all parameters, we used their median values obtained from 500 rounds of UKF. In our experimental studies, we considered robotic team scenarios with robots with 6 local states and with measurement and control signals size of 3. In our implementation, we measured and analyzed the total number of CPU cycles used for the decentralized UKF task, and the critical path delay of each execution cycle of decentralized UKF, which include both the computation and the communication delays on all the robots (or boards). We used Linux Perf to measure the number of CPU cycles associated with decentralized UKF [6].

In our proposed method, task replication is selectively applied such that the total CPU utilization is minimized while the total critical path delay ($T_{critical}$) is less than $T_{threshold}$. To show the effect of $T_{threshold}$ on the result, we run the algorithm for various values of

$T_{threshold}$ on a team of 15 robots. Table 3.1 shows the result for one iteration of decentralized UKF. UKF-M1 and UKF-M2 are represented as 1 and 2, respectively. $T_{threshold}$ and $T_{critical}$ are in msec, and we report the total number of CPU cycles (in K). To show the effect of selective replication, we compared it with configuration where the Min-cut max-flow replication has been applied without any resource constraint and hence, the decentralized UKF task graphs has been Fully Replicated, called FR-UKF, adopted from [98]. To show the effect of replication, we also run the decentralized UKF with No Replication, called NR-UKF. The best $T_{critical}$ that can be achieved by FR-UKF is 145ms, which is higher than the $T_{critical}$ of SR-UKF (134 ms). This is due to the fact that the computation delay of the last node in FR-UKF is higher than the CPU time saved by task replication. The $T_{critical}$ of NR-UKF is 827ms which shows the speedup gained by reducing the size of transferred data using replication. By increasing the $T_{threshold}$, as expected, the total number of CPU cycles is reduced using the proposed method.

To show the effect of our proposed method with the same $T_{threshold}$, we set the $T_{threshold}$ to the minimum value that can be met by the FR-UKF. Table 3.2 shows the total number of CPU cycles and the $T_{threshold}$ for both methods for various number of agents. In all cases, our proposed method (SR-UKF) outperforms the FR-UKF. For 15 agents, the total number of CPU cycles associated with decentralized UKF task is improved by $2.41x$ compared to FR-UKF.

3.7 Conclusions

we presented a method to find an optimal trade-off between computation and communication of decentralized linear task chain running on a network of mobile agents. Our proposed selective task replication enables communication-computation trade-off in decentralized task chains and minimizes the overall local computation overhead while keeping the critical

Table 3.2: The number of CPU cycles for FR-UKF and SR-UKF with the same $T_{threshold}$

N	T_{thre}(ms)	FR-UKF	SR-UKF	Ratio
5	54	5451	3728	1.46
6	58	10001	6422	1.55
7	89	16909	9428	1.79
8	93	27114	15264	1.77
9	104	41340	23219	1.78
10	109	60725	32153	1.88
11	118	86472	43190	2.00
12	123	119167	56763	2.09
13	125	160540	72933	2.20
14	129	212410	91878	2.31
15	149	275191	114074	2.41

path delay under a threshold delay. We applied our approach to decentralized UKF, and demonstrated and evaluated our proposed method on a network of 15 Raspberry Pi3 devices. Our experimental results show that, using the proposed method, the overall computation overhead of decentralized UKF is reduced by 2.41x, when compared to computation replication method without resource constraint.

In the next chapter, we consider applications that cannot be distributed among the agents, and has to be run in a centralized mode on node with a stronger processing unit such as an edge.

Chapter 4

Time-coordinate computation-communication-sensing in edge computing systems

4.1 Introduction

The networked embedded devices might have access to a remote node with a strong processing unit, such as a FPGA [104], a GPU [59], or a custom design hardware [46, 47]. This node can be a local server, an edge or a cloud. During this chapter, we refer to networked embedded device as end device, and we focus on edge computing, where the end devices sends their data to the edge for processing. Usually, the edge resources are shared among multiple applications and either a fixed schedule [119, 110], or a flexible schedule [100, 74, 103], can be used to allow an efficient utilization of the resources. In this chapter, an edge with a fixed time slotted schedule has been studied. In a time slotted non preemptive system, if the data arrive earlier than the start of allocated time slot, the data has to wait on the edge. On

the other hand, if the data arrives after the time interval that the computing resource can accept the data to process, it has to wait for the next corresponding time slot. Therefore, the data arrival time affect the wait time on the edge[1].

In an *ideal environment*, the network delay is fix, hence the arrival times. Therefore the arrival times can be adjusted to match with the corresponding time slot by simply measuring the network delay and adjusting the transmission time based on it. One way for data arrival time staggering is to shift the end device transmission time, which means the sensor data has to wait on the end device to be sent. However, in many applications, such as monitoring or object tracking, the data should be as fresh as possible, which is referred as Age of Information (AoI). Moreover, changing the time interval between the sampling times can affect the application quality, such as object tracking and video compression applications. In [72], the author reported increasing the false detection rate up to 31% for a frame latency of 16ms. Therefore, instead of adjusting the transmission time, we adjust the sensing time, which will delay the transmission time as well. In practice, there are many variables that affects the data arrival times, including the network delay and end device preprocessing time.

On way to adjust the transmission time is to synchronize the device clock with the edge, predict the delay of parameters that might affect the arrival time, such as preprocessing and network, and then explicitly set the end device sensing time considering the sum of all the predicted latencies and the actual arrival time.

4.2 Related works

Time synchronization between multiple wirelessly connected devices is challenging because environment affects the time synchronization. For instance, the number of hops [111], and

[1]A predictable arrival time leads to better resource allocation and task scheduling in a systems with flexible schedule as well [119]

temperature [35], which changes the clock rate. In Timestamp-Free Network Synchronization [24, 121], the client initiate the synchronization with a server by sending a packet to the server, then the server after a certain time, response to the client. Afterwards, client calculate the clock skew based on the sending and receiving time. Some papers such as [127, 77, 45] used Kalman filter to synchronize clock between wirelessly connected devices. Their motivation was to make the sampling time synchronized. Synchronized sampling increases the data quality for IoT systems that the data is gathered from multiple end devices, such as drones taking pictures in a rescue mission [87], or IoT based bridge health monitoring systems [84]. In [105], multiple cameras are used for localizing ground robots with synchronized sampling time, as it is necessary for shape reconstruction. Time synchronization can be done through time servers using protocols such as NTP, or using GPS. Synchronized sampling is also used for Cooperative localization, where synchronized data that is collected from multiple robots has to be fused using algorithms such Kalman filter.

There are some methods to predict the network delay based on estimation methods such as Kalman filter and Markov model[39]. [29] measured and reported the delay of network from users to major cloud operators, and concluded that the network links between major network and clouds are efficient, and the majority of network delay variation comes from the links between users and the internet providers. Predicting the network delay and adapting the end device based on it has been studied in many papers. [56] proposed an adaptive methodology that changes the compression ratio based on the predicted network bandwidth between the user and a server. In [93, 132, 10, 131], the computation offloading to edge adapts to the network delay and available resources to optimize various parameters such as energy consumption and end to end delay. Predicting the preprocessing delay is also challenging and needs exhaustive profiling for each device and application. For example, [97, 98] used profiling to estimate the computation and communication delay to select the right amount of computation replication on a set of networked agents.

Each of these steps needs a complicated estimation method and extensive profiling. These methods are usually not general and they are specific to a network technology or an end device hardware. Moreover, implementing them is time consuming and has a high engineering cost. Considering the wide range of devices and network technologies in networked embedded systems, deploying the above method is not practical.

In this chapter, we propose a sensing-communication-computation time coordinate framework to minimize the Age of Information. First, we compute Target arrival time for periodic applications based on the arrival time distribution, and then we provide a solution to estimate the current arrival time, which is used for finding the staggering time. If the end device shifts the sensing time by the gap between the Target arrival time, and the estimated arrival time, the actual arrival time will get closer to the Target arrival time. Below we describe two main challenges in adjusting the data arrival times in our proposed method:

- **Target arrival time**: The arrival time is not fixed and changes over time, hence, we treat the arrival time as a random variable. At first, considering the arrival time distribution, we find the best arrival time (Target arrival time) that minimize the task wait time. Since the arrival time is a random variable, after finding the optimal arrival time, we need to adjust the sensing time of the end device by the gap between the Target arrival time, and the actual arrival time. Since the actual arrival time is noisy and changes over time, we need to compute the estimated arrival time.
- **Estimated arrival time**: Since the edge does not have access to arrival times ahead of time, and the arrival time is noisy, finding an estimated arrival time that represents the current arrival time is challenging.

The proposed method does not require network and processing delay estimation techniques, and only depends on the data arrival times, which is easily accessible from the edge. In addition, it does not need a prior knowledge about the network delay distribution, and

estimate the arrival time during runtime when receiving a streaming data. At first, we provide a delay model for the system, and then we provide a method to compute the Target arrival time. Afterwards, we propose two methods based on 1) sliding window averaging, and 2) Kalman filter to estimate the arrival time. We use the Target arrival time and the estimated arrival time to send 'Wait' command to the end device. Since in some synchronized networked embedded systems, such as Cooperative Localization, the sensing time of all the participating agents should be at the same time, we extend our method to support these systems as well. At the end, we evaluate the proposed method over a multiple networked embedded boards (Raspberry Pi), and a computing platform as the edge.

4.3 system model and motivation

The end device senses a parameter using a sensor (e.g. camera), preprocesses it (e.g. encoding, edge detection), and sends the data to the edge using wired/wireless network (e.g. WiFi, Ethernet) for further processing (a task such as e.g. image classification using DNN). We refer to the data that end device sends to the edge as data or frame interchangeably. For an end device that sends data periodically with the frequency of f to the edge, the data arrival time can be modeled as $T_a = T_{send} + D_{network}$ where $T_{send} = T_{sense} + D_{preprocessing}$. Data arrival time affects the performance of the system. For an edge that is optimized for a certain arrival time , if the actual arrival time $T_{arrival}$ skews from it , the performance might be negatively affected. For example, [97] has proposed a method to reduce the response time by optimizing the task mapping and schedule considering a fixed arrival time.

Arrived data have to wait on the edge till the consumer (here computation resource) becomes available. We represent the wait time by D_{wait}. Therefore, the delay between the sensing time on the end device and processing time on the edge of data i, which is also referred as Age of Information will be:

Source	Delay parameter
CPU freq	$D_{preprocessing}$
internal clk skew	T_{sense}
change in data size	$D_{network}$, $D_{preprocessing}$
CPU/memory utilization	$D_{preprocessing}$
hops, congestion, movement	$D_{network}$

Table 4.1: the source of data arrival time variation

$$AoI(i) = T_{process}(i) - T_{sense}(i) = D_{preprocessing}(i) + D_{network}(i) + D_{wait}(i) \tag{4.1}$$

In this chapter, we focus on decreasing the average AoI by reducing the average D_{wait}. Reducing $D_{preprocessing}$ and $D_{network}$ has been the subject of many research papers, and it is out of the scope of this dissertation. Many of existing methods are orthogonal to our approach and can be used together for further enhancement of the response time of the system.

There are many factors that might cause data arrival time variation. Table 4.1 shows some of these variables and the parameter that they might affect. The internal clock skew can change the T_{sense} since the device cannot track the right sensing times. The data size might change from time to time for reasons such as data compression and change in a data resolution. For example, application might need to capture pictures with higher resolution, which means an increase in the data size. The CPU/memory utilization and also CPU frequency affect the delay of preprocessing. The other source of arrival time variation is network which is affected by many parameters such as the number of hops, congestion, and the device mobility.

For example, Figure 4.1 shows the effect of end device CPU frequency on the data arrival time. The x axis represents the frame number, and the y axis shows the relative arrival time compared to expected arrival time in milliseconds, i.e $mod(T_{arrival}, 1000/FPS)$. The end device is a raspberry pi that captures pictures using a camera, and send the compressed

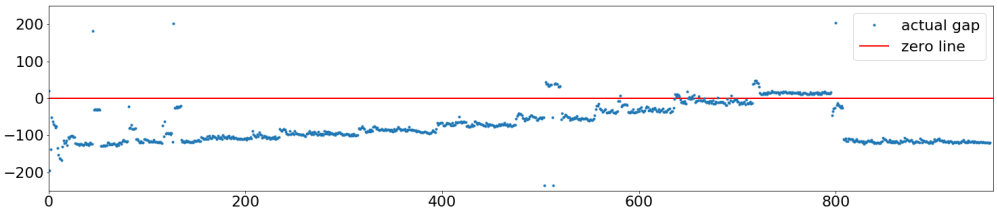

Figure 4.1: Data arrival times for the end device with varying CPU frequency

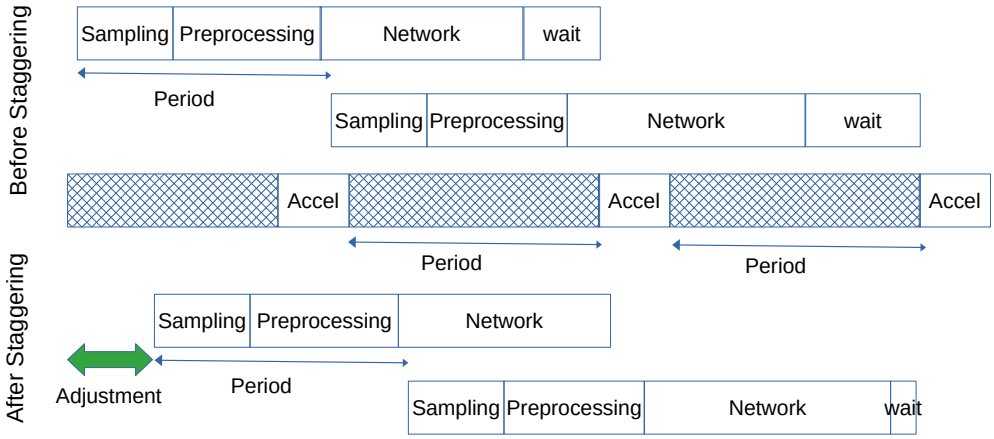

Figure 4.2: Wait time before and after staggering for one device

frames to the edge using WiFi network with the frequency of 2 for 5 minutes. Note that in this figure, the time interval between sensing time is exactly 500ms. For this figure, we changed CPU frequency gradually from 1.5GHz to 600MHz and then back to 1.5GHz. The difference between arrival times is around 150ms.

Developing a model to accurately predict the effect of each parameters on the data arrival time is hard. Not only there are many time varying parameters which requires extensive profiling, but also there are a wide range of end devices, and this diversity makes developing a single model for all those devices impossible. Therefore, in this chapter, we propose an online method to model the effect of all these parameters all together and bring the actual arrival times as close as possible to the Target arrival time.

Figure 4.2 shows the effect of sensing time on the D_{wait}. Each row shows the timeline of a data, which includes sampling, and preprocessing on the end device, sending through network, and waiting on the edge for the computation resource to become available. In the

middle, the timeline of computation resource on the edge has been shown, where dashed rectangle means the computation resource is not available. The edge works in a time slotted mode and the resource timing is based on a edge resource time table (ERTT). As it is shown in the figure, by staggering the sensing time, the wait time is reduced. For one of the data (3rd data), the wait time is zero because the computation resource is available when the arrives. In an *ideal system*, where the $D_{preprocessing}$ and $D_{network}$ are fixed, the Age of Information (AoI) can be minimized by staggering the sensing time by D_{wait}, which reduces the D_{wait} to 0. However, the delay varies over time. Therefore, we propose an online method to minimize the average D_{wait}.

We model the access to the edge computing platform as a time slotted system. Each time slot is dedicated to a task that processes the data sent by an end device. For the sake of simplicity, we assume that the time interval between each two slots, that correspond to data sent from the same device, is $1/f$, a.k.a $Period$, and it is equal to the time interval that the end device capture data using its sensor. Each slot k, based on ERTT, starts at $T_{slot,k}$, and computation resource accept data for processing for a duration of D_{slot}. Note that the total time that the computation resource is available on the edge is more than D_{slot}. If data i arriv es between $T_{slot,k}$ and $T_{slot,k} + D_{slot}$, the edge can process $data_i$ right away, i.e. $D_{wait,i} = 0$. Otherwise, it has to wait for the computation resource to become available. If x is the data i arrival time, the $D_{wait,i}$ will be:

$$\begin{cases} T_{slot,k} - x, & \text{if } x < T_{slot,k} \\ 0, & \text{if } T_{slot,k} <= x < T_{slot,k} + D_{slot} \\ (T_{slot,k} - Period) - x, & \text{if } x >= T_{slot,k} + D_{slot} \end{cases} \tag{4.2}$$

In an ideal system, the data arrival times matches with T_{slot}, which makes the D_{wait} zero.

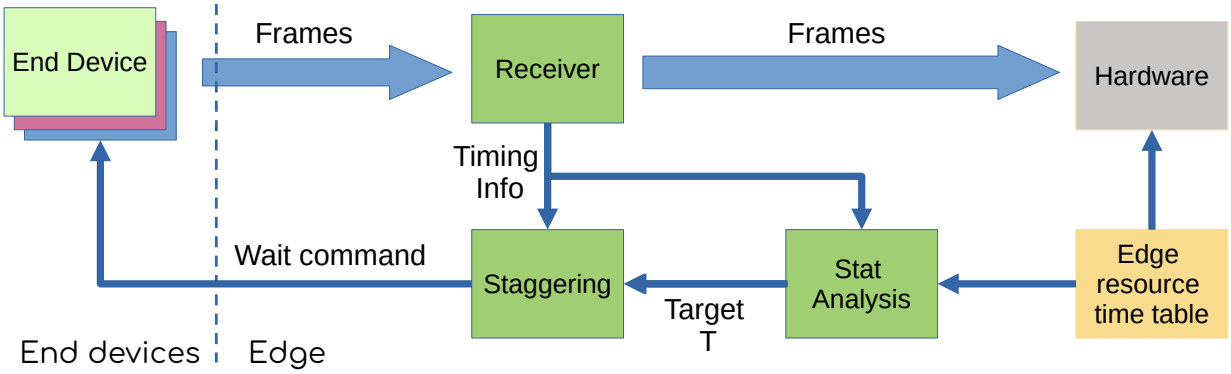

Figure 4.3: The proposed method to minimize the AoI

However, in practice, arrival times fluctuate over time, and hence, D_{wait} will increase. Based on eq. 4.2, by staggering the arrival times (x), we can reduce the D_{wait}.

In this chapter, at first, we propose a method to compute the Target arrival time during runtime to minimize the average D_{wait} based the timing constraints of end devices. Enforcing a certain timing is challenging due to many parameters that affects the arrival times. Hence, we also propose a Staggering module that estimate the arrival time, and stagger the sensing time to make the arrival times as close as possible to the desired arrival time using a feedback loop. The proposed method is online and adapts to the changes in the system during runtime.

4.4 Proposed method

Figure 4.3 shows the overview of the proposed method. For each end device, edge receives the data. Then, based on the edge resource time table (ERTT), it computes the Target arrival time using statistical analysis. Afterwards, the staggering module computes the staggering time based on the arrival times and the Target arrival time, and send it as a 'Wait command' to the end device. Note that on the end device, no profiling or measurement has to be done.

Algorithm 6 shows the proposed method for the server (here the Edge). For each end device $device_i$, edge receives the data, and tags it with the arrival time $T_{a,i}$. Then it computes

the Target arrival time($T_{a,i}^{target}$) that minimize the D_{wait} considering the edge resource time table (T_{slot} and D_{slot}) and the statistical information of the arrival times ($T_{a,i}$). Staggering module, after computing the the estimated arrival time, calculates the staggering time, and then, sends a command to the end device to adjust the sensing time, if it is necessary. This method is online because the distribution of the arrival time changes over time.

Algorithm 7 shows the proposed method for the client (here End device). The only difference between this algorithm and a regular client algorithm is that the end device needs to receive 'Wait' command from the server (edge), and changes the time interval between sampling according to it, once per each received command. Therefore, the proposed method does not have computation overhead for the end device.

Algorithm 6: sensing-communication-computation time coordinate framework - server side (Edge)

input : $ERTT$
output: $TaskQueue$, //*global shared task queue*

```
1  //Multi thread: one per end device
2  for each Device device_i do
3      while True do
4          //Receive
5          data, T_{a,i} = ReceiveTask()
6          //Stat_analysis
7          T_{a,i}^{target} = GetTargetArrvlTime(T_{a,i}, ERTT)
8          //Staggering(Kalman/sliding window averaging)
9          Stagger(T_{a,i}^{target}, T_{a,i})
10     end
11 end
```

4.5 System types based on sensing time

In this section, we study two types of systems based on the sensing time synchronization between agents. we categorized the applications that sends tasks to the Edge with a fixed period based on their sensing time characteristics. For each of these application types, we

Algorithm 7: sensing-communication-computation time coordinate framework - client side (End device)

```
   input : FPS
 1 while True do
 2 |   //sense
 3 |   SensorData, T_sense ← Sense(sensor)
 4 |   //preprocessing: e.g. encoding
 5 |   data ← Preprocess(SensorData)
 6 |   //Send
 7 |   SendData(server, data)
 8 |   //Sync: Non blocking
 9 |   staggeringtime = RecieveCommand(Server,'Wait')
10 |   //timing: blocking
11 |   Sleep(1000/FPS − T_sense − staggeringtime)
12 end
```

model the arrival times and the expected wait time, and using statistical method, we compute the Target arrival time to decrease the wait time (Figure 4.4).

Type 1: Single-agent sensing staggering: For this type, the sensing time of each individual application can be adjusted such as video streaming, and single node object tacking. Therefore, the edge can send a Wait command to each individual device to change the sensing time. Note that changing the sensing time does not affect the sensing rate, but they are only shifted.

Type 2: Synchronized multi-agent sensing staggering: In these type, multiple end nodes measure a parameter with certain time interval from each other, and send it to the edge to accelerate. In other word, the relative sensing time is fixed, but its absolute time can be changed. The device clock of coordinated nodes has to be synchronized using a time server with protocols such as NTP, or GPS. For these type of applications, the sensing time of all the coordinated end nodes has to be changed together. Because there is data dependency between measurement of the end device, data have to be processed with a certain order on the edge. Cooperative localization, and bridge health monitoring are two examples of these applications that all the devices has to capture the data at the same time. Time skew

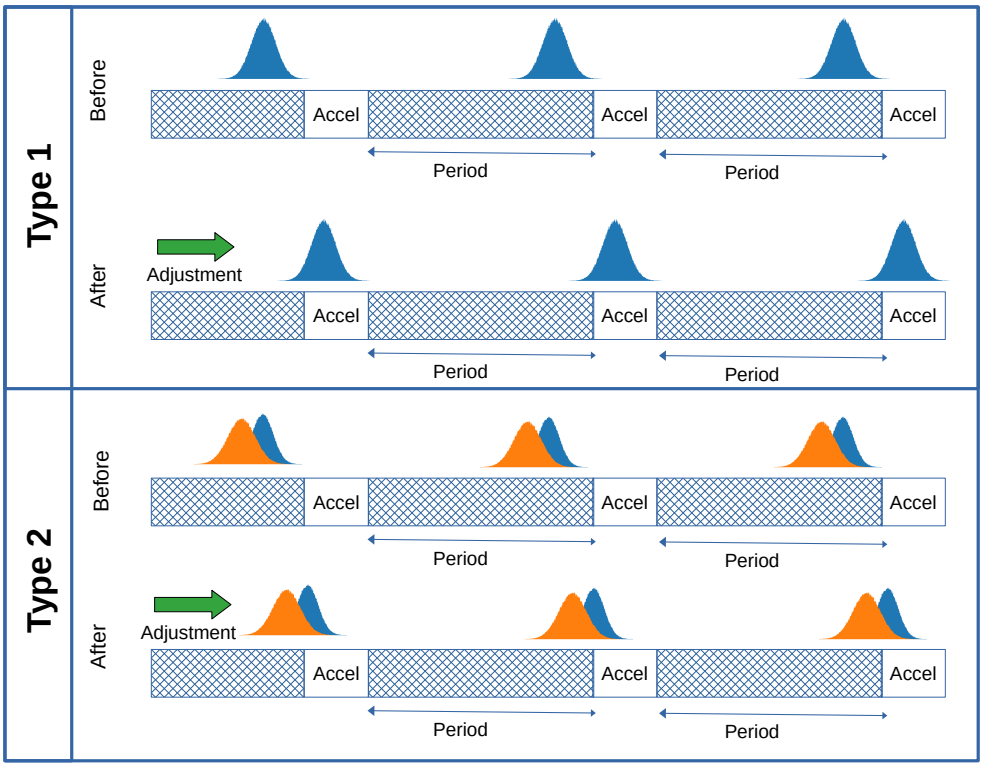

Figure 4.4: Adjusting timing based on the type of system

between the measurement in these application might lead to performance degradation or even system failure.

4.6 Computing the staggering time

In this section, we provide the statistical analysis used to find the Target arrival time. Figure 4.4 shows the proposed method based on the system type. For each system type(discussed in the previous section), At first, the mathematical model for the task wait time will be provided. Afterwards, we calculate the expected value of the task wait time, and then we minimize it by finding the Target arrival time.

4.6.1 Type 1: Single-agent sensing staggering

To minimize the wait time, just minimizing the absolute gap between arrival time and the start time of accelerator (T_{slot}) is not enough because the tasks that arrive after $T_{slot} + D_{slot}$ has to wait for the next time that accelerator becomes available, therefore their wait time will be $Period - T_a$, not $|T_a - T_{slot}|$. The sensing time can be adjusted to minimize the task wait time. To this end, we want to minimize the expected value of the wait time (represented by f) given the arrival time distribution, $Period$, T_{slot}, D_{slot}, with regards to average arrival time μ, i.e.:

$$\underset{\mu}{\operatorname{argmin}}\, \mathbf{E}[f(x)] \tag{4.3}$$

where

$$\mathbf{E}[f(x)] = \begin{cases} \int_{-inf}^{0} -x\,dx & + \\ \int_{0}^{D_{slot}} 0\,dx & + \\ \int_{D_{slot}}^{+inf} (Period - x)\,dx & \end{cases} \tag{4.4}$$

If we represent the cumulative density function by $F(a)$, then we will have

$$\mathbf{E}[f(x)] = -\mu + Period(1 - F(D_{slot})) + \int_{0}^{D_{slot}} x\,dx \tag{4.5a}$$

$$< -\mu + Period(1 - F(D_{slot})) + D_{slot}F(D_{slot}) \tag{4.5b}$$

we approximated $\int_0^{D_{slot}} x\,dx$ with $D_{slot}F(D_{slot})$ to make the equation simple and easier to compute.

By setting the derivative of eq. 4.5b w.r.t. μ, to zero, we can find the extrema of the wait time. Here, we assume the x follows a normal distribution. Note that the proposed method can be applied on other distributions such as exponential distribution. By taking derivative from equation 4.5b with regard to μ, we will have:

$$\frac{\mathrm{d}\mathbf{E}[f(x)]}{\mathrm{d}\mu} = -1 + (Period - D_{slot})\frac{1}{\sigma}\phi(\frac{D_{slot}-\mu}{\sigma}) \tag{4.6}$$

where ϕ is the probability density function of standard normal distribution. Therefore, equation 4.5b will be minimized/maximized for:

$$\phi(\frac{D_{slot}-\mu}{\sigma}) = \frac{\sigma}{Period - D_{slot}} \tag{4.7}$$

The answer to the above equation results in two values for μ, so we examine both values to choose the μ that minimize eq. 4.5. We enforce the μ by staggering the sensing time at the end device. However, since the distribution of the arrival time changes over time, we will need a method to estimate the arrival time, and stagger the sensing time based on it. we will propose the staggering method in the next section.

4.6.2 Type 2: Synchronized multi-agent sensing staggering

For this type, we assume all the end devices has to sample data at the same time. One of the applications that has this property is UKF based Cooperative Localization. If we run UKF before having all the data, when the missing data arrives, we have to restore the UKF to the previous state and compute all the steps again. Therefore, data from all devices taken

at the same time step, is required for computing the UKF. Since the absolute sensing time of the team can change, but the relative sensing time of team should remains intact, if the absolute sensing time needed to be adjusted to lower the wait time, it has to be changed for the entire team.

The equation for the D_{wait} is similar to the Type 1 application with a minor change. If x_i is the arrival time of ith device from a team of n, then $D_{wait}(x_1, x_2, ..., x_n)$ will be

$$\begin{cases} -y, & \text{if } y < 0 \\ 0, & \text{if } 0 =< y < D_{slot} \\ Period - y, & \text{if } y >= D_{slot} \end{cases} \tag{4.8}$$

The expected value of D_{wait} is

$$\mathbf{E}[f(y)] = -\mu + Period(1 - F_y(D_{slot})) + \int_0^{D_{slot}} y\, dy \tag{4.9a}$$

$$< -\mu + Period(1 - F_y(D_{slot})) + D_{slot} * F_y(D_{slot}) \tag{4.9b}$$

where y is $max(x_1, x_2, ..., x_n)$. We assume that x_i are i.i.d, therefore, $p(y < Y)$ will be $p(x_1 < Y)p(x_2 < Y)...p(x_n < Y)$. In general, we can minimize this equation with regard to μ using numerical methods.

To make sure that the sensing time changes at the same time for the entire team, there should be enough gap between the time edge send a 'Wait' command and the time that the

sensing time on the device will be changed. This gap should be large enough to accommodate for the downlink network latency of the entire team and the end device preparation.

4.7 Staggering module

Since $D_{preprocessing}$ and $D_{network}$ distribution change over time, estimating the arrival time is challenging. To tackle this problem, we propose a module, called Staggering module, that staggers the sensing time on the end device, so the T_a gets as close as possible to Target arrival time (T_a^{target}). The proposed module is a feedback loop that for each device, during runtime, monitors the task arrival times, and guides the end device to adjust sensing time, so the arrival times of the incoming future tasks get close to the Target arrival time. The Edge estimate the difference between actual arrival time and Target arrival time, and then the edge will send a 'Wait command' to the end-node to adjust the sensing time, if it is necessary. Note that the 'Wait command' only contains an integer number, which can be easily fit in one network packet. This method does not need a synchronized internal clock, nor explicitly measuring the network delay. Hence, it reduces the development time, eases the system deployment, and can easily detect and adapt to the changes in environment without requiring complex estimation methods. The core of this algorithm is arrival time estimation. The proposed estimation method should be tolerant to the network jitter and random delay changes. For this, we proposed two methods: 1) Sliding Window Averaging based method, and 2) Kalman filter based method.

4.7.1 Sliding Window Averaging based Staggering module

Figure 4.5 shows the proposed sliding window averaging based method. For each end device *device*, the edge calculates the average (m) of the difference between the arrival time of the

Algorithm 8: Sync with sliding window average

input : T_a^{target}, T_a

$Cnt = Cnt + 1$;
$m = SlidingWinAvaraging(T_a - T_a^{target}, WinSize)$
if $|m| > ThreshMeanTime$ *and* $Cnt > ThreshCnt$ **then**
    $SendCommand(device, 'Wait', m)$
    $ResetSlidingWinMean(device)$
    $Cnt = 0$
end

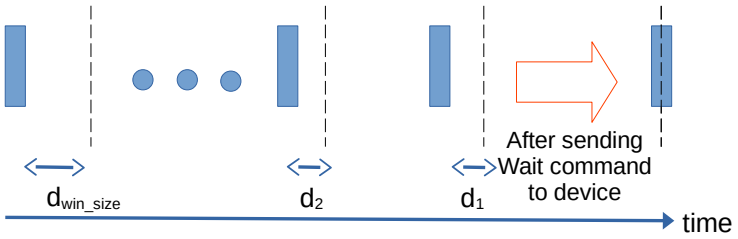

Figure 4.5: time staggering for an end device: The rectangles are the arrived tasks at the edge during runtime, the dotted lines are the Target arrival time, and d_i is their gap

data, and the Target arrival time. Then, the edge sends a command to the end device to adjust the sampling time by m. Calculating the average is incremental and lightweight with time complexity of $O(1)$ for a streaming data.

Algorithm 8 shows the details of the proposed method. The input to the algorithm is the target (T_a^{target}) and actual arrival time (T_a). It calculates the Sliding Window average (m) of the difference between T_a and T_a^{target}. We used the Sliding Window average because 1- using a single differences as the adjustment delay will make the system sensitive to network jitter, hence, averaging the differences is necessary. 2- the delays from long before may not characterize the current status of the system, therefore, we take average of the differences over a limited window size. Afterwards, if the m is above a threshold ($ThreshMeanTime$), and also if the number of tasks after sending the previous 'Wait command' is more than a threshold number ($ThreshCnt$), the edge will send a 'Wait command' to end device to adjust its sampling time by m. The former condition will prevent the edge to send commands for small differences, and the later condition will allow the end device to fix its timing before edge

send a new command, furthermore, it limits the communication overhead. After sending the command, we reset the array that we save the arrival times in it and also Cnt.

4.7.2 Kalman Filter based Staggering module

SWA is simple, however, it is knows for being sensitive to the noise. Therefore, using SWA for estimation will make the edge to send many commands to the end device. Kalman filter is a popular Bayesian method for estimating the state of the system using the system and measurement model. This method is more accurate than SWA, because based on the measurement and process noise, dynamically changes the contribution of the measurements in estimating the state of the system. Kalman filter consists of two steps of prediction and update. Equation 4.10 shows the prediction and update step. The input for each step k+1 is x_k, P_k, u_k, z_k, where u_k is the control signal, z_k is the measurement, and x_k and P_k are the estimated x and P from the previous step k. The R and Q are the measurement and process covariance matrices, which depends on the system characteristics. In our system, they can be adaptive and change over time, therefore, we used adaptive Kalman filter in [13].

$$x_{k+1} = A(x)_k + Bu_k \tag{4.10a}$$

$$P_{k+1} = A(P)_k A^T + Q \tag{4.10b}$$

$$K = P_k H^T (HP_k H^T + R_k)^{-1} \tag{4.10c}$$

$$z = (z_k - Hx_k) \tag{4.10d}$$

$$x_k^- = x_k + kz \tag{4.10e}$$

$$P_k^- = P_k - KHP_k \tag{4.10f}$$

where R_k can be computed with the following equation:

$$v_k = (z_k - Hx_k^-) \tag{4.11a}$$

$$C_v = \frac{1}{n} \sum_{i=k-n+1}^{k} (v_{k-i} v_{k-i}^T) \tag{4.11b}$$

$$R_k = C_v + HP_kH^T \tag{4.11c}$$

where n is the size of the moving window for the measurement. u_k is the adjustment delay that the server sends to the end device. The output of adaptive Kalman filter is the system state (here the estimated arrival time) and the covariance matrix, which indicates the uncertainty about the estimated state.

Algorithm 9 shows the Kalman based Staggering module. Similar to the sliding window averaging method, we have the delaying mechanism, using *cnt*, to allow the end device adjust its timing before sending a new 'Wait' command. In normal distribution, the 68%–95%–99.7% of the data fall within the 1,2, and 3 σ interval of the μ. To reduce the number of command that edge sends to the end device, we send the command only if it is within the $ConfidenceLvl * \sigma$ of the $ThresholdTime$. After sending the command, we run the Kalman again with u of x reflect its effect on the system state. We study the effect of the $ConfidenceLvl$ on the number of commands and the task wait time in the next section.

Algorithm 9: Sync with Kalman filter

input : T_a, T_a^{target}

1 $Cnt = Cnt + 1$;
2 $x, \sigma = Kalman(T_a - T_a^{target}, 0)$
3 **if** $|x| - \sigma * ConfidenceLvl > ThreshTime$ *and* $Cnt > ThreshCnt$ **then**
4 $\quad SendCommand(device, 'Wait', x)$
5 $\quad x, \sigma = Kalman(0, x)$
6 $\quad Cnt = 0$
7 **end**

For the Type 2 systems, the edge, obtains all the arrival times from the threads that are

receiving tasks from the end devices. Afterward, it computes the maximum arrival time and feed it to Algorithm 9 as T_a. Then, it sends the command to the entire team, so they change their sensing time at the same time.

4.8 Evaluation

In this section, we evaluate the effect of various parameters on the D_{wait}. Then we provide a detailed experimental results on the a set of networked embedded devices.

4.8.1 The effect of Target arrival time on the minimum D_{wait}

Figure 4.6 shows how expected value of D_{wait} changes with respect to Target arrival time (μ a.k.a T_a^{target}) for a couple of Type 1 applications with different configurations. We examined the combination of two *Period*s (100, and 200 ms), and the arrival time with variance (σ) of 10 and 15, all with D_{slot} of 10ms. D_{wait} for all the combinations are very close for the $\mu < -20$. The reason is that the data arrival time is shifted early enough such that almost all the frames arrive before the T_{slot}, here is 0. For this interval, as the μ moves toward -20, the D_{wait} reduces almost linearly with the slope of -1 as the average arrival time get closer to the T_{slot}. After that, depends on the σ, D_{wait} reaches to its global minimum. For a lower σ, this global minimum is lower, and the corresponding μ is closer to 0, as well. Then, the expected value of D_{wait} of increases until it reaches the global maximum (around $\mu = 20$), because more number of data packets will arrive after the D_{slot}, that have to wait for the next corresponding available slot. In this time interval, the gap between the applications with the period of 100 and 200 ms increases since the arrived frames has to wait longer to get processed the next time that the accelerator becomes available. As the μ get close to 40, the gap between combinations with the same *Period* decreases since majority of frames

arrive after D_{slot}, and they have to wait for the next corresponding slot.

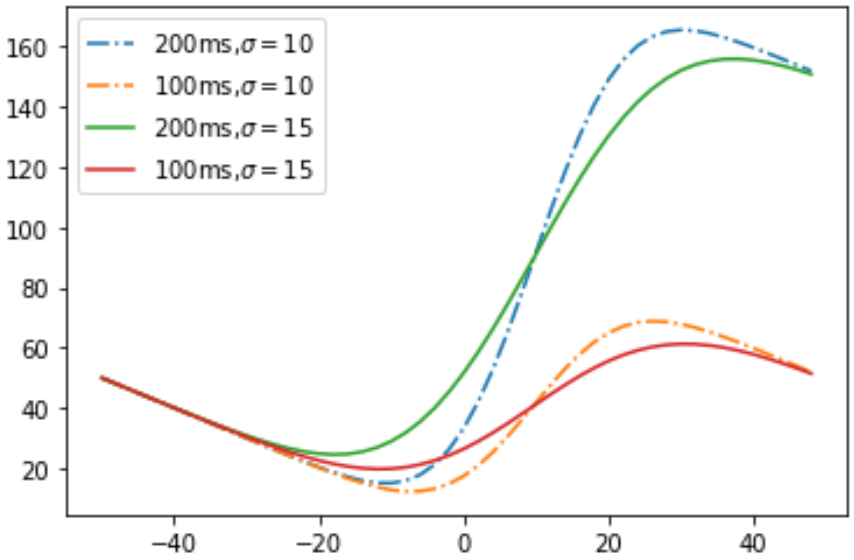

Figure 4.6: effect of Target arrival time (μ) on the D_{wait} for various frequencies with D_{slot} of 10ms

4.8.2 The effect of D_{slot} on the minimum D_{wait}

We examined the effect of D_{slot} on the minimum expected value of D_{wait} for applications with Type 1 and 2. We set the *Period* to 100 and 500 ms, and assume all the arrival times follows a normal distribution with σ of 10. For the Type 2 applications, we considered a team of 10 end device which has to capture the data at the same time. For each application, we find the minimum μ for all the D_{slot}s. In general, the minimum D_{slot} for the same *Period* and the same σ is lower for the Type 1 application, because of the the probability distribution of maximum arrival time of the Type 2 application arrival times, i.e. $max(x_1, x_2, ..., x_n)$. For the same application type, the one with lower *Period* has lower D_{wait} since the penalty of the data not arriving on time is lower. For example, for application type 1 and D_{slot} of 0, the D_{wait} is around 22ms for the application with period of 100, while it is 28 for the 500. As the D_{slot}, the minimum expected value of D_{wait} decreases, and it gets close to 0 for all applications around D_{slot} of 50. The reason is that with higher D_{slot}, more frames will fall

into $[T_{slot}, T_{slot} + D_{slot}]$ interval where frames do not need to wait to be executed on the edge.

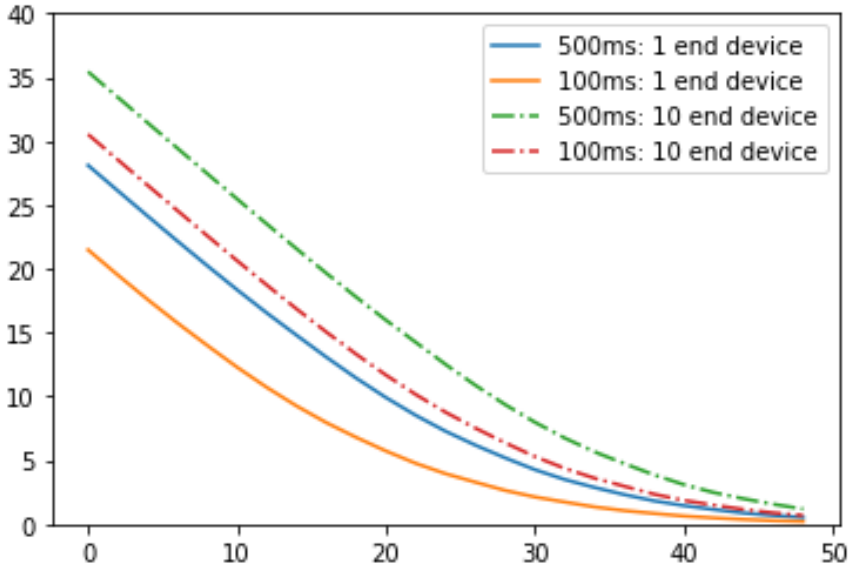

Figure 4.7: effect of D_{slot} on the minimum D_{wait} for various frequencies

4.9 On-device Evaluation

In this section, we evaluate the proposed method on multiple networked embedded devices that are connected to an edge.

4.9.1 Platform Setup

For the experiments, we used a Xilinx Ultrascale+ MPSoC ZCU104 board which has ARM Cortex-A53 CPUs and FPGA Fabrics, as the Edge and multiple Raspberry Pi 4B as the End devices, all running Linux kernel v.4.19. The End devices (Edge) are connected to a wireless access point, a TP-LINK Archer A7, through a 5GHz WiFi (Ethernet), and communicate using TCP/IP socket in an office environment. For each experiment, the end devices, starting at a random time, periodically sends requests for acceleration to the edge for 5 minutes. We measured and reported the time interval between the frame arrival at the edge and the start

time of the computation resource (wait time).

We used Inland REV 1.3 camera to capture pictures with resolution of 640x460 and 460x320 and compressed it using opencv with jpg format, and sent it to the edge. To evaluate the performance of our method, in addition to the natural noise of the system, during runtime, we used two sets of noise level: $(0 < \mu < 70, 0 < \sigma < 30)$ and $(0 < \mu < 70, 0 < \sigma < 10)$ for the devices with FPS of [2,3,5] and [8,10] respectively.

The $ThreshCnt$ is set to $2 * FPS$, therefore, the Edge device will send the 'Wait command', if necessary, at least 2 seconds after the last one, hence the command will have enough time to affect the end device sensing time. The computation overhead of Staggering module is negligible and the edge CPU utilization during runtime was under 3%. Note that we compute equation 4.5 once for integer σs on the edge, and cache it to prevent recalculation.

Time slotted edge: For the evaluation purpose, we consider an edge which works in a time slotted mode. The time slots are fixed and each time slot is assigned to an accelerator. In addition, the time interval between the start time of the slots assigned to each application is fixed and is equal to the application period. Note that our proposed method can be extended to support systems without these constraints as well. Each accelerator time slot includes fetching, execution, and idle time(D_{slot}). The edge can process the frame only when its corresponding accelerator is loaded. For simplicity, we assume that the execution time is negligible. For example, [116] reported runtime of 495 microsecond for running UKF on an FPGA. In addition, when requests processed in batch, the overall latency is lower than processing them individually. For example, [92] has measured 1.54ms for processing one request to a MNIST 4 layer neural network on FPGA, while processing two requests only takes 1.7 ms. Therefore, we assume that the slot duration is long enough to process multiple requests in a batch mode with negligible overhead compared to individual execution. Therefore, we assume the wait time due to queuing is negligible (this happens when one request missed its time slot, and has to be processed with the next request that has arrived

on time). This is a reasonable assumption for the applications with short execution time. For a given set of applications and FPS, we generate a fixed accelerator schedule, that the time interval between the time slots allocated for each application i is the same as the application period, i.e. FPS_i. Note that even if the time interval of all the slots are not equal, our proposed method can be used with simple modifications. We assume the 50 ms time slot for each accelerator, which includes fetching, execution, and idle time.

4.9.2 KF based method in action

Figure 4.8 shows the KF based method for an application with FPS of 10, confidence coefficient of 3, and D_{slot} of 0 for around 2 minutes. The top figure represents the estimated arrival time and estimation confidence level. For easier representation, we changes the equation in Algorithm 9 from $|x| - coef * \sigma > threshold$ to $x > coef * \sigma + threshold$ and $x < -(coef * \sigma + threshold)$ for $x > 0$ and $x < 0$ respectively. In this figure, whenever the x exceeds the confidence level line, the edge sends a command to the end device. The confidence interval decreases when the noise variance reduces, which means the estimated arrival time is more accurate and can be used for sending a more effective command. In the middle figure, the actual arrival time, Target arrival time, and the Commands has shown. The vertical solid green lines represent (tagged with a value in ms) the command, which negative and positive tagged values means the end node has to shift the sensing time to an earlier or later time. In this figure, all the values are negative. As for this benchmark, the D_{slot} is 0, any frame with the arrival time above the zero misses the corresponding time slot and has to wait for the next time the accelerator become available. The added noise transition is represented by dashed vertical lines and tagged with its mean and variance. The Target arrival time fluctuates over time, however, the average Target arrival time is closer to zero for the lower noise level, which brings the actual arrival times closer to zero, hence, reduces the wait time. The bottom figure shows the wait time. Note that after

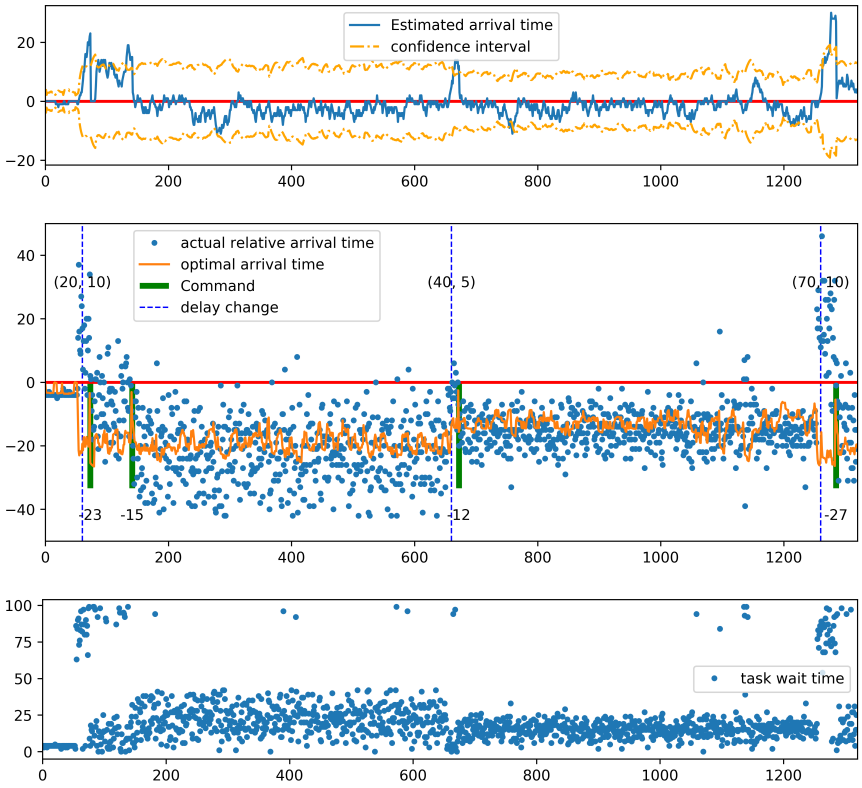

Figure 4.8: KF based method measurments for FPS of 10: (top) estimated arrival time and confidence level compared to Target arrival time (middle) Target arrival time, Commands, and actual arrival times, (bottom) frame wait time

each noise level transition, there are many frames that missed their corresponding time slot, therefore, their wait time is around 100ms. Figure 4.9 show the frame wait time for the same benchmark when the staggering module is disabled. As it is shown, many of frames has received after the T_{slot} and has to wait for around 100 ms to be processed.

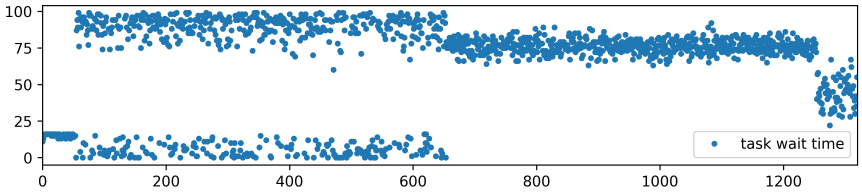

Figure 4.9: frame wait time with One Time Staggered (OST) method

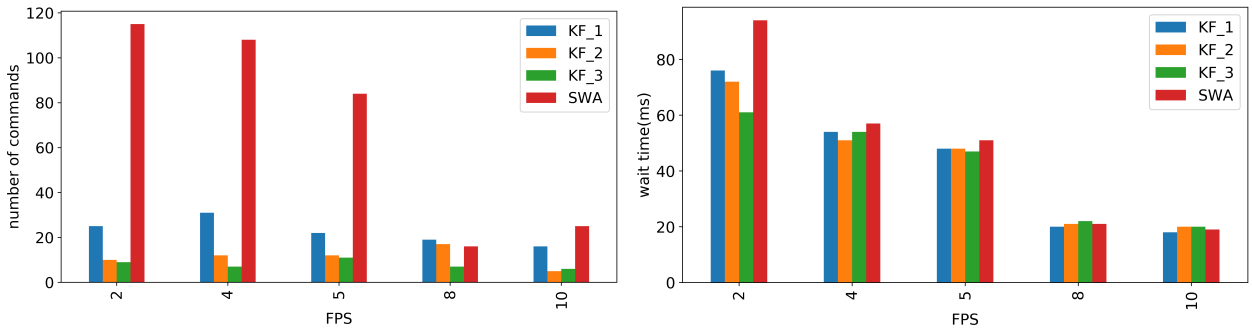

Figure 4.10: (left)the number of commands and (right) the average wait time for SWA and KF for confidence level coefficient of 1,2, and 3 for various FPS

4.9.3 The number of commands for KF and SWA

Figure 4.10 compares the KF and SWA method in terms of the number of commands and the average wait time. In addition, it shows the effect of confidence level coefficient parameter in KF method. As it is shown, the number of commands decreases for the higher coefficient without negatively affecting the overall wait time. The number of commands for KF based method is significantly (up to 12x) lower than SWA method, which shows the effectiveness of KF.

For KF based methods, the coefficient of 1 resulted in the highest number of commands for for all the benchmarks. However, for the coefficient of 2 and 3 the number of commands are close to each other. The number of commands is lower for the devices with lower added noise level, i.e. 8 and 10. The average wait time for both set of noise level is always decreasing with regards to the FPS because of the higher penalty for missing a time slot (the less FPS, the higher penalty).

4.9.4 Proposed method vs baseline method

Table 4.2 compares the KF and SWA based Staggering module for various FPSs.In this table, the KF confidence level coefficient is 3. We set the D_{slot} to 0. The difference between average

Table 4.2: comparison of KF and SWA methods

FPS	wait time(ms)				missed slot %		
	KF-3	SWA	OTS	PKN	KF-3	SWA	PKN
2	61	95	120(49%)	44	4.8	13.4	1.4
4	54	57	92(41%)	39	8.2	12.1	3.3
5	47	51	138(66%)	37	8.9	12.9	4.6
8	22	21	85(74%)	15	3.8	6.3	2.1
10	20	19	64(69%)	14	6.1	7.5	2.8

wait time of KF and SWA methods is negligible. The number of frames that has not arrived before the start of missed the allocated slot and has to wait for the next slot is lower for the KF based method, which means the KF performed better in detecting the arrival times. We have calculated the theoretical wait time for the added noise using equation 4.5, and we refer to is as Priory Known Noise (PKN) method. Although the difference between the PKN wait time and KF based is up to 17ms in some benchmarks, the ratio of the difference compared to the *Period* is less than 6%. The difference is because the PKN wait time is calculated only based on the added noise and does not consider the noise of sampling/preprocessing/WiFi. In addition, the edge is processing a streaming data and does not have the whole data. Therefore, when the noise distribution changes, it can be detected by the proposed method after some time, during which the frames arrives at a non optimal time (refer to Figure 4.8). To better illustrate the effect of the proposed method, we measured the wait time when the Staggering module module is active for the first 30 seconds of the experiment, and after that, it will be deactivated. In Table 4.2, we refer it as the One Time Staggered (OTS). Using KF-3, the wait time has improved from 41% up to 74% compared to OTS.

4.9.5 Type 2 experimental result

As mentioned earlier, in these systems, the sensing time is synchronized among the entire team. The edge can process the data taken at a time step only when the data from the all members of the team arrives to the edge. For example, in UKF based CL, the agents send

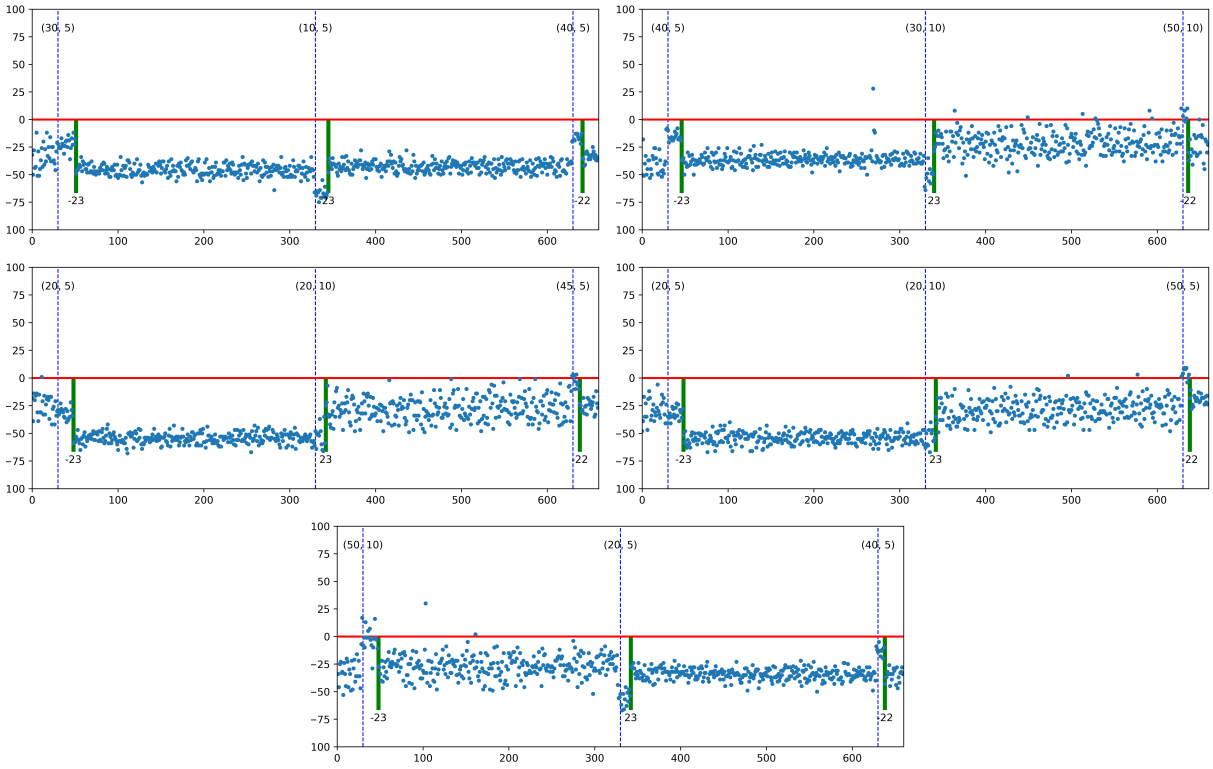

Figure 4.11: KF based method measurements for FPS of 5 for 5 agents with synchronized sensing time: Commands, and actual arrival times

their control signals and measurements to the edge to calculate their location at each time step. The other example of this type of system is the bridge health monitoring systems that all the samples has to be taken at the same time. To evaluate the effect of our methodology, we synchronize the time of all the devices, and then start sending the frames to the edge. Then we measure the arrival times, and compute the wait time base on the frame that arrived the earliest. Our baseline is the OTS method, where we only run the staggering method for the first 30 seconds of the experience.

Figure 4.11 shows the arrival times and Wait commands for a benchmark with 5 agents, each sending data to edge with the FPS of 5 for around 2 minutes. The dashed lines shows the added noise with the format of (μ, σ). The staggering time is according to the agent that its frames arrives later than others. Hence, the latest arrived frame usually arrives before the T_{slot}. Similar to the Type 1 application, during the noise level transition, some frames arrives

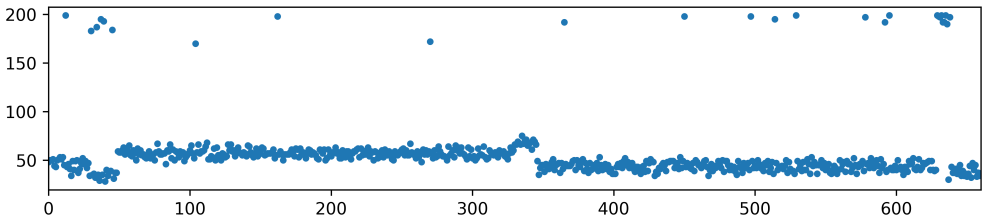

Figure 4.12: KF based method measurements for FPS of 5 for 5 agents with synchronized sensing time: wait time for the earliest arrived frame corresponding to the experience in Figure 4.11

Table 4.3: comparison of KF and OTS for a team of 5 agents with synchronized sensing time during 5 minutes experience

FPS	wait time		number of commands
	KF-3	OTS	
5*2	129	460(71%)	20
5*4	110	227 (51%)	25
5*5	53	175(69%)	40
5*8	47	109 (56%)	20
5*10	36	81(55%)	25

after the T_{slot}. The overall wait time for the earliest arrived frame is shown in Figure 4.12. Table 4.3 shows the wait time and number of commands for Kalman filter based staggering method and the OTS for 5 devices with various FPS. The Kalman filter based method has improved the wait time by 71% compared to OTS.

4.10 Conclusion

Since the computation capacity of embedded nodes is limited, offloading computation to the edge and cloud is data arrival time can affect the responsiveness of the system as if the data arrives too early, or late that it misses the allocated accelerator, it has wait on the edge to be processed. Adjusting the arrival time is challenging due to the on device processing and network delay variation. In this chapter, for iterative applications, we proposed a method to find the Target arrival time based on the arrival time variation using statistical methods to minimize the expected value of the frame wait time on the edge. Since the arrival time

is noisy and its distribution might change over time, it is essential to denoise the streaming arrival times, and estimate the arrival time. Therefore, we provided a Sliding Window Averaging and a Kalman filter based method to estimate the arrival time, and then, send the gap between the estimated arrival time and Target arrival time to the end node as the staggering time. The proposed method is online, and only needs the arrival time to function. In addition, it has a very low engineering cost as it does not to profile and estimate the delay of each of the components. At the end, we extended our proposed method to support the synchronized networked embedded systems, where end nodes has to capture data at the same time. Out experimental results on multiple networked Raspberry Pi shows the effectiveness of our method in reducing the frame wait time on the edge.

Chapter 5

Conclusion and future works

The networked embedded devices has limited computation and communication resources. An application can be distributed among the team members or offload to the edge, if it is accessible. Distributing an application among the networked embedded devices is challenging since it imposes communication overhead. On the other hand, the data is generated locally on the devices. In this dissertation, we proposed a methodology to distribute application among the networked embedded devices. We used computation replication technique, which is widely used in chip design, to reduce the communication overhead. Since computation replication increases the CPU utilization on the agents, we proposed a selective computation replication to minimize the replication overhead, while keeping the end-to-end delay of the application under a user defined threshold.

When agents have access to a server, such as an edge, computation offloading is a promising method to reduce the agent power consumption and to speedup the data processing. Data freshness, which can be measured by Age of Information metric, is crucial for many applications such as remote surgery. In a time slotted edge, if tasks arrives too early, or later than the allocated time slot, they have to wait for the next dedicated time slot. In this dissertation,

we proposed a method to reduce the wait time by adjusting the sampling time. The proposed method, does not need complicated communication and computation delay estimator. We use a feedback loop to adjust the end device sampling time to minimize the task wait time on the edge.

The computation-communication co-optimization is essential for networked embedded systems. The research in this field, which is partially covered in this dissertation, can be continue in the following directions:

5.1 Decentralization

In this dissertation, We proposed the algorithmic method for computation replication to reduce the communication overhead of distributing the application. Deploying this method in compilers can be very beneficial to reduce the design time as they have access to all the basic blocks information. To this end, computation and communication latencies has to be measured or estimated, and provided to the compiler.

5.2 Centralized method (Edge)

5.2.1 Allocating the idle time to the D_{slot}

In allocating time for D_{slot}, it should be considered that it is the edge idle time that is given to each slot to tolerate the task arrival time variation better. Longer D_{slot} will decrease the D_{wait}, and hence the AoI of the application, however, it decrease the edge utilization. Figure 5.1 show the amount of D_{wait} for various D_{slot} during one second compared to the D_{wait} when the D_{slot} is 0. Increasing D_{slot} improve the D_{wait}, however the gain decreases. For the

same D_{slot}, the gain is higher for the applications with longer period. However, for the it translates to a higher idle time as well. In addition, the gain for higher σ is more for the same D_{slot}. Figure 5.2 shows the ratio of the gain to the added idle time due to D_{slot} for the same setting. Among these combinations, the gain ratio is higher for combination with the higher $Period$ and σ.

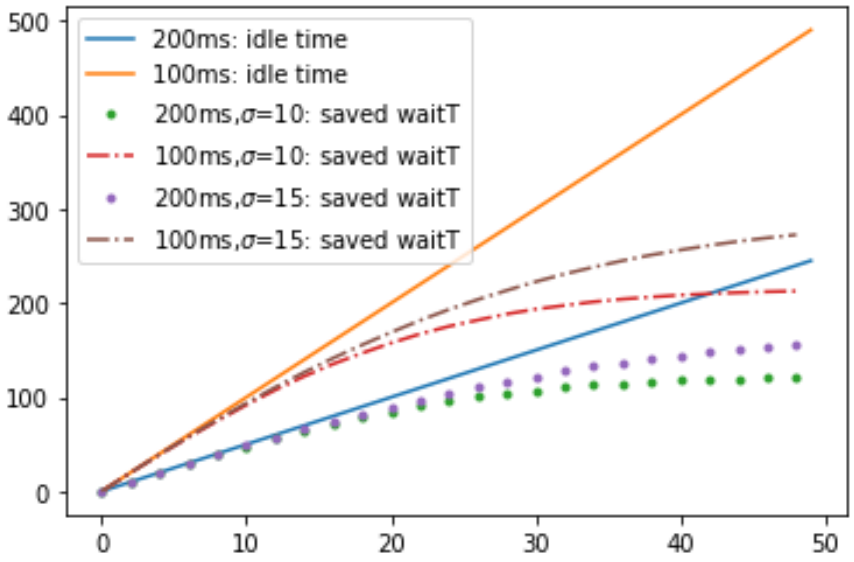

Figure 5.1: The effect of D_{slot} on the saved D_{wait} and idle time for various frequencies and variations

In this section, we study the distribution of the edge idle time between applications to minimize the overall average D_{wait}. lest assume we have n applications with period of $P_1, P_2, .., P_n$ (i.e. with frequency of $f_1, f_2, .., f_n$), and arrival time of with variation of $\sigma_1, \sigma_2, ..., \sigma_n$. We want to allocate the idle time of edge, $idleTime$, to the $D_{slot1}, D_{slot2}, ..., D_{slotn}$, where $\sum_{i=1}^{n} f_i D_{sloti} \leq idleTime$, in order to minimize the overall D_{wait} over a 1 second time interval, i.e.:

$$\underset{D_{slot1}, D_{slot2}, ..., D_{slotn}}{\operatorname{argmin}} \sum_{i=1}^{n} \mathbf{E}[D_{wait,i}(D_{sloti}, p_i, x; \mu_i, \sigma_i)] * f_i \tag{5.1}$$

where $\sum_{i=1}^{n} f_i D_{sloti} \leq idleTime$, and μ_i is a function of D_{sloti}, p_i, σ_i which is discussed

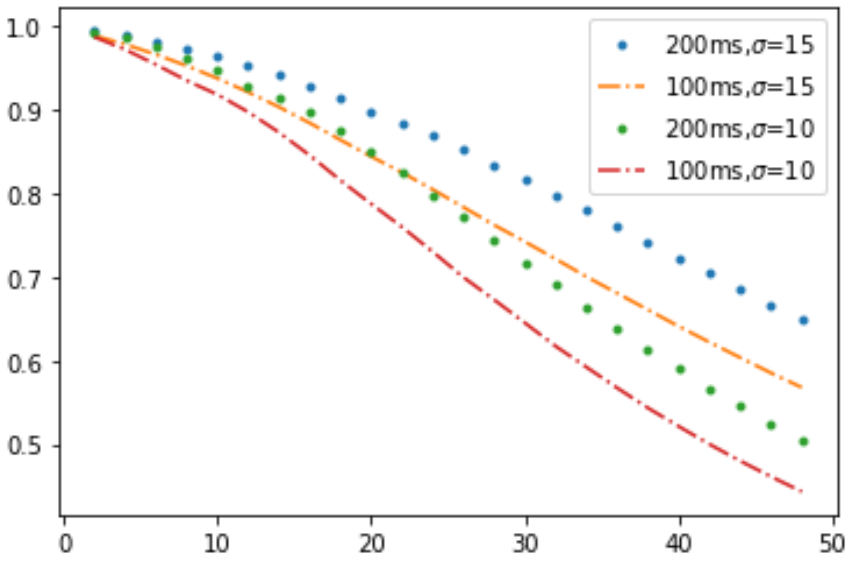

Figure 5.2: The effect of D_{slot} on the ratio of saved D_{wait} to idle time for various frequencies and variations

earlier. The is a knapsack problem and can be solved using dynamic programming. Figure 5.3 shows the total D_{wait} for various application combinations when the overall $idleTime$ varies between 0 and 500ms. In the figure, each tuple is $(frequency, \sigma)$. The overall D_{wait} is the lowest for $(5, 5), (5, 5)$, which has the lowest frequency and variation, and it decreases sharply before the $idleTime$ of 100, and eventually it gets close to 0. If we don't distribute the $idleTime$ between the slots, the D_{wait} will be the same as when the $idleTime$ is 0. For all the combinations, the D_{wait} decrease when the $idleTime$ increases.

5.2.2 Applications without adaptive sampling time

There are some applications that the parameters has to be sampled at certain absolute time and it can not be changed. Since in this type of applications, the sampling time cannot be change, the solution to reduce the D_{wait} is to shift of T_{slot}.

This problem is exactly the same as optimization problem for Type 1 systems. Then the adjustment value can be computed by optimizing equation 4.5b. Adjusting the T_{slot} might

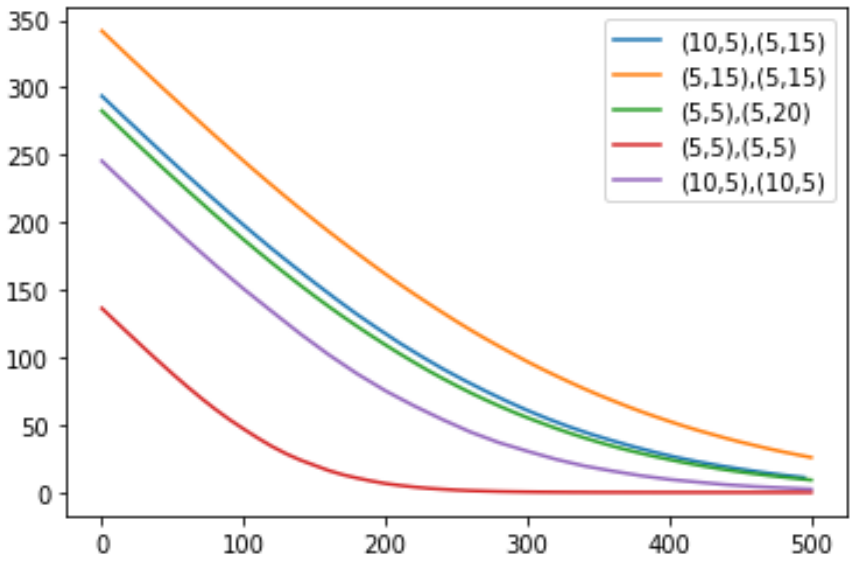

Figure 5.3: The effect of $idleTime$ on the total expected value of D_{wait} over 1 second various frequencies and variations

requires change in the schedule of other slots, which might not be possible if there are other Type 3 applications are running on the Edge, which changing their T_{slot} might degrade their performance. In addition, if other other types of applications are running on edge, the sampling time of their end devices has to change as well.

Bibliography

[1] Esp8266. https://en.wikipedia.org/wiki/ESP8266.

[2] Gnss. https://www.nist.gov/pml/time-and-frequency-division/popular-links/time-frequency-z/time-and-frequency-z-g.

[3] Gps. https://www.gps.gov/systems/gps/performance/accuracy/.

[4] Internet of things (iot). https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[5] Jetson. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/.

[6] perf: Linux profiling with performance counters.https://perf.wiki.kernel.org/.

[7] raspi. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[8] D. Aghamirzaie, S. A. Razavi, M. S. Zamani, and M. Nabiyouni. Reduction of process variation effect on fpgas using multiple configurations. In *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, pages 85–90. IEEE, 2010.

[9] A. Akca and M. Ö. Efe. Multiple model kalman and particle filters and applications: A survey. *IFAC-PapersOnLine*, 52(3):73–78, 2019.

[10] L. Ale, N. Zhang, X. Fang, X. Chen, S. Wu, and L. Li. Delay-aware and energy-efficient computation offloading in mobile-edge computing using deep reinforcement learning. *IEEE Transactions on Cognitive Communications and Networking*, 7(3):881–892, 2021.

[11] A. AlHammadi, A. AlZaabi, B. AlMarzooqi, S. AlNeyadi, Z. AlHashmi, and M. Shatnawi. Survey of iot-based smart home approaches. In *2019 Advances in Science and Engineering Technology International Conferences (ASET)*, pages 1–6. IEEE, 2019.

[12] B. Allotta, L. Chisci, R. Costanzi, F. Fanelli, C. Fantacci, E. Meli, A. Ridolfi, A. Caiti, F. Di Corato, and D. Fenucci. A comparison between ekf-based and ukf-based navigation algorithms for auvs localization. In *OCEANS 2015 - Genova*, pages 1–5, 2015.

[13] A. Almagbile, J. Wang, and W. Ding. Evaluating the performances of adaptive kalman filter methods in gps/ins integration. *Journal of Global Positioning Systems*, 9(1):33–40, 2010.

[14] D. Amiri et al. Edge-assisted sensor control in healthcare iot. In *Globecom SAC EH*. IEEE, 2018.

[15] K. Antevski, M. Groshev, G. Baldoni, and C. J. Bernardos. Dlt federation for edge robotics. In *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 71–76. IEEE, 2020.

[16] K. Antevski, M. Groshev, L. Cominardi, C. Bernardos, A. Mourad, and R. Gazda. Enhancing edge robotics through the use of context information. In *Proceedings of the Workshop on Experimentation and Measurements in 5G*, pages 7–12, 2018.

[17] I. Azimi et al. Hich: Hierarchical fog-assisted computing architecture for healthcare iot. *ACM Transactions on Embedded Computing Systems (TECS)*, 2017.

[18] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Communication-optimal parallel and sequential cholesky decomposition. *SIAM Journal on Scientific Computing*, 32(6):3495–3523, 2010.

[19] M. Bastopcu, B. Buyukates, and S. Ulukus. Selective encoding policies for maximizing information freshness. *IEEE Transactions on Communications*, 69(9):5714–5726, 2021.

[20] A. J. Ben Ali, Z. S. Hashemifar, and K. Dantu. Edge-slam: edge-assisted visual simultaneous localization and mapping. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 325–337, 2020.

[21] A. Benoit et al. A survey of pipelined workflow scheduling: Models and algorithms. *ACM Computing Surveys*, 2013.

[22] S. Biookaghazadeh, M. Zhao, and F. Ren. Are {FPGAs} suitable for edge computing? In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[23] N. Bredeche and N. Fontbonne. Social learning in swarm robotics. *Philosophical Transactions of the Royal Society B*, 377(1843):20200309, 2022.

[24] D. R. Brown and A. G. Klein. Precise timestamp-free network synchronization. In *2013 47th Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2013.

[25] D. Callegaro and M. Levorato. Optimal computation offloading in Edge-Assisted UAV systems. In *Globecom SAC TI*, 2018.

[26] Z. Chang, S. Liu, X. Xiong, Z. Cai, and G. Tu. A survey of recent advances in edge-computing-powered artificial intelligence of things. *IEEE Internet of Things Journal*, 2021.

[27] T. Cieslewski, S. Choudhary, and D. Scaramuzza. Data-efficient decentralized visual slam. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 2466–2473. IEEE, 2018.

[28] I. Colbert, J. Daly, K. Kreutz-Delgado, and S. Das. A competitive edge: Can fpgas beat gpus at dcnn inference acceleration in resource-limited edge computing applications? *arXiv preprint arXiv:2102.00294*, 2021.

[29] L. Corneo, N. Mohan, A. Zavodovski, W. Wong, C. Rohner, P. Gunningberg, and J. Kangasharju. (how much) can edge computing change network latency? In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2021.

[30] S. Dey and A. Mukherjee. Robotic slam: a review from fog computing and mobile edge computing perspective. In *Adjunct Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services*, pages 153–158, 2016.

[31] V. Dinh and S. S. Kia. A server-client based distributed processing for an unscented kalman filter for cooperative localization. In *Multisensor Fusion and Integration for Intelligent Systems (MFI), 2015 IEEE International Conference on*, pages 43–48. IEEE, 2015.

[32] V. Dinh and S. S. Kia. A server-client based distributed processing for an unscented kalman filter for cooperative localization. In *2015 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 43–48. IEEE, 2015.

[33] J. Dong, F. Zheng, J. Lin, Z. Liu, F. Xiao, and G. Fan. Ec-ecc: Accelerating elliptic curve cryptography for edge computing on embedded gpu tx2. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(2):1–25, 2022.

[34] H. Ebrahimi, M. S. Zamani, and S. A. Razavi. A switch box architecture to mitigate bridging and short faults in sram-based fpgas. In *2010 IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 218–224. IEEE, 2010.

[35] A. Elsts, X. Fafoutis, S. Duquennoy, G. Oikonomou, R. Piechocki, and I. Craddock. Temperature-resilient time synchronization for the internet of things. *IEEE Transactions on Industrial Informatics*, 14(5):2241–2250, 2017.

[36] E. Faniadis and A. Amanatiadis. Deep learning inference at the edge for mobile and aerial robotics. In *2020 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 334–340. IEEE, 2020.

[37] D. Fox, W. Burgard, H. Kruppa, and S. Thrun. A probabilistic approach to collaborative multi-robot localization. *Autonomous robots*, 8(3):325–344, 2000.

[38] E. Gamal et al. Optimal replication for min-cut partitioning. In *Computer-Aided Design, IEEE/ACM International Conference on*, 1992.

[39] R. C. L. Gámez, P. Martí, M. Velasco, and J. M. Fuertes. Wireless network delay estimation for time-sensitive applications. *Autom Control Dept Tech. Univ Catalonia Catalonia Spain Tech Rep ESAII RR-06*, 12, 2006.

[40] A. Giannitrapani, N. Ceccarelli, F. Scortecci, and A. Garulli. Comparison of ekf and ukf for spacecraft localization via angle measurements. *IEEE Transactions on aerospace and electronic systems*, 47(1):75–84, 2011.

[41] S. Goel, A. Kealy, B. Lohani, and G. Retscher. A cooperative localization system for unmanned aerial vehicles: Prototype development and analysis. In *Proceedings of 10th International Symposium on Mobile Mapping Technology (MMT 2017)*, pages 6–8, 2017.

[42] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[43] R. Hadidi et al. Distributed perception by collaborative robots. *IEEE Robotics and Automation Letters*, 2018.

[44] T. Halsted, O. Shorinwa, J. Yu, and M. Schwager. A survey of distributed optimization methods for multi-robot systems. *arXiv preprint arXiv:2103.12840*, 2021.

[45] B. R. Hamilton, X. Ma, Q. Zhao, and J. Xu. Aces: Adaptive clock estimation and synchronization using kalman filtering. In *Proceedings of the 14th ACM international conference on Mobile computing and networking*, pages 152–162, 2008.

[46] A. HeydariGorji, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves, and P. H. Chou. Stannis: Low-power acceleration of dnn training using computational storage devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[47] A. HeydariGorji, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves, and P. H. Chou. In-storage processing of i/o intensive applications on computational storage drives. *arXiv preprint arXiv:2112.12415*, 2021.

[48] A. Howard, M. J. Mataric, and G. S. Sukhatme. Putting the'i'in'team': An ego-centric approach to cooperative localization. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 1, pages 868–874. IEEE, 2003.

[49] A. Howard, M. J. Matark, and G. S. Sukhatme. Localization for mobile robot teams using maximum likelihood estimation. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 1, pages 434–439. IEEE, 2002.

[50] P. Huang, L. Zeng, X. Chen, K. Luo, Z. Zhou, and S. Yu. Edge robotics: Edge-computing-accelerated multi-robot simultaneous localization and mapping. *IEEE Internet of Things Journal*, 2022.

[51] P. Huang, L. Zeng, K. Luo, J. Guo, Z. Zhou, and X. Chen. Colaslam: Real-time multi-robot collaborative laser slam via edge computing. In *2021 IEEE/CIC International Conference on Communications in China (ICCC)*, pages 242–247. IEEE, 2021.

[52] L. J. Hwang and A. El Gamal. Min-cut replication in partitioned networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1):96–106, 1995.

[53] A. Javani, M. Zorgui, and Z. Wang. Age of information in multiple sensing. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019.

[54] A. Javani, M. Zorgui, and Z. Wang. On the age of information in erasure channels with feedback. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pages 1–6, 2020.

[55] A. Javani, M. Zorgui, and Z. Wang. Age of information for multiple-source multiple-server networks. *arXiv preprint arXiv:2106.07247*, 2021.

[56] W. Jing, G. Xuetao, and Z. Yang. An adaptive encoding application sharing system based on remote display. In *2013 Third International Conference on Intelligent System Design and Engineering Applications*, pages 266–269. IEEE, 2013.

[57] J. Jo, S. Jeong, and P. Kang. Benchmarking gpu-accelerated edge devices. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 117–120. IEEE, 2020.

[58] S. J. Julier and J. K. Uhlmann. A new extension of the kalman filter to nonlinear systems. In *AeroSense: the 11th International Symposium on Aerospace/Defence Sensing, Simulation and Controls*, pages 182–193, 1997.

[59] P. Kang and S. Lim. A taste of scientific computing on the gpu-accelerated edge device. *IEEE Access*, 8:208337–208347, 2020.

[60] N. Karam, F. Chausse, R. Aufrere, and R. Chapuis. Localization of a group of communicating vehicles by state exchange. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 519–524. IEEE, 2006.

[61] S. Kaul, R. Yates, and M. Gruteser. Real-time status: How often should one update? In *2012 Proceedings IEEE INFOCOM*, pages 2731–2735. IEEE, 2012.

[62] S. S. Kia et al. Cooperative localization for mobile agents: A recursive decentralized algorithm based on kalman-filter decoupling. *IEEE Control Systems Magazine*, 2016.

[63] S. S. Kia, J. Hechtbauer, D. Gogokhiya, and S. Martinez. Server assisted distributed cooperative localization over unreliable communication links. *IEEE Transactions on Robotics*, 34(5):1392–1399, 2018.

[64] S. S. Kia, S. Rounds, and S. Martinez. Cooperative localization for mobile agents: a recursive decentralized algorithm based on kalman-filter decoupling. *IEEE Control Systems*, 36(2):86–101, 2016.

[65] Z. Kurt-Yavuz and S. Yavuz. A comparison of ekf, ukf, fastslam2. 0, and ukf-based fastslam algorithms. In *2012 IEEE 16th International Conference on Intelligent Engineering Systems (INES)*, pages 37–43. IEEE, 2012.

[66] P.-Y. Lajoie, B. Ramtoula, Y. Chang, L. Carlone, and G. Beltrame. Door-slam: Distributed, online, and outlier resilient slam for robotic teams. *IEEE Robotics and Automation Letters*, 5(2):1656–1663, 2020.

[67] C. Li, J. Lu, and W. Su. Ekf based distributed cooperative localization for a multirobot team. In *2016 14th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1–6. IEEE, 2016.

[68] M. Li, Z. Chang, Z. Zhong, and Y. Gao. Relative localization in multi-robot systems based on dead reckoning and uwb ranging. In *2020 IEEE 23rd International Conference on Information Fusion (FUSION)*, pages 1–7, 2020.

[69] Y. Li, Y. Wang, W. Yu, and X. Guan. Multiple autonomous underwater vehicle cooperative localization in anchor-free environments. *IEEE Journal of Oceanic Engineering*, 44(4):895–911, 2019.

[70] C.-S. Lin, C.-S. Lin, Y.-S. Lin, P.-A. Hsiung, and C. Shih. Multi-objective exploitation of pipeline parallelism using clustering, replication and duplication in embedded multi-core systems. *Journal of Systems Architecture*, 59(10):1083–1094, 2013.

[71] L. Lin, X. Liao, H. Jin, and P. Li. Computation offloading toward edge computing. *Proceedings of the IEEE*, 107(8):1584–1607, 2019.

[72] L. Liu, H. Li, and M. Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[73] R. Liu, C. Yuen, T.-N. Do, D. Jiao, X. Liu, and U.-X. Tan. Cooperative relative positioning of mobile users by fusing IMU inertial and UWB ranging information. In *2017 IEEE International Conference onRobotics and Automation (ICRA)*, pages 5623–5629. IEEE, 2017.

[74] T. Liu, Y. Zhang, Y. Zhu, W. Tong, and Y. Yang. Online computation offloading and resource scheduling in mobile-edge computing. *IEEE Internet of Things Journal*, 8(8):6649–6664, 2021.

[75] L. Luft, T. Schubert, S. I. Roumeliotis, and W. Burgard. Recursive decentralized collaborative localization for sparsely communicating robots. In *Robotics: Science and Systems*, 2016.

[76] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi. Resource scheduling in edge computing: A survey. *IEEE Communications Surveys & Tutorials*, 2021.

[77] W. Masood, J. F. Schmidt, G. Brandner, and C. Bettstetter. Disty: Dynamic stochastic time synchronization for wireless sensor networks. *IEEE Transactions on Industrial Informatics*, 13(3):1421–1429, 2016.

[78] S. H. Moallempour, S. A. Razavi, and M. S. Zamani. Tsv reduction in homogeneous 3d fpgas by logic resource and input pad replication. In *2011 IEEE International 3D Systems Integration Conference (3DIC), 2011 IEEE International*, pages 1–5. IEEE, 2011.

[79] R. C. Motta, K. M. de Oliveira, and G. H. Travassos. On challenges in engineering iot software systems. In *Proceedings of the XXXII Brazilian symposium on software engineering*, pages 42–51, 2018.

[80] E. K. Naeini, I. Azimi, A. M. Rahmani, P. Liljeberg, and N. Dutt. A real-time ppg quality assessment approach for healthcare internet-of-things. *Procedia Computer Science*, 151:551–558, 2019.

[81] P. Nascimento, B. Kimura, D. Guidoni, and L. Villas. An integrated dead reckoning with cooperative positioning solution to assist gps nlos using vehicular communications. *Sensors*, 18(9):2895, 2018.

[82] A. Neekabadi, S. Samavi, N. Karimi, E. Nasr-Esfahani, S. Razavi, and S. Shirani. Lossless compression of mammographic images by chronological sifting of prediction errors. In *2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 58–61. IEEE, 2007.

[83] A. Neekabadi, S. Samavi, S. Razavi, N. Karimi, and S. Shirani. Lossless microarray image compression using region based predictors. In *2007 IEEE International Conference on Image Processing*, volume 2, pages II–349. IEEE, 2007.

[84] C. Neff, M. Mendieta, S. Mohan, M. Baharani, S. Rogers, and H. Tabkhi. Revamp 2 t: real-time edge video analytics for multicamera privacy-aware pedestrian tracking. *IEEE Internet of Things Journal*, 7(4):2591–2602, 2019.

[85] E. D. Nerurkar and S. I. Roumeliotis. A communication-bandwidth-aware hybrid estimation framework for multi-robot cooperative localization. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1418–1425. IEEE, 2013.

[86] E. D. Nerurkar, S. I. Roumeliotis, and A. Martinelli. Distributed maximum a posteriori estimation for multi-robot cooperative localization. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 1402–1409. IEEE, 2009.

[87] R. Olaniyan and M. Maheswaran. Synchronous scheduling algorithms for edge coordinated internet of things. In *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10, 2018.

[88] J. Pan and J. McElhannon. Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal*, 5(1):439–449, 2017.

[89] S. Panzieri, F. Pascucci, and R. Setola. Multirobot localisation using interlaced extended kalman filter. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2816–2821. IEEE, 2006.

[90] B. Peng, G. Seco-Granados, E. Steinmetz, M. Fröhle, and H. Wymeersch. Decentralized scheduling for cooperative localization with deep reinforcement learning. *IEEE Transactions on Vehicular Technology*, 68(5):4295–4305, 2019.

[91] P. Poncela, E. Ruiz, and K. Miranda. Factor extraction using kalman filter and smoothing: This is not just another survey. *International Journal of Forecasting*, 37(4):1399–1425, 2021.

[92] T. Posewsky and D. Ziener. Throughput optimizations for fpga-based deep neural network inference. *Microprocessors and microsystems*, 60:151–161, 2018.

[93] G. Premsankar, M. Di Francesco, and T. Taleb. Edge computing for the internet of things: A case study. *IEEE Internet of Things Journal*, 5(2):1275–1284, 2018.

[94] A. Prorok and A. Martinoli. A reciprocal sampling algorithm for lightweight distributed multi-robot localization. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3241–3247. IEEE, 2011.

[95] S. Ranaweera and D. P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 445–450. IEEE, 2000.

[96] S. Ranaweera and D. P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *IPDPS*. IEEE, 2000.

[97] S. A. Razavi, E. Bozorgzadeh, and S. S. Kia. Communication-computation co-design of decentralized task chain in cps applications. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1082–1087. IEEE, 2019.

[98] S. A. Razavi, E. Bozorgzadeh, K. Kim, and S. S. Kia. Resource-aware decentralization of a ukf-based cooperative localization for networked mobile robots. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 296–303. IEEE, 2018.

[99] S. A. Razavi and M. Saheb Zamani. Improving bitstream compression by modifying fpga architecture. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 167–170, 2013.

[100] S. A. Razavi, H.-Y. Ting, T. Giyahchi, and E. Bozorgzadeh. On exploiting patterns for robust fpga-based multi-accelerator edge computing systems. In *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022.

[101] S. A. Razavi, M. S. Zamani, and K. Bazargan. A tileable switch module architecture for homogeneous 3d fpgas. In *2009 IEEE International Conference on 3D System Integration*, pages 1–4. IEEE, 2009.

[102] I. Rekleitis, G. Dudek, and E. Milios. Multi-robot cooperative localization: a study of trade-offs between efficiency and accuracy. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2690–2695 vol.3, 2002.

[103] T. Ren, J. Niu, B. Dai, X. Liu, Z. Hu, M. Xu, and M. Guizani. Enabling efficient scheduling in large-scale uav-assisted mobile edge computing via hierarchical reinforcement learning. *IEEE Internet of Things Journal*, 2021.

[104] S. Rezaei, E. Bozorgzadeh, and K. Kim. Ultrashare: Fpga-based dynamic accelerator sharing and allocation. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–5, 2019.

[105] J. Rodríguez-Araújo, J. J. Rodríguez-Andina, J. Fariña, and M.-Y. Chow. Field-programmable system-on-chip for localization of ugvs in an indoor ispace. *IEEE Transactions on Industrial Informatics*, 10(2):1033–1043, 2014.

[106] S. I. Roumeliotis. *Robust mobile robot localization: from single-robot uncertainties to multi-robot interdependencies*. PhD thesis, University of Southern California, 2000.

[107] S. I. Roumeliotis and G. A. Bekey. Distributed multirobot localization. *IEEE Transactions on Robotics and Automation*, 18(5):781–795, 2002.

[108] M. Sadeghi, S. A. Razavi, and M. S. Zamani. Reducing reconfiguration time in fpgas. In *2019 27th Iranian Conference on Electrical Engineering (ICEE)*, pages 1844–1848. IEEE, 2019.

[109] V. K. Sarker, J. P. Queralta, T. N. Gia, H. Tenhunen, and T. Westerlund. Offloading slam for indoor mobile robots with edge-fog-cloud computing. In *2019 1st international conference on advances in science, engineering and robotics technology (ICASERT)*, pages 1–6. IEEE, 2019.

[110] M. Sharifi, A. Abhari, and S. Taghipour. Modeling real-time application processor scheduling for fog computing. In *2021 Annual Modeling and Simulation Conference (ANNSIM)*, pages 1–12, 2021.

[111] F. Shi, X. Tuo, S. X. Yang, J. Lu, and H. Li. Rapid-flooding time synchronization for large-scale wireless sensor networks. *IEEE Transactions on Industrial Informatics*, 16(3):1581–1590, 2019.

[112] O. Shorinwa, J. Yu, T. Halsted, A. Koufos, and M. Schwager. Distributed multi-target tracking for autonomous vehicle fleets. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3495–3501. IEEE, 2020.

[113] A. K. Sikder, A. Acar, H. Aksu, A. S. Uluagac, K. Akkaya, and M. Conti. Iot-enabled smart lighting systems for smart cities. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 639–645. IEEE, 2018.

[114] D. Simon. Kalman filtering with state constraints: a survey of linear and nonlinear algorithms. *IET Control Theory & Applications*, 4(8):1303–1318, 2010.

[115] S. Singh, R. Sulthana, T. Shewale, V. Chamola, A. Benslimane, and B. Sikdar. Machine-learning-assisted security and privacy provisioning for edge computing: A survey. *IEEE Internet of Things Journal*, 9(1):236–260, 2021.

[116] J. Soh and X. Wu. A modular fpga-based implementation of the unscented kalman filter. In *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 127–134. IEEE, 2014.

[117] A. Tazarv, S. Labbaf, A. M. Rahmani, N. Dutt, and M. Levorato. Data collection and labeling of real-time iot-enabled bio-signals in everyday settings for mental health improvement. In *Proceedings of the Conference on Information Technology for Social Good*, pages 186–191, 2021.

[118] A. Tazarv, S. Labbaf, S. M. Reich, N. Dutt, A. M. Rahmani, and M. Levorato. Personalized stress monitoring using wearable sensors in everyday settings. In *2021 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, pages 7332–7335. IEEE, 2021.

[119] H.-Y. Ting, T. Giyahchi, A. A. Sani, and E. Bozorgzadeh. Dynamic sharing in multi-accelerators of neural networks on an fpga edge device. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 197–204, 2020.

[120] J.-C. Trujillo, R. Munguia, E. Guerra, and A. Grau. Visual-based slam configurations for cooperative multi-uav systems with a lead agent: an observability-based approach. *Sensors*, 18(12):4243, 2018.

[121] H. Wang, F. Yu, M. Li, and Y. Zhong. Clock skew estimation for timestamp-free synchronization in industrial wireless sensor networks. *IEEE Transactions on Industrial Informatics*, 17(1):90–99, 2020.

[122] Y. Xiong, N. Wu, Y. Shen, and M. Z. Win. Cooperative localization in massive networks. *IEEE Transactions on Information Theory*, 2021.

[123] B. Xu, S. Li, A. A. Razzaqi, and J. Zhang. Cooperative localization in harsh underwater environment based on the mc-anfis. *IEEE Access*, 7:55407–55421, 2019.

[124] C. Xu, S. Jiang, G. Luo, G. Sun, N. An, G. Huang, and X. Liu. The case for fpga-based edge computing. *IEEE Transactions on Mobile Computing*, 2020.

[125] Z. Yan, W. Guan, S. Wen, L. Huang, and H. Song. Multirobot cooperative localization based on visible light positioning and odometer. *IEEE Transactions on Instrumentation and Measurement*, 70:1–8, 2021.

[126] H. H. Yang and D. Wong. Optimal min-area min-cut replication in partitioned circuits. *IEEE transactions on computer-aided design of integrated circuits and systems*, 1998.

[127] Z. Yang, J. Pan, and L. Cai. Adaptive clock skew estimation with interactive multi-model kalman filters for sensor networks. In *2010 IEEE International Conference on Communications*, pages 1–5. IEEE, 2010.

[128] R. D. Yates, Y. Sun, D. R. Brown, S. K. Kaul, E. Modiano, and S. Ulukus. Age of information: An introduction and survey. *IEEE Journal on Selected Areas in Communications*, 39(5):1183–1210, 2021.

[129] N. Yazdani and D. E. Lucani. Online compression of multiple iot sources reduces the age of information. *IEEE Internet of Things Journal*, 8(19):14514–14530, 2021.

[130] V. Yazici and C. Aykanat. Constrained min-cut replication for k-way hypergraph partitioning. *INFORMS Journal on Computing*, 26(2):303–320, 2013.

[131] S. Zarandi and H. Tabassum. Delay minimization in sliced multi-cell mobile edge computing (mec) systems. *IEEE Communications Letters*, 25(6):1964–1968, 2021.

[132] K. Zhang, Y. Mao, S. Leng, S. Maharjan, and Y. Zhang. Optimal delay constrained offloading for vehicular edge computing networks. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[133] Y. Zhang, M. Hsiao, Y. Zhao, J. Dong, and J. J. Engel. Distributed client-server optimization for slam with limited on-device resources. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5336–5342. IEEE, 2021.

[134] Y. Zhao, X. Li, Z. Jia, L. Ju, and Z. Zong. Dependency-based energy-efficient scheduling for homogeneous multi-core clusters. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 1299–1306. IEEE, 2013.

[135] J. Zhu and S. S. Kia. Cooperative localization under limited connectivity. *IEEE Transactions on Robotics*, 35(6):1523–1530, 2019.

[136] Z. Zong et al. Ead and pebd: two energy-aware duplication scheduling algorithms for parallel tasks on homogeneous clusters. *IEEE Transactions on Computers*, 2011.