OPENMP BASED ACTION ENTROPY ACTIVE SENSING IN CLOUD COMPUTING

BY

YUWEI LIU

Submitted in partial fulfillment of the requirements for the

degree of

Master of Science

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May, 2020

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Yuwei Liu

candidate for the degree of Master of Science

Committee Chair

Vincenzo Liberatore

Committee Member

Vincenzo Liberatore

Committee Member

M. Cenk Cavusoglu

Committee Member

An Wang

Date of Defense

March 17, 2020

Table of Contents

List of Tables

List of Figures

OpenMP based Action Entropy Active Sensing in Cloud Computing

Abstract

By

YUWEI LIU

Action entropy active sensing is a newly proposed task-oriented active sensing method that selects the sensing action by minimizing the uncertainty in the future task. When the number of the particles increases, the time cost will become tremendously high. In this work, we applied cloud and parallel computing to the original model to improve the performance of the action entropy active sensing model. The computation part is set on Amazon Web Service EC2 instances and the algorithm is revised by implementing OpenMP. The communication between the local machine and the cloud is under the structure of Robot Operating System. Our simulation results and evaluations illustrate that the new model we proposed can improve the performance of action entropy active sensing.

Chapter 1 Introduction

Currently, robots already participate in our daily lives such as auto-driving car, sweeping machine etc. All of them have a system to deal with different data and situations, in other words, robotics problems, which is one category of sequential making problems because they are continuous in nature. In sequential decision making problems, modeling errors, changes in the environment and numerical approximations can lead to uncertainty, which is unavoidable and makes the problem much more complex. If we select the most useful sensor data and generate a correct adjustment to the system, the problems of uncertainty will be diminished. This method is applied to many robotics problems which are continuous and have tremendous complexity.

In a robotics problem, sensing actions are defined as actions that can have an influence on sensing but they will not have any effect on the state of the system just like the action we take to look at different directions and the surrounding environment while driving without changing the direction of the car. For a robot, sensing actions can be referred as the action that the robot takes to get the data. In this case, the method we mentioned above can be defined as the action that chooses the sensing action to get more useful data. This method is called *active sensing*.

Active sensing aims to reduce the uncertainty in the state of the system by optimizing variables such as entropy, conditional entropy and mutual information [1]. There is also a special case, when the current information of the state is used to complete a task in which it is not efficient to minimize the uncertainty because it may not select the action that gives the most useful information for the task. For instance, suppose that the state space is the position of a barrier that may occur on the right or left to a robot. The probability of the position of the barrier on the left or right is the same. If the robot wants to change the direction to its right, it needs to look to the right first to make sure that there is no barrier. Thus the position of the barrier is the only factor that determines whether the robot can change its direction. Looking to the right should be the sensing action the robot takes. If we try to minimize the uncertainty of the state space, the robot could look to either right or left because the probability of the barrier's position is the same. Indeed, looking to the left can also reduce the uncertainty, but does not help the robot to turn right. In this situation, minimizing the uncertainty may not provide the most useful information.

The action-entropy active sensing method, which choose the sensing actions that minimize the entropy of the future task actions, has better performance than the traditional active sensing method [1]. However, when performing that method to get the sensing action, an unpleasant period of time is spent which has a bad effect on the robot, especially when the robot needs to make a judgement in a short time.

This paper presents a new model based on the action-entropy active sensing model by using cloud computing. Our objective is to move the computing part of the model to the cloud to reduce computation time. A secure connection will be established between the local machine and the cloud. The cloud side will generate the sensing action and send it to the local machine. After the local machine performs the task, both side will update their states synchronously. Parallel computing will also be utilized on the cloud side to increase the efficiency of the proposed algorithm of the action-entropy active sensing.

The rest of the thesis will be introduced as follows. Background is introduced in Chapter 2. Related work is reviewed in Chapter 3. The new cloud active sensing model is described in Chapter 4. Simulation results are presented in Chapter 5. Discussion are taken in Chapter 6. Conclusion and future work are mentioned in Chapter 7.

Chapter 2 Background

In this chapter, we will introduce the background of the cloud robotics, the cloud service we use in the project, the connection method between the machine and the cloud and the parallel computing method.

2.1 Cloud robotics

The cloud is a model that provides a great amount of shared resources such as server, storage, network applications and services and can be accessed without limitations of location and hardware [2]. We can use the computing resources and get the result we demand only by sending the request via Internet from the local computer [3]. There are broad categories of cloud-related products we use in our daily lives such as Google Drive, which can store our document without buying a hardware or worrying about the storage. The cloud also makes it possible for users to share data [4].

With development of robot systems, Cloud Robot and Automation systems is a novel term proposed in recent years. The Cloud Robot and Automation system does not rely on the configurations on the local hardware which means there is no need to place all the functional hardware on the local robot. Computation, code and instructions are sent from the network and help the operation of the robot.

Fig 1 shows the model that most robots and automation systems interact with the cloud.

Fig 1 Cloud Robotics in daily life(reproduced from [2])

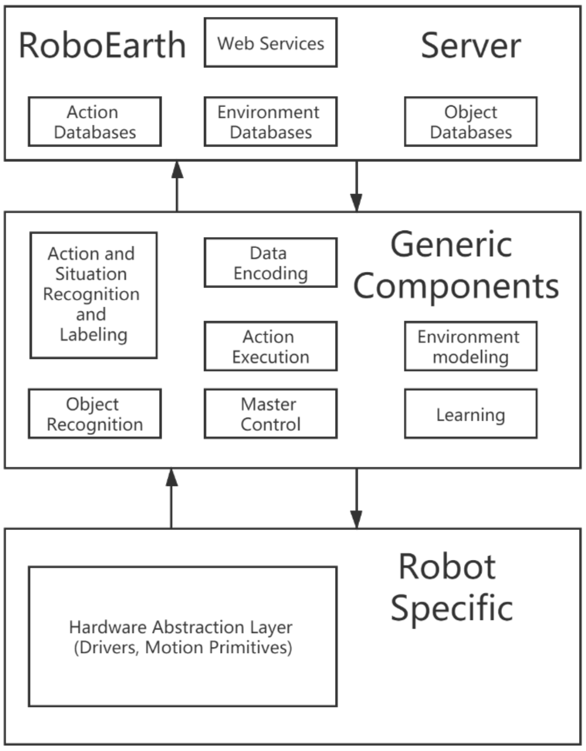

Fig 2 The structure of RoboEarth

The RoboEarth project was started in 2009 and drew a blueprint for the robot system in the future. It defined “World Wide Web” for robots over which the robots can share information and store the data on a huge database repository. Under the structure of RoboEarth, robots can communicate with each other and learn each other’s behavior and information on the environment which the other robots operate [5][6]. Fig 2 shows the architecture of the Roboearth Project.

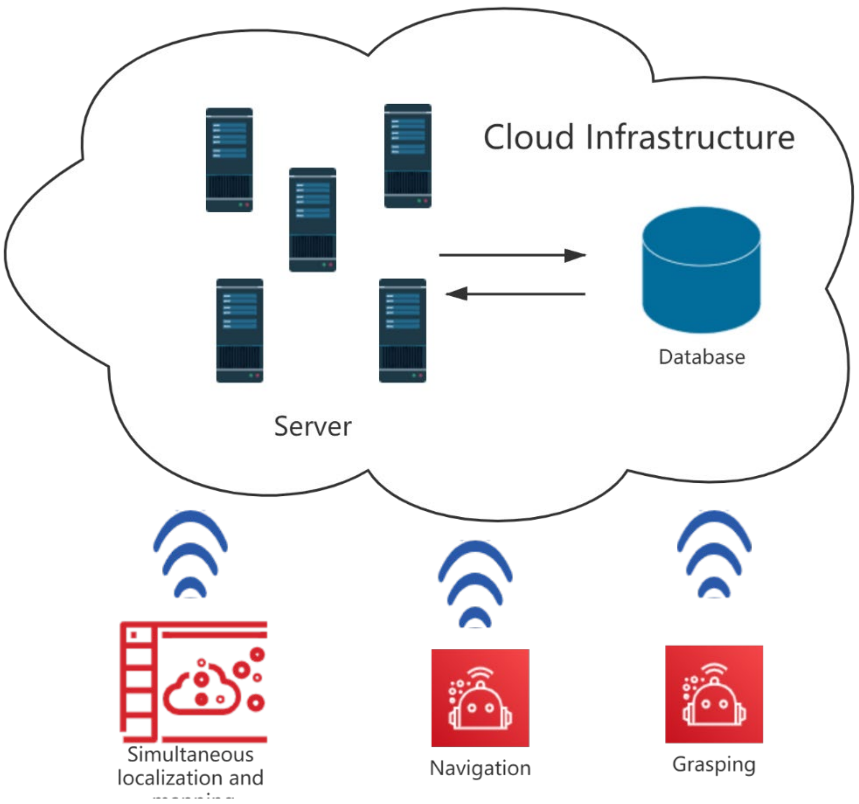

Fig 3 The Cloud Robotics Architecture

Cloud Robotics was first proposed in 2010 [8]. The main purpose of cloud robotics is to apply the convenience of the cloud to the performance of the robots. The robots can take great advantages of the storage, computing resources and other service from the cloud. The cloud also makes it easier for the communication between the robots.

The cloud robotics architecture is built by two major parts as Fig 3 shows: The cloud and the bottom facility and equipment. A large number of servers and databases forms the cloud part and the bottom part usually contains mobile robots, robot machines and other equipment [7].

2.2 Amazon Web Service

Nowadays, large companies have already developed efficient computing architecture, which can be provided for personal and commercial use. Amazon Web Service is one among them. The "Cloud" is used to describe those services where the computing is processed because the user can only access it through internet.

Since July 2002, Amazon Web Service has been providing the user with multiple products such as computing, networking, content delivery etc. [9][10]. It has been spread all over the world and have multiple data centers in different countries. Launched in 2006, Elastic Cloud Compute, which is called EC2, is the product that we use for this project. EC2 service provides users with virtual machines in different specifications. The price ranges according to the performance and physical settings. A virtual machine, which is installed with the specific version of operating system packages, is named as an Amazon Machine Image(AMI) [11]. In this project, data collected by the local machine will be transmit to the EC2 instance on AWS, processing on the cloud and then send it back to local. Every single EC2 compute unit has the same performance as a 1-

1.2 GHz Intel 2007 Xeon or Opteron processor. Taking cost into consideration, c5n-9x large instance is chosen in this project. The instance we choose has 36 cores which is sufficient for the parallel computing and the price is $1.94 per hour.

2.3 OpenVpn Connections

Virtual Private Network is a network that provides private services on the existing public networks. VPN can provide a method, which is secure for data and IP packages transmitted between networks [12]. By using VPN, secure, encrypted network connection can be established over insecure public networks.

IPSec, *PPTP* and *TLS* are three major types of VPN[13][14][15]. *IPsec* is the most popular type of VPN communication that works on the network layer. It is a framework that offers private communications on the public networks. IPsec also contains security protocols for mutual authentication and the negotiation of cryptographic keys for use during the session on the network layer. All the data they transmit via the VPN will be completely encrypted. However, the drawback of IPSec is lack of flexibility and control when end goal is to protect a specific application [16]. In addition, IPSec is not widely deployed nowadays [18].

The Point-to-Point Tunneling Protocol (PPTP), which exists on the data link layer, is also one of the protocols that implement the function of VPN. It launches a TCP control channel between the client and server. Generic Routing Encapsulation is used to encapsulate PPP packets to send over this same control channel [17]. The PPTP method has some security flaws in the protocol and is now obsolete [14].

TLS (Transport Layer Security) *VPN* is the VPN built on the Transport Layer Security protocol which works on the application layer. *TLS VPN* is much more flexible than *IPSec* [[19]. It just needs some software to setup a server without any hardware or network requirements. It can also be divided to *TLS portal VPN* and *TLS tunnel VPN* [15]. *TLS portal VPN* allows user to connect to a gateway or portal so that the user can get access to multiple network by using TLS connection. *TLS tunnel VPN* requires the user to install third party software to use it. This method will transmit all the internet traffic to the VPN server regardless of the portal.

OpenVpn is primarily a *TLS tunnel VPN* that can tunnel any local network adapter through a single TCP or UDP port. Most of OpenVpn relies on the security provided by OpenSSL. It contains two different methods for authentication: Static Key mode and TLS mode[20][21].

In the static key mode, keys have already been distributed to the user before the connection. The keys work as the authentication mechanism and means of encrypting/decrypting the data from the tunnel.

In TLS mode, the data is encrypted through symmetric cryptography. The sender and the receiver negotiate to decide the algorithm and the cryptographic keys to use before the transmission starts. Public-key cryptography is used to identify the communicating parties over an unencrypted channel. To make the connection more reliable, a message authentication code, which ensures the integrity of the message, is included in each message to prevent the data loss.

For the security of the connection between AWS and the local network, Amazon uses OpenVPN Access Server to establish VPN connection between the cloud and the local. The OpenVPN Access Server can integrate the function of the OpenVPN server, the capabilities of enterprise management, the simplified user interface for the connection and the software that is designed to run on different kinds of operating systems. Indeed, Amazon makes the OpenVpn more powerful by adding a web interface. We only need to use the browser to login to the webpage and set the configurations of OpenVPN without the complex configuration files.

2.4 ROS Communication

Robot Operating System is a system that enables the communication between robots and other devices through the TCP/IP network [22]. The ROS core contains the different API to create ROS nodes which can use the network. ROS core also has the scripts and command lines that can monitor the connections, nodes and messages on the network or other interface such as rviz or gazebo. ROS also has a very large database of packages, which are designed specifically to an application or a device.

ROS architecture contains three levels: the file system level, the computation graph level and the community level. The *file system level* contains the resources on the disk such as packages and manifests. The *community level* contains the resources, for example the distributions, repositories and ROS wiki, which can be shared with different developers and communities. The communication process is included in the *computation graph level* where ROS processes the data together in a peer-to-peer network [22]. ROS communication has few key concepts that the user must know: Nodes, Master, Messages, Topics and Services.

Nodes are the basic units in the ROS network that operate computation, perform some tasks or communicate with the ROS network. The nodes will register in the network with a unique ID and list all the topics and services that they want to communicate. The nodes can be written in many programming languages such as C++ or Python.

The master is a special node which will be started when the network is built. It will handle the registration, subscriptions and disconnection of every node to the network and links for each topic or service so that messages can reach its destination.

Messages are the packets that are sent on the network. It is a data structure which contains typed fields such as Boolean, int bits, float bits, string, time and duration. The primary communication method in which nodes send data to each other is publishing messages to topics.

ROS Topic is defined as the name of the hubs over which nodes communicate with each other. Topics handles the publish-subscribe communication. Nodes can connect to a topic by its name either as a publisher which can send data or as a subscriber which can receive the data. Topics do not have limitations on the number of nodes that work as publisher or subscriber. The topic also has a defined message template and the data must be sent in the format of the template.

Services are an alternative to the topics that use a request and response paradigm instead of publish-subscribe system. The node that uses services will only receive the data after it makes a request. A service is provided by only one node and only one request can be sent at the same time. The topic contain description of both the request and the response message type.

2.5 OpenMP

Nowadays, most computers are equipped with multicore processors. Parallel processing is widely used to improve the efficiency and computing performance of the computer application. On the one hand, some computer applications may need a huge amount of computing which exceeds the maximum capability of the fastest single processor, making it necessary to harness the aggregate capabilities of multiple processors to offer a faster speed of computing. On the other hand, even if the application is in the capability of a high-speed single processor, the price is much lower when performing under multiple normal processors, which may have the similar performance.

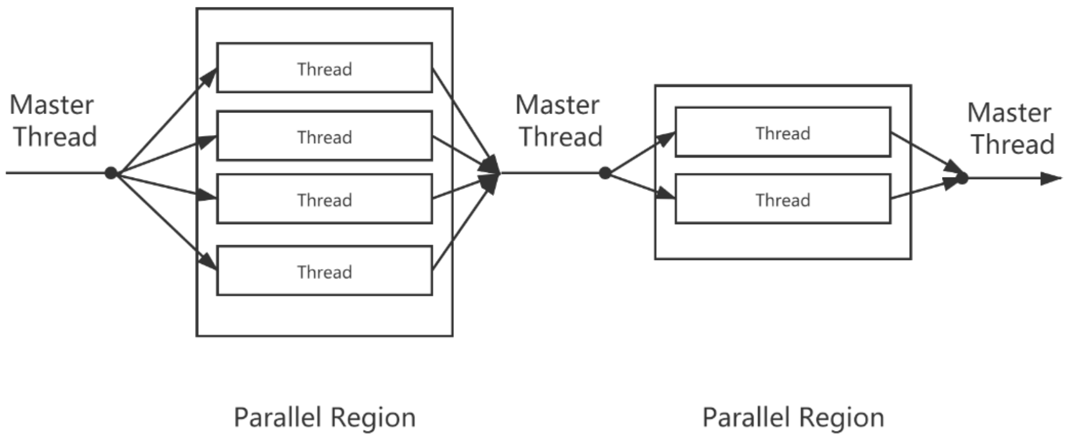

Fig 4 The structure of OpenMP

OpenMP is an application programming interface (API) which is used in parallel programming. It supports multiple programming language such as C, C++ and Fortran [23]. The structure of OpenMP is referred as the master-fork architecture. The main task, which is planned to be executed consequently, will act as the master thread. The master thread will divide its task to a fixed number of slave threads according the user's configuration. Each slave thread runs its task, in other words, part of the code independently on different cores of the processor. Finally, when all the tasks and the parallel part are executed, the system will move on. Fig 4 shows the structure of

OpenMP.

For loop iterations, OpenMP can divide the loop in independent groups which are executed in parallel. Each group is allocated to a thread that performs the iteration. All the iterations can be executed in parallel at the same time [23]. The method to divide the task to different groups is flexible. In OpenMP, scheduling is the function that decide the number of groups and the iterations in each group. It also determines how the threads deal with the iterations. A group that is not empty is termed as chunk.

There are five different types of scheduling in OpenMP: static, dynamic, guided, auto and runtime [24].

Static scheduling means the loop will be divided in groups that have the same number of iterations. When the total number of the iterations cannot be divided by the number of groups, the group size will be set as equal as possible. In dynamic scheduling, a queue will be set store the iterations which will be executed next. When a thread finish its task, the iteration in the queue will be sent immediately to the thread from the top of the queue. Extra overhead may be involved in dynamic scheduling. Guided scheduling also has a queue to store the iterations. The difference between guided scheduling and dynamic scheduling is that the number of iterations in each group changes in the processing. The size will change from large to small to make the load of the iterations balanced. Auto scheduling can choose different types of mapping of iterations to threads but it is not usually used. In runtime scheduling, an environment variable is used to specify which one of the static, dynamic and guided should be used for scheduling.

2.6 Fast Library for Approximate Nearest Neighbors

Nearest neighbor search(NSS) is one of the important problems in the study of classification, image recognition, machine learning and etc. It can be defined as follows: given a set of points $P = p_1,p_2,...p_n$ in a metric space M, given a query point $q \in M$, we need to find out the point in P that is nearest to q.

FLANN (Fast Library for Approximate Nearest Neighbors) is a library proposed in 2009 for solving the NSS problem by performing fast approximate nearest neighbor searches [25]. Although it is written in C++, it can be used in many other programing languages such as MATLAB and Python when the library is provided.

FLANN contains many classes and functions: Flann::Index is the nearest neighbor index class which is used to abstract neighbor search indexes. The class has a distance function that computes the distance between two different points. Flann::Index::buildIndex builds the nearest neighbor search index. Functions such as Flann::Index::addPoints, Flann::Index::removePoint and Flann::Index::getPoint are used to change the points or get the points' information after the foundation of the index. There are also searching method such as Flann::Index::knnSearch, Flann::Index::radiusSearch that perform different searching method for a set of query points.

Chapter 3 Related work

This chapter will introduce works that are related to active sensing algorithms. We will also describe research that aims to improve the algorithm's performance by using parallel computing and OpenMP.

3.1 Early Research on active sensing

Active sensing algorithm was first proposed by Schmaedeke in 1993 [26]. Based on the information theory, sensor-target assignment is performed by maximizing the difference in Kullback-Leibler divergences formulated as a linear program. Manyika and Durrant-Whyte proposed a data fusion method that utilized Kalman filter to maximize the information content of the state [27]. In 2005, Partially-Observable Markov Decision Processes (POMDPs) was proposed to model the decision making problems under uncertainty **Error! Reference source not found.**. However, only small problems with few states can be solved. To improve the efficiency of POMDPs, Littman et al. proposed Q-learning Markov Decision Process which ignored the sensing aspect of the task [29]. Pineau et al proposed Point-Based Value Iteration (PBVI) method, which used a small set of belief points and the hyperplanes at the belief points to approximate a POMDP value function. The research mentioned above mainly focus on the POMDP algorithms designed for the discrete problems. Based on PBVI, Porta et al extend the PBVI method to a subset of sequential POMDPs where Gaussian mixtures are used to describe the model [30].

3.2 Active Sensing via Task-Action Entropy Minimization

Different from POMDP-based method, Greigarn, et al presented a new task-oriented active sensing method in 2018 that integrated the task into active sensing by choosing sensing actions to minimize the uncertainty of future test actions. A state-space planner is designed to provide a policy to minimize the uncertainty. According to the policy, a belief over a task action space is obtained by mapping the belief over the state space. A nearest-neighbor method is used to estimate the future task action from the particles. Moreover, this method can obtain the information and perform the task at the same time.

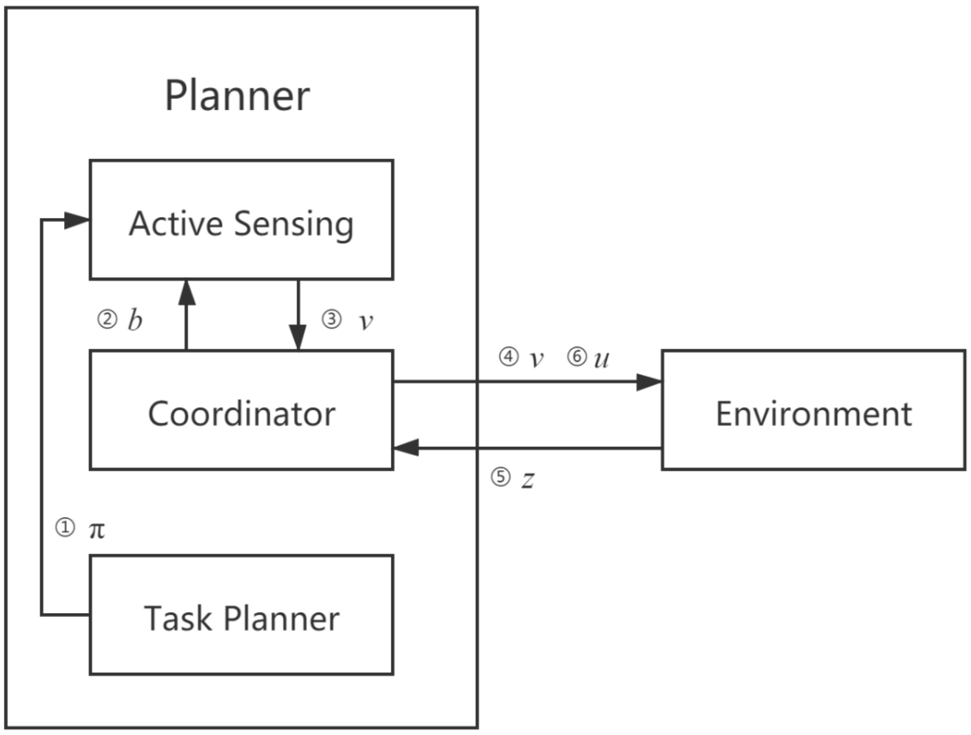

Fig 5 The planner model

Fig 5 shows the model of this method in which the algorithm collaborates with a task planner to solve the decision-making problem under uncertainty. In this model, π stands for the policy generated by the task planner. The policy is the rule that the active sensing module uses to select the sensing action. b is the internal belief of the coordinator which contains the environment data of the robot. v is the sensing action which is similar to the action looking to the right in robot moving analogy mentioned in Chapter 1, u is the task action which is similar to moving to the right in the robot moving example. z is the observation got from the environment just like the information the robot gets when it looks to its right.

The active sensing module performs the active sensing task, the task planner solves the planning problem in the state space and the coordinator calculates the internal belief and interact the environment. The process is described as follows:

1. In step 1, when the planner is initialized, the task planner will first solve the planning problem in the state space and get the policy π. Then the policy will pass to the active sensing module.
2. In step 2, the coordinator will pass its original internal belief b to the active sensing module. In this process, the active sensing module gets the initial information of the robot such as the location. Rules for selecting the best sensing action are also received at the same time.
3. Then in step 3, after the internal belief b and policy π are received, the active sensing module will map the belief through the policy to get a probability

distribution of the task action. After calculating the task-action entropy, the active sensing module will get a sensing action *v* which minimizes the entropy of task action and passes it to the coordinator. In other words, the active sensing module returns the proper sensing action that ask the robot to get the information, for example looking to the right in the robot moving scenario, which may help the robot to reach its goal, from the environment.

4. In step 4, the coordinator then performs the sensing action *v* in the environment. The robot looks to the right to grab image information in our scenario.
5. In step 5, the coordinator receives the measurement z, which is also named as the observation such as the image information grabbed by the robot. Then the coordinator updates its belief. Namely, the robot updates the information of the environment in its own system according to the observation it got.
6. Then in step 6, the coordinator selects the task action u according to the policy based on the most likely state from the most recent belief. Then the task action u is performed and the coordinator updates its internal belief according to the belief prediction. In our scenario, the Robot turns right when no obstacles are found and update its location information.

According to Greigarn,'s paper, the belief is defined in formula (1):

$$b(x_t) = p(x_t | u_{1:t}, v_{1:t}, z_{1:t}), \forall x_t \in \chi \qquad (1)$$

χ stands for the state space, z stands for the measurement space. *p()* is the probability distribution.

When the robot performs the sensing action and receive the measurement z, the belief is updated according to formula (2):

$$b(x_t) = \eta p(x_t|u_t, x_t)\bar{b}(x_t) \quad (2)$$

The coordinator predicts the belief according to formula (3):

$$\bar{b}(x_t) = \sum_{x_{t-1}\in\chi} p(x_t|u_t, x_{t-1})b(x_{t-1}) \quad (3)$$

The active-sensing algorithm is shown in Algorithm 1.

Algorithm 1 Minimum Task-Action Entropy Active Sensing

```
1: procedure GETSENSINGACTION($\boldsymbol{\pi}, \boldsymbol{P_x}$)
2:   for all $\mathbf{v} \in \mathbf{V}$ do
3:       $\boldsymbol{h_v = 0}$
4:       for $\boldsymbol{i = 1, \dots, M}$ do
5:           sample $x$ from $P_x$
6:           sample $z$ from $p(z|v, x)$
7:           $\boldsymbol{P'_x = MeasurementUpdate(P_x, v, z)}$
8:           $\boldsymbol{P_u = \pi(P'_x)}$
9:           $\boldsymbol{h_v {+}= \hat{h}P_u(u)}$
10:      end for
11:  end for
12:  return $argmin_v h_v$
13: end procedure
```

In the algorithm, the input is the policy π and the particles P_x which represents the internal belief $\bar{b}$. For each sensing action, the algorithm calculates the task-action entropy. Then the sensing action that has the smallest entropy will be returned.

To support his theory, he proposed algorithms which reduced the ambiguity of the task-related sensing action that minimizes the conditional entropy of the next task-related action. Simulations with discrete systems and continuous systems has been performed by doing the experiments which contain Fork in The Road, Symmetric Corridor, Peg-in-Hole and Peg-in-Maze Problem. After the experiment, Greigarn drew the conclusion that the proposed has higher success rate than other POMDPs algorithms[1].

3.3 Research on algorithm performance on parallel computing and cloud computing.

Cloud computing is now widely used in research due to its computing resources and worldwide accessibility [31]. Some research has been done in recent years apply cloud computing to increase the speed of the algorithm. Also, parallel computing is widely applied in the simplification of algorithms. Most research deals with reduction of the computing time of algorithms that have tremendous amount of test data.

Yang et al applied OpenMP to Winograd parallel algorithm of matrix multiplication [32]. Winograd algorithm was widely used in matrix multiplication. However, when the matrix dimension to a large number, the time cost is huge. Instead of changing the algorithm behavior, they applied OpenMP to the Winograd algorthim. The algorithm is divided to three parts. Two parts runs at the same time on two independent threads and store the data in an array. Then the last part deals with the data in the array. They test the algorithm by semi-classical molecular dynamics test and get the result that the OpenMP based algorithm is 20 times faster than the original one.

There is research that applies the parallel computing on the cloud. Reyes-Ortiz et al applied parallel computing to the cloud to compare two distributed computing frameworks implemented on commodity cluster architectures [33]. Google Cloud Platform service was used to run and evaluate the algorithms. They got the result that MPI/OpenMP had better performance in processing speed and consistency. Peng and Wang proposed a cloud computing application method based on OpenMP/MPI which was feasible, effective and superior to the traditional parallel computing method [34]. Wottrich et al found out that factors such as overhead in the communication, workload balance and fault tolerance had bad influence on the performance of cloud-based parallel programming and they proposed a new model based on OpenMP and MapReduce. The proposed model was tested on Amazon AWS services and the result illustrated that their model has a better control on the parallel computing, all the effects by the factors are decreased to an acceptable rate [35].

Methods with hybrid parallel computing and GPU, in which parallel computing part is executed by the GPU cores instead of CPU, are also widely implemented in recent years. By moving the parallel computation part to the GPU, the algorithm may get the speedup at a large scale. Yang et al created a parallel programming algorithm that used hybrid CUDA, OpenMp and MPI programming [36][37][38]. Guo et al compares the performance of OpenMP, CUDA and OpenGCC by applying them to the algorithm that obtains the aerial target's reflected radiation [39]. Memeti et al also compares the productivity, performance and energy consumption of OpenMP and CUDA [40]. In most recent research, Sun et al applied OpenMP surface point cloud to solve effectively the problem of time delay in the simplification of the large-scale data that contains the information of the road surface cloud [41].

Chapter 4 Methodology

This chapter will introduce the revised model based on Greigarn,'s planner model. The action-entropy active sensing algorithm will be revised by using OpenMP. The detailed communication methodology will also be introduced.

4.1 The revised model

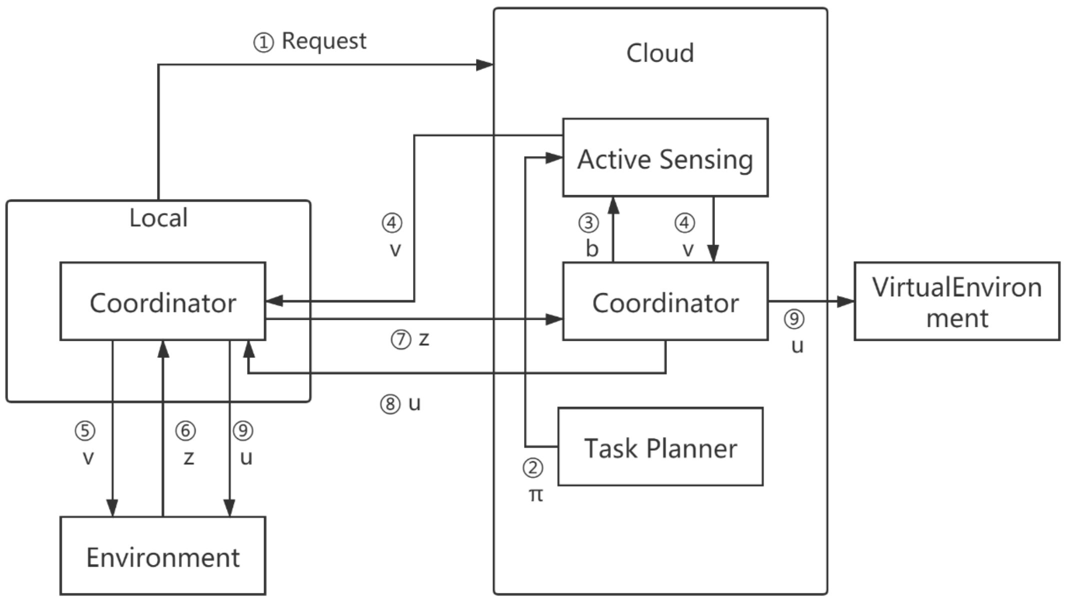

Fig 6 The Revised Model

Based on Greigarn's planner model and the theory of cloud robotics, we separate the planner into two parts. As Fig 6 shows, the local planner only has the coordinator part, which maintains the internal belief and interacts with the environment. On the cloud side, the cloud part does not change much from the original planner. The active sensing module generates the sensing action, the task planner module will generate the policy and the coordinator stores the belief and get observations form the environment. A virtual environment is set on the cloud because the cloud part also need to simulate the task action and update the belief.

Fig 7 shows the communication process between the local planner and the cloud. The process of Fig 6 is described as follows:

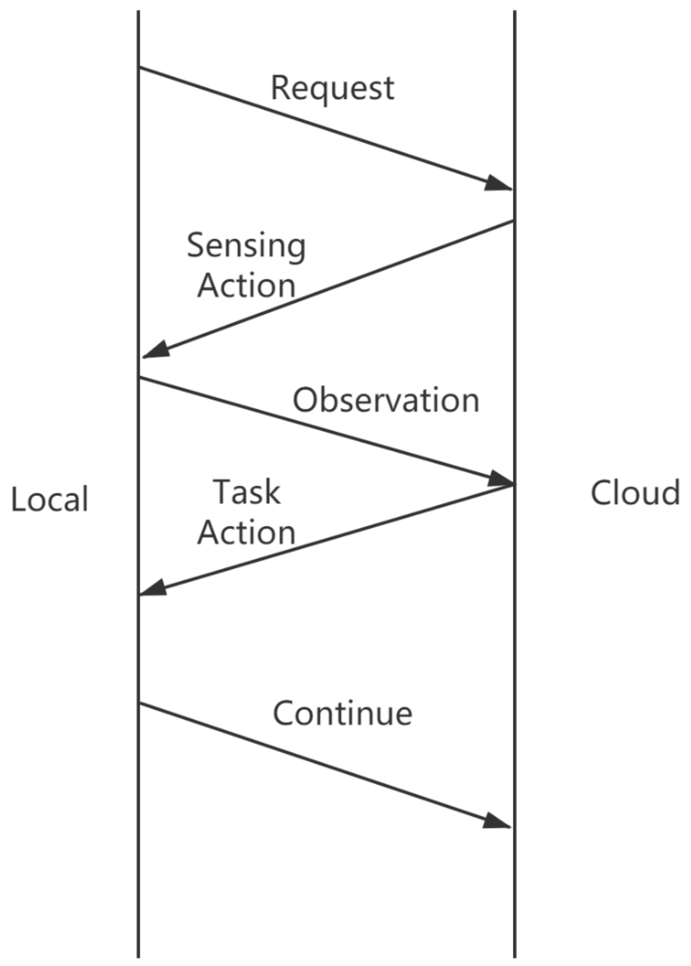

Fig 7 The Communication Process

1. In step 1, when both the local and the cloud planners are initialized, the local

part will send a request to the cloud for the sensing action.

2. Once the cloud receives the request, the task planner on the cloud side will solve the planning problem in the state space for the policy π and pass it to the active sensing module on the cloud side as step 2 shows.
3. Then in step 3, the coordinator on the cloud side passes the internal belief b to the active sensing module.
4. Then the active sensing module maps the belief according to the policy π to get a probability distribution of task action, calculate the task action entropy and select the best sensing action to minimize the task action entropy as step 4 shows. After the active sensing module on the cloud generates the best sensing action, the sensing action will be delivered to the coordinator on the local.
5. In step 5, when the coordinator receives the best sensing action, it will interact with the environment by performing the sensing action.
6. Then the local planner gets the observation from the environment in step 6. The observation is also known as the measurement z.
7. In step 7, the local coordinator will send the observation back to the coordinator on the cloud. After sending the observation, the local coordinator will update its belief according to the sensing action and the observation. When the coordinator on the cloud receive the observation, it will also update its belief according to the best sensing action and the observation received.
8. Next in step 8, the coordinator on the cloud will select the task action u

according to the policy and send it back to the local planner.

9. Then in step 9 the coordinator will perform the task action in the environment and update its internal belief.

After the cloud planner finish its task, it will wait for the local signal to continue. In the meantime, the local coordinator will also perform the task action in the real environment after it receives the task action. Then the local coordinator updates its internal belief and sends the continue signal which is the same as the request signal if the goal is not reached to the cloud for the next sensing action. When the goal is reached, a terminate signal will be sent to tell the cloud planner to stop computing.

4.2 Connection with Cloud

For the communication part, Amazon Web Services (AWS) is selected, which provides on-demand cloud computing platforms to individuals. Amazon Elastic Compute Cloud (Amazon EC2) is used to work as computing machines and VPN servers.

Two EC2 instances are launched on AWS. One works as the high performance machine and the other one works as the VPN server. Both of those instances are installed with Ubuntu as the operating system. The high performance machine uses Ubuntu 16.04 and has been installed with ROS Kinetic, which is the same version of ROS. The memory of the high performance machine is 8GB and the instance type is c5.9xlarge which has the highest speed of all the Ubuntu16.04 EC2 instances. The VPN server is configured by the default procedure of AWS OpenVpn set up. Both EC2 instances are in the same VPC, which stands for Virtual Private Cloud. The VPC makes it possible for the user to allocate a logically independent part of the AWS cloud. Both instances are configured in the same LAN and the same subnet. OpenVPN is used to get access to AWS. In this case, username/password method is used for the certifications in the connection.

To make the connection easier, AWS CLI, which stands for Command Line Interface, is used to set up OpenVPN by command lines in Linux. The username and password are stored in a text file. When the connection begins, the system will first read the file to get the username and password for the OpenVPN so that there is no need to type them every time when setting up the connection.

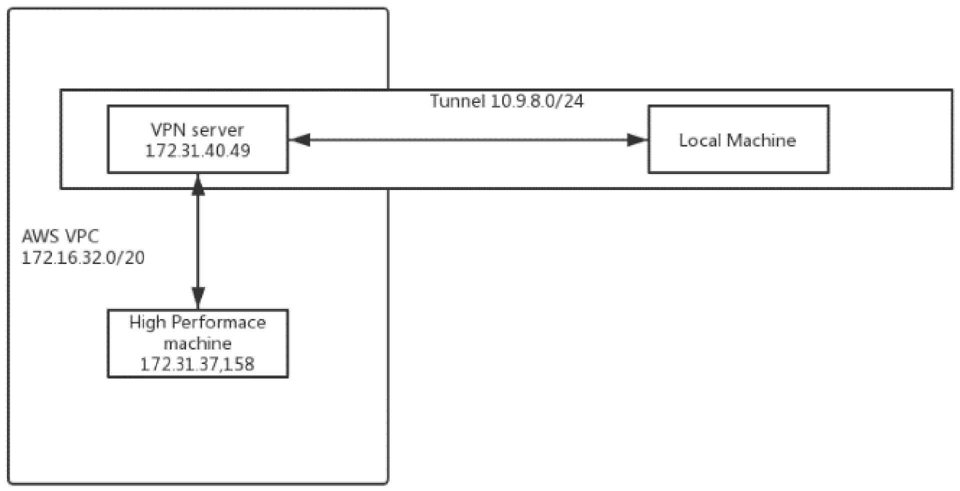

Fig 8 The Network Structure

The connection is described in Fig 8. Both EC2 instances are in the same subnet on AWS with the private IP address of 172.31.32.0/20. Routing is performed in the OpenVPN settings to allow the local machine get access to all the instances in VPN server's VPC as well as the subnet. The IP address for the tunnel, which is set up between the local machine and the VPN server is 10.9.8.0/24. In the routing table of the AWS VPC, the VPN server is set as the gateway of any destination to the address of 10.9.8.0/24. AWS Source/Destination check is disabled so that the traffic form the local machine can be sent to the high performance machine on the cloud. When running ROS on both cloud and the local. ROS core is launched on the high performance machine on the cloud and also the master URI IP address is set to its private IP address in the AWS VPC. All local machine's ROS IP are set to the IP addresses they are assigned in the VPN connection.

4.3 Communication model based on publish-subscribe

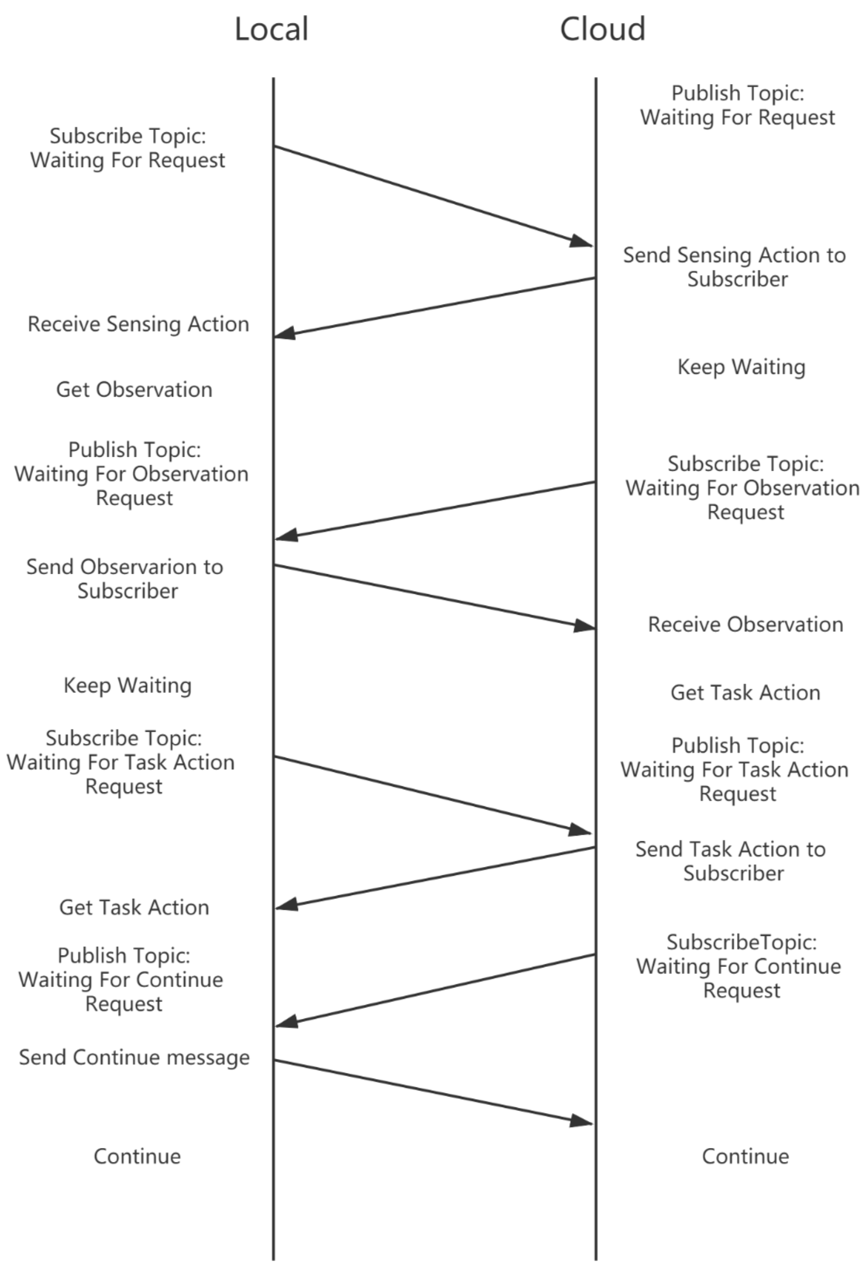
Local
Cloud
Publish Topic: Waiting For Request
Subscribe Topic: Waiting For Request
Send Sensing Action to Subscriber
Receive Sensing Action
Keep Waiting
Get Observation
Publish Topic: Waiting For Observation Request
Subscribe Topic: Waiting For Observation Request
Send Observarion to Subscriber
Receive Observation
Keep Waiting
Get Task Action
Subscribe Topic: Waiting For Task Action Request
Publish Topic: Waiting For Task Action Request
Send Task Action to Subscriber
Get Task Action
SubscribeTopic: Waiting For Continue Request
Publish Topic: Waiting For Continue Request
Send Continue message
Continue
Continue

Fig 9 Publish-Subscribe Model

Publish-subscribe communication is quite common in many ROS project. For each communication, the sender first announces a topic to ROS master and keep waiting for subscriber to subscribe. Once a receiver subscribes to the topic, the sender can send messages to all subscriber. Based on the theory, the model is designed as Fig 9 shows. Fig 10 and 11 show the process of both local and cloud planners. Fig 12 is the rqt graph in ROS which shows the connection between the topic and nodes.

The communication can be divided to four major parts:

Get Sensing Action: The cloud planner is initialized at first. It will publish the topic "Waiting for request" after the initialization. Then the cloud will continue waiting for the local to request the sensing action as the first decision in Fig 11 shows. As the first loop in Fig 10 shows, the local planner will subscribe the "Waiting for Request" topic when the initialization finished. If the local does not find the topic, it will continue send the request to ROS master to check whether the topic exists. Once the cloud planner gets the information from the ROS master that there is a subscriber request for sensing action, the cloud planner will generate the sensing action and send it back to the local planner. The sensing action data is stored in a message file which is a carrier of data in ROS. For example, the coordinate of the sensing action will be stored in the message file in the format of three float numbers. The local planner will receive the message file, read the data and store the data in local.

Get Observation: When the local planner gets the sensing action, the coordinator on the local will perform it in the environment and get the observation. After the observation is generated, the local planner will publish the topic "Waiting For Observation Request" on ROS master as the second decision part in Fig 10 shows. As the second loop in Fig 10 shows, the cloud planner will continue seeking for the topic “Waiting For Observation Request” on ROS master until the topic is found. Then the cloud planner will subscribe to the topic and the local planner will send the observation to the cloud via ROS message. In the meantime, the local planner will also update its belief according to the sensing action and the observation.

Get Task Action: When the cloud receives the observation, the coordinator will first update its belief and then publish the topic “Waiting For Task Action Request” on ROS master. Similarly, the local planner will continue looking for the topic on ROS master until it finds the topic. Both procedures are shown in the third loop of Fig 10 and Fig 11 respectively. Then the cloud planner will send the task action back to the local via ROS message. Once the message is sent and received, both cloud and local will perform the task action in the environment and update their believes according to the task action which is selected according to the policy generated from the task planner on the cloud.

Continue: As the fourth loop shows in both Fig 10 and 11, when the local planner finish performing the task action, it will publish the topic “Waiting For Continue Request” on ROS master. The cloud will keep looking for it on the ROS master until the connection is established. The local will send the continue signal. Then the local will move to the next step and start requesting for sensing action. The cloud will move to the next step and publish the “Waiting for Request” topic on ROS master once it receives the continue signal. When the goal is reached, the local planner will send the stop signal instead of the continue signal to the cloud planner. Then both two planners will stop computing. This process is shown in the fifth decision part in both Fig 10 and 11.

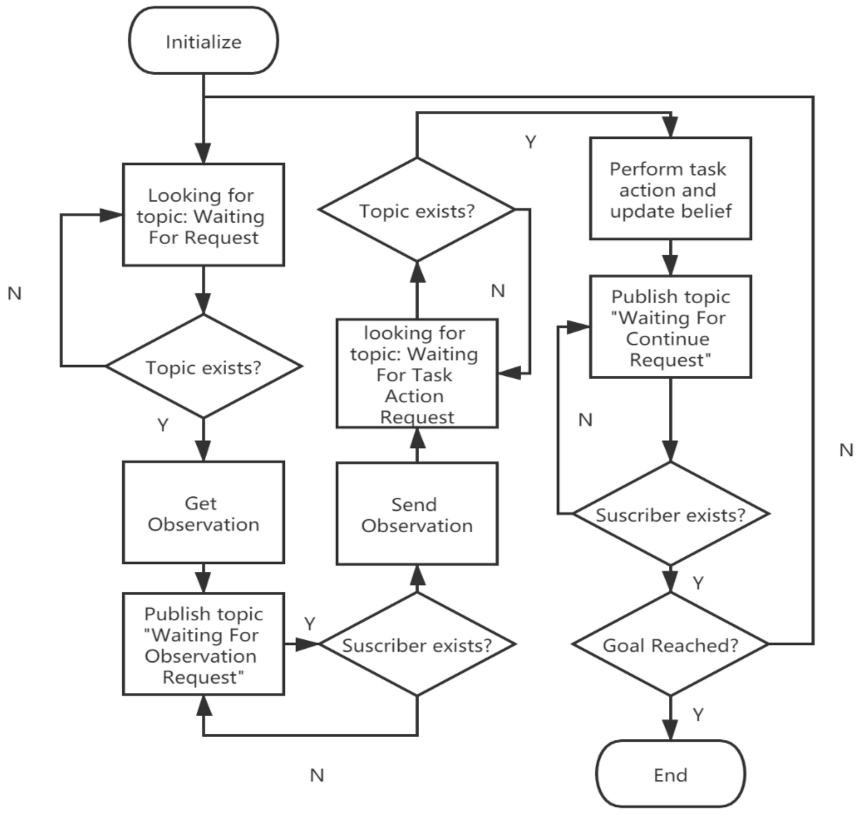
Initialize
Looking for topic: Waiting For Request
Topic exists?
N
Y
Get Observation
Publish topic "Waiting For Observation Request"
Y
Suscriber exists?
N
Send Observation
looking for topic: Waiting For Task Action Request
Topic exists?
N
Y
Perform task action and update belief
Publish topic "Waiting For Continue Request"
N
Suscriber exists?
Y
Goal Reached?
N
Y
End

Fig 10 The process of the local planner of Publish-Subscribe Model

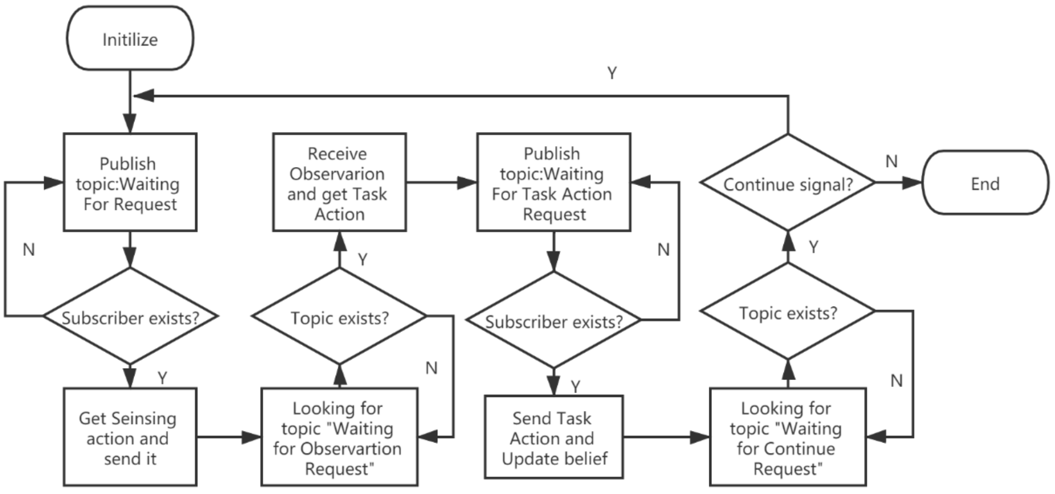
Initilize
Y
Publish topic:Waiting For Request
Receive Observarion and get Task Action
Publish topic:Waiting For Task Action Request
Continue signal?
N
End
N
Y
N
Y
Subscriber exists?
Topic exists?
Subscriber exists?
Topic exists?
Y
N
Y
N
Get Seinsing action and send it
Looking for topic "Waiting for Observartion Request"
Send Task Action and Update belief
Looking for topic "Waiting for Continue Request"

Fig 11 The process of the cloud planner of Publish-Subscribe Model

Fig 12 shows the rqt graph of the Publish-Subscribe communication model. In the communication process, the cloud planner sends sending actions and task actions to the local planner under the topic “Waiting For Request” and “Waiting For Task Action”. The local planner sends the observation and continue signal to the cloud planner under the topic “Waiting For Observation” and “Waiting For Continue Signal”.

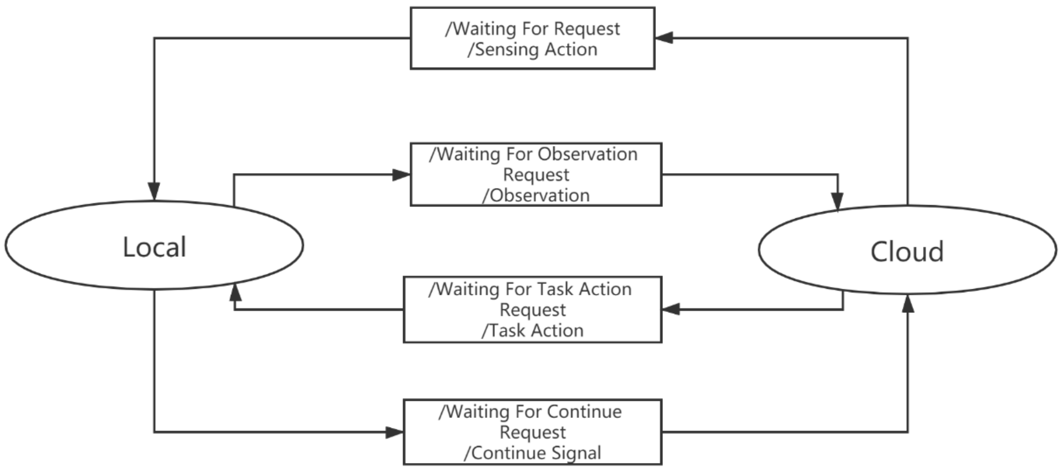
/Waiting For Request
/Sensing Action
/Waiting For Observation Request
/Observation
Local
Cloud
/Waiting For Task Action Request
/Task Action
/Waiting For Continue Request
/Continue Signal

Fig 12 The rqt graph of the Publish-Subscribe model

4.4 Communication model based on client-server

The client-server communication method is quite similar to the request and reply model. The data is stored in a service file which contains a pair of messages. One is the request and the other one is the reply. In the communication, the server node offers a service and the client node calls the service by sending request to the master. The master will help the client to find the related server. When the server receives the request, it will send back the reply message. Fig 13 shows the communication model for active sensing based on client-server.

Different from the publish-subscribe model which sets up connections every time through topic, the client-server model only establishes one connection at the beginning. However, in most client-server model, the message stored in service is maintained in the same type. For example, in the add-two integers model, the data type for the request part are two integers. The reply part only has one integer. In this situation, the client can continuously send the integers to the server and receive the sum of the integers. The server part only performs the add up function for the integers received. For the active sensing model, we have different types of data that need to be transported between the local and the cloud. Several different functions are also needed to get the actions. It is more complex to deal with different types of data and different functions in one connection.

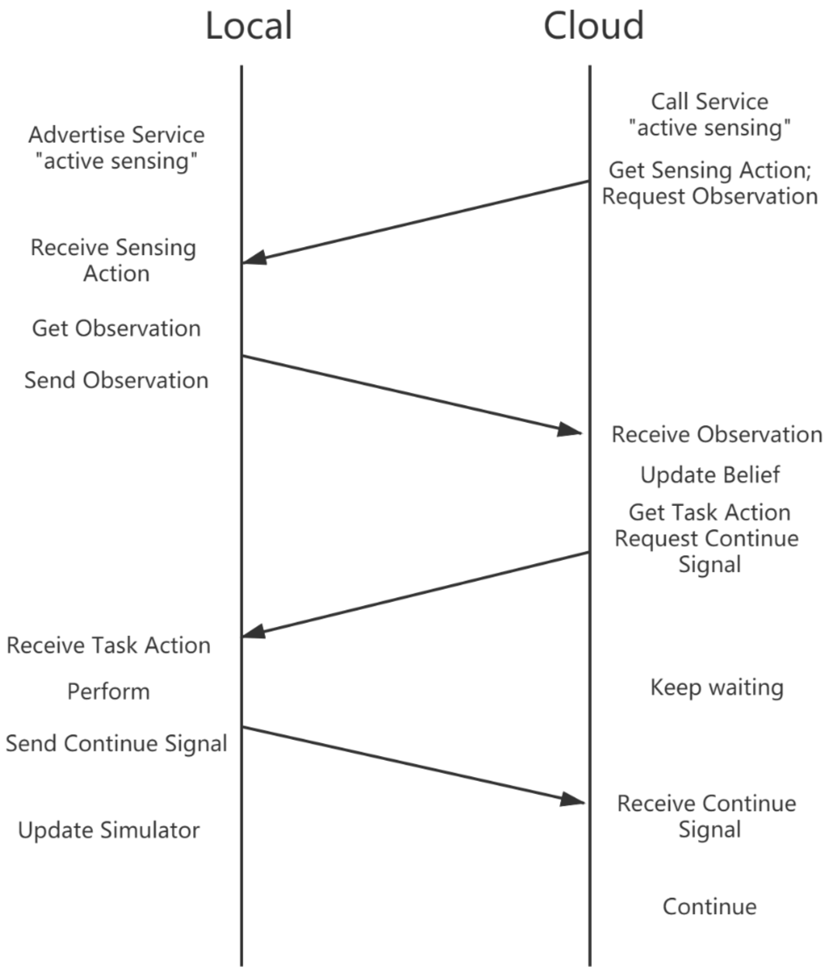

Fig 13 The Client-Server Model

To solve this problem, we designed the service file as Fig 14 shows. All the actions are vectors which have 3 float numbers to store their coordinate. An integer is inserted to represent the data type of the vector. Another integer is also inserted as continue signal. When the client sends the data, it will change the data type. The server will choose the related function to deal with the request data once it is received. Then the server will store the result of the function in the reply message and change the data type to ask the client to deal with the data in the related function of the client.

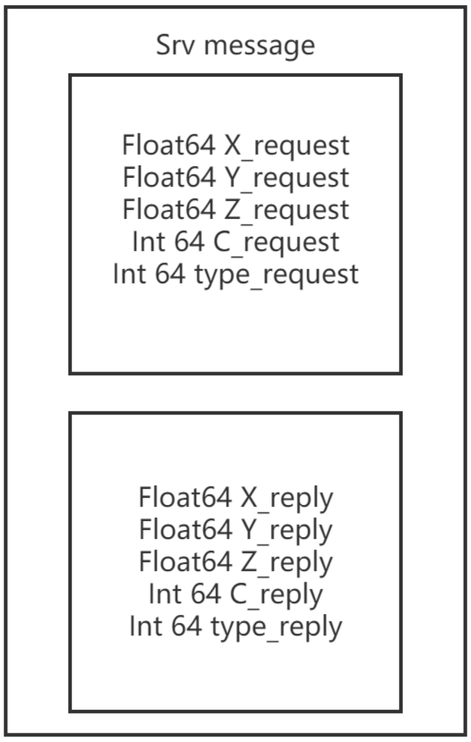

Fig 14 The Service message file

In this model, the process of the communication can only be divided to two major parts:

Get Sensing Action and Send Observation: As shown in Fig 15 and 16, when both the planners finish their initialization, the local planner will advertise the service “active sensing” to ROS master. In the meantime, as the first loop of fig 16 shows, the cloud planner will continue looking for the service until the connection is established. The task planner on the cloud will generate the policy π and send it to the active sensing module. In the meantime, the coordinator on the cloud will also send the belief to the active sensing module. The active sensing module will generate the sensing action and store it in the service file. Three float numbers X_request, Y_request, Z_request will store the data of the vector while C_request will be set to 0 as default. The integer

Type_request will be set to 1 which stands for the sensing action. As the first decision part in Fig 15 shows, the local planner will keep waiting for the service file. When the local planner receives the services file and the integer Type_request is equal to 1, the coordinator on the local will perform the sensing action and interact with the environment to get the observation. Then the observation will be stored in X_reply, Y_reply, Z_reply in the service file. The C_Reply will still remain to 0 because the whole process has not finished. The integer Type_reply will be set to 2 which means the type of the data is observation. After sending back the observation, the local planner will continue waiting for the new request.

Get Task Action and Send Continue Signal: The cloud planner will get the reply from the local planner immediately after the local sends back the observation. As the second decision part shows in Fig 16, if the Reply_Type is equal to 2 which means the observation, the coordinator on the cloud will update its belief and generate the task action and store it in the service file. The task action will replace the sensing action in the service file by changing X_request, Y_request and Z_request. The C_request will still be set to 0 and the Type_request will be set to 3 which means the task action. Then the cloud planner calls the service again. The local is still waiting for service request after they sent back the observation. As the second decision part shows in Fig 15, once the local planner receives the call from the cloud, it will read the data from the service file and get the task action. Then the local machine will perform the task action. If the local planner reaches its goal, it will set C_reply to 0 and stop running as the third decision part shows in Fig 15.If the goal is not reached, X_reply, Y _reply, Z_reply will be set to 0 and the C_reply will change to 1 which means tell the cloud to continue. The Type_reply will be set to 4 which means the continue signal. After reply to the cloud, the local planner will move to the next step to wait for the new sensing action. When the cloud planner receives the continue signal, it will move on and generate the next sensing action. When the cloud planner finds the continue signal to be 0, it will also stop running. This is shown in the third and fourth decision part in Fig 16.

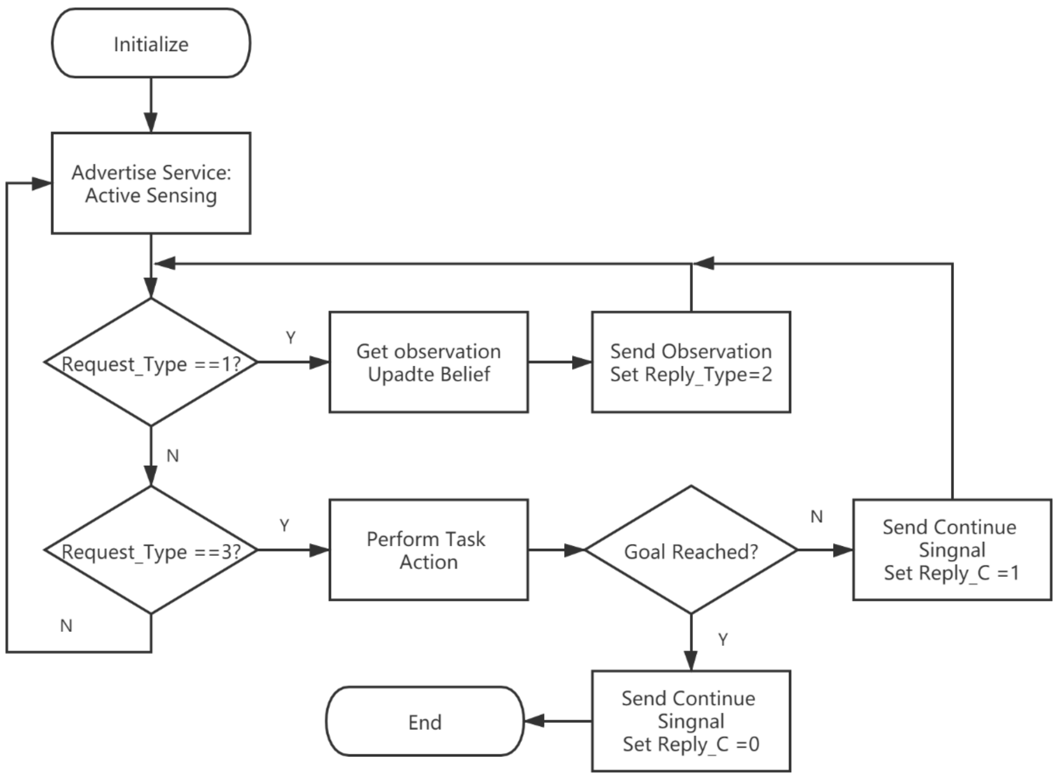
Initialize
Advertise Service: Active Sensing
Request_Type ==1?
Y
Get observation Upadte Belief
Send Observation Set Reply_Type=2
N
Request_Type ==3?
Y
Perform Task Action
Goal Reached?
N
Send Continue Singnal Set Reply_C =1
N
Y
End
Send Continue Singnal Set Reply_C =0

Fig 15 The Process of the local planner of Client-Server Model

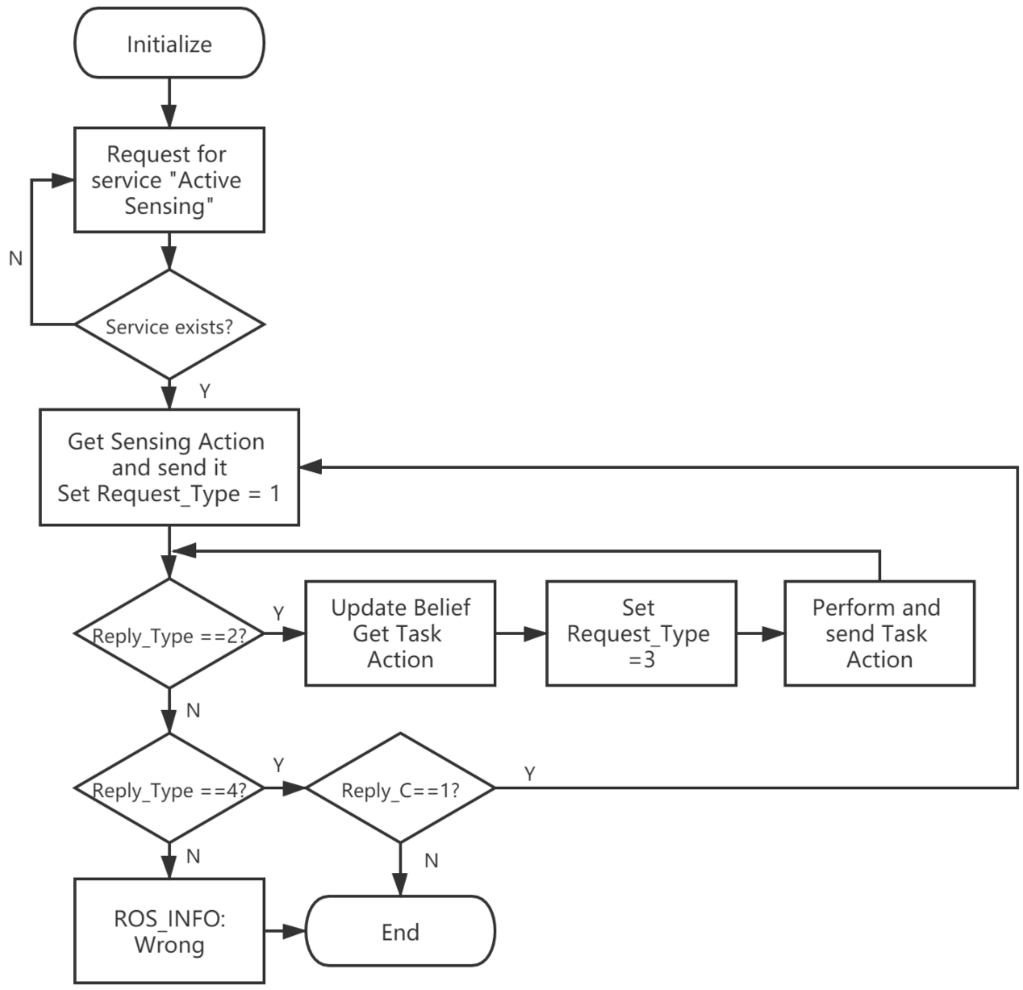
Initialize
Request for service "Active Sensing"
N
Service exists?
Y
Get Sensing Action and send it Set Request_Type = 1
Reply_Type ==2?
Y
Update Belief Get Task Action
Set Request_Type =3
Perform and send Task Action
N
Reply_Type ==4?
Y
Reply_C==1?
Y
N
N
ROS_INFO: Wrong
End

Fig 16 The process of the cloud planner of Client-Server model

Fig 17 shows the client-server model in ROS. Both local and cloud planners are registered when advertising or calling for a service. Data are transmitted in the type of request or reply of a service file which is related to the service "Active sensing".

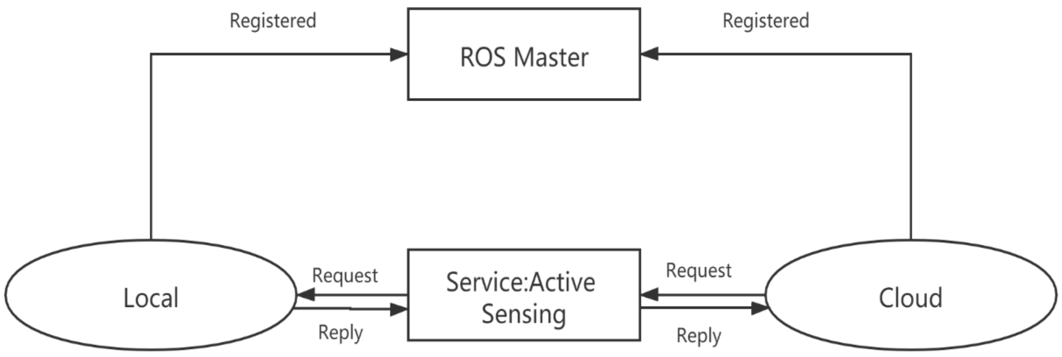

Fig 17 The Client-Server model in ROS

4.5 Revised Active Sensing Algorithm based on OpenMP

Particle filters are used to model belief propagation in continuous state spaces in the implementation of the Minimum Task-Action Entropy Active Sensing algorithm. However, the action-entropy active sensing algorithm has time complexity $O(dn_vMNlogN)$, in which d is the dimension of the particles, n_v is the number of sensing actions, M is the time that Monte Carlo simulation is performed, N is the number of particles [42]. When the number of the particles increases, the time cost becomes unacceptable. It is found that there are two nested loops in the algorithm which makes it possible for the application of parallel computing. However, the first loop is the for-each loop and the inner loop stands for the Monte Carlo simulation. If parallel computing is applied, the sequence will be random and may affect the result. For example, the particles in the second sensing action can interact with the results calculated in the first sensing action. In this situation, the lowest conditional entropy is incorrect.

Algorithm 2 OpenMP based Minimum Task-Action Entropy Active Sensing

```
1: procedure PARALLELGETSENSINGACTION($\pi, P_x$)
2:   $h_v array = zeros[Amount\ \ of\ \ sensing\ \ actions]$
Parallel computing part
3:   for j = 1 ,…, Number of Sensing actions do
4:        for $i = 1, \dots, M$ do
5:            sample $x$ from $P_x$
6:            sample $z$ from $p(z|v.at(j), x)$
7:            $P'_x = MeasurementUpdate(P_x, v.at(j), z)$
8:            $P_u = \pi(P'_x)$
9:            $h_v array[j] += \hat{h}P_u(u)$
10:       end for
11:  end for
End Parallel computing
12:  return $argmin_v h_v array$
13:  end procedure
```

To solve this problem, we change the algorithm as Algorithm 2 shows. We define an array which has the size of the number of the sensing actions. For each sensing action, the sum of the cumulative entropy will be stored in the related position in the array. When the calculation finished, all the cumulative entropy will be compared to find out the minimum.

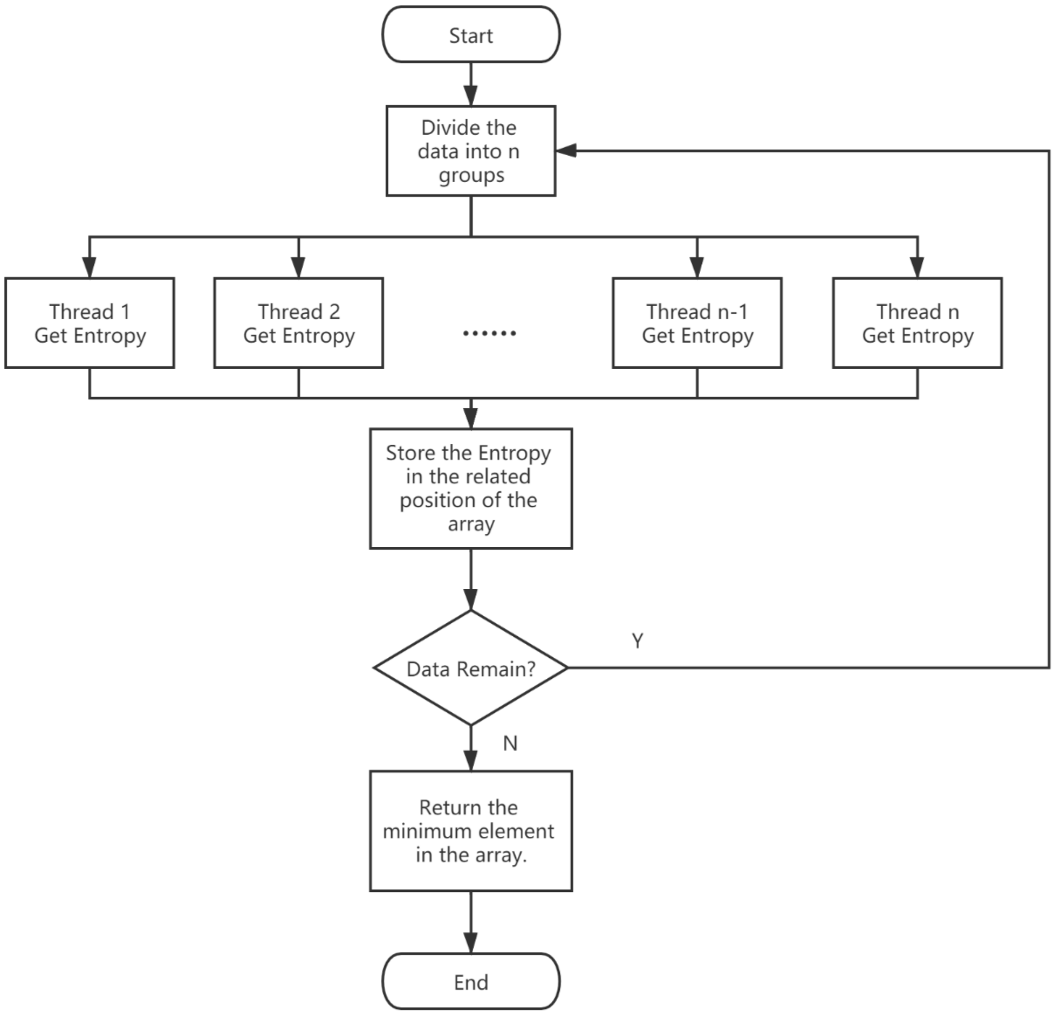
Start
Divide the data into n groups
Thread 1 Get Entropy
Thread 2 Get Entropy
......
Thread n-1 Get Entropy
Thread n Get Entropy
Store the Entropy in the related position of the array
Data Remain?
Y
N
Return the minimum element in the array.
End

Fig 18 The Process of the revised algorithm

When the procedure begins, all the sensing actions and observations are combined as a matrix unit which has the coordinate of *i* and *j*. *j* stands for the serial number of the current sensing action and *i* stands for the serial number of the current observation. Then the sample state will be generated from the particles. The sample observation will be generated from the Monte Carlo simulation by providing the current sensing action as line 5 and 6 shows. After update the weights by the current sensing action and observation. The cumulative entropy will be calculated and stored in the related position in the array of entropies. According to OpenMP's fork-slave structure, each unit in the matrix will be executed in parallel and independently. When the calculation is done, they will combine to the final cumulative entropy in the related position of the array. After the array of the entropy is filled with the calculating result, the minimum element in the array will be selected as the lowest conditional entropy. The related sensing action will be returned as the result.

Chapter 5 Simulation and Results

In this chapter, the peg-in-hole problem will be presented for simulation. Greigarn compares the action-entropy active sensing algorithm with state-entropy active sensing algorithm in decision making for continuous problems by simulating the peg-in-hole problem [1]. The revised algorithm is still designed to perform in the continuous system so that we choose this simulation. Besides the problem, we will compare the time cost of the original algorithm and the revised algorithm. The revised model will also be tested in LAN and the cloud. The relationship between the number of the thread in parallel and the speedup will also be recorded.

5.1 Peg-in-Hole Problem

Lots of robots are now dealing with the task in which they need to insert a peg or the screw into the hole [43]. In most cases, if the uncertainty of the location of the peg regards to the hole is reduced by manipulation and force feedback, the problem is solved [44]. The problem is suitable for the simulation in continuous system. The state space of the problem can be divided in subspaces based on the stage of the execution. The peg will move towards the hole when it is not in front of the hole. The position of the peg relative to the hole's axis is the subspace which is similar to the position of the right lane's car in the problem mentioned in Chapter 1. When the peg is in front of the whole, it will change its direction and insert into the hole. A 2D version of the problem will be performed in our research.

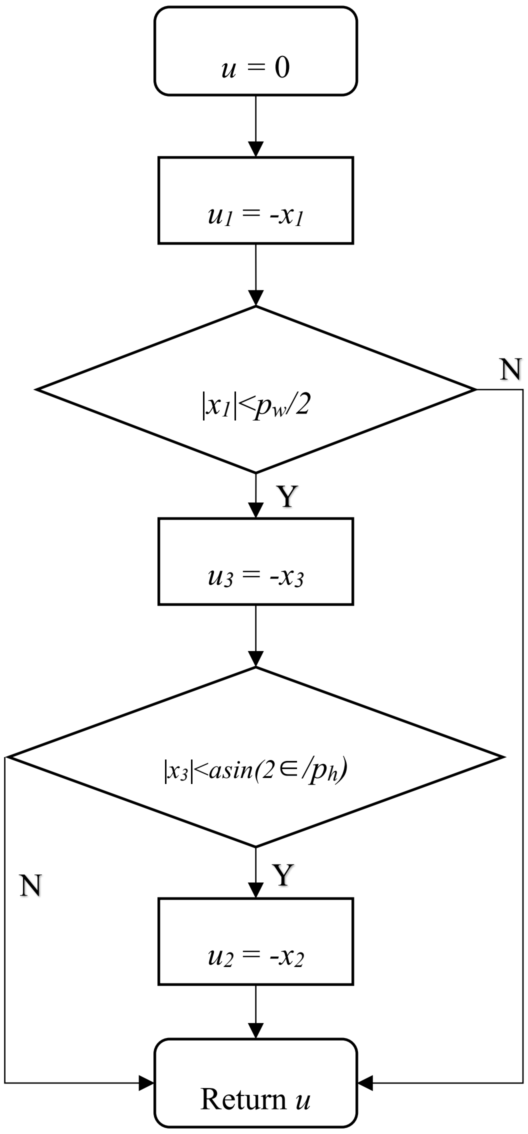

Fig 19 The Peg in Hole Problem

Fig 19 shows the policy of the Peg in Hole problem. x stands for the location of the peg, p_w and p_h represent the width and height of the peg. The action u is initialized to 0 at first. The state of the peg at time t will be shown by $x_t \in R^3$, which contains the x and y coordinates and the direction of the peg towards the destination hole. Then the action u will be set to $-x_1$ which means the direction to the hole on the x-axis. Then the policy will find out whether the peg's estimated location is less than half of peg's width to the hole along the x-axis. If the estimated location is not far away from the destination, the action will be returned to tell the peg to move to that location. When the peg reaches the location, the policy will generate the new action and redo the procedure. If the first estimated location is in the range, then u will be set to $–x_3$ which contains the direction information to the origin such as the angles. Then the policy will check whether the peg is too tilted or not. If the second estimated location is too tilted, the action will be returned and reorient the peg. If the peg is in good position, then u_2 will be set to $–x_2$ which means let the peg drop into the hole. Then the final action is generated which can lead the peg to the hole.

The hole is supposed to be large enough for the peg to drop and located in the origin of both x and y axis. The initial belief is generated from Gaussian distribution $b_0 = N(\mu_x, \sigma_x)$. The model can be described as $x_t = x_{t-1} + u_t \min\left(1, \frac{\alpha}{\|u_t\|}\right) + n_u$, where x_t stands for the coordinate and direction of the peg at time t, u_t stands for the action that controls and change the direction of the peg at time t. n_u is the motion noise which is assumed to follow a Gaussian distribution $N(0, \sigma_u)$. α is the step size of the simulation. The sensing action $v_t \in \{1,2,3\}$ selects the direction of the state space that should be observed. Suppose $x_{i,t}$ is the ith element of x_t, then the observation as known as the measurement can be written as $z_t = x_{v_t,t} + n_v$, where n_v is the measurement noise which assumed to follow a Gaussian distribution $N(0, \sigma_v)$

According to Greigarn,'s settings, the width and height of the peg are set to 1 and 2. The tolerance of the hole is 0.1. The parameters of the Gaussian Distribution for the original internal belief is $\mu_x = [4,2,0]^T$, $\sigma_x^2 = diag([2,1,1.57])$. The parameters for the motion noise and the measurement noise distribution are $\sigma_u^2 = diag([0.01,0.01,0.01])$, and $\sigma_v^2 = 0.001$. The step size α is set to 0.1.

The simulation will be repeated for 1000 times. If the peg drops into the hole or the number of the step reaches the maximum number which is set to 500, the individual trial will be terminated. If the peg fails to insert into the hole for 1000 times, the task fails. All the parameters are stored in a YAML file for configuration.

model	state_size	3
	peg_width	1
	peg_height	2
	hole_tolerance	0.1
	init_mean	[4,2,0]
	init_cov	[2,1,1.57]
	montion_cov	[0.01,0.01,0.01]
	sensing_cov	0.001
	collision_tol	0.001

Table 1 Configurations in YAML file

5.2 The Failure of the Publish-Subscribe model

In the simulation, we encountered the situation that ROS failed to run the callback function in which sensing actions, observations and task actions were generated. After the first few communications between two sides. The publisher will stay waiting for the subscriber even if the subscriber has already subscribed to the topic. The problem happens on both sides when there is a new subscription to the new topic after publishing the former one. For example, when the local planner finish subscribing form the topic "Waiting for Sensing Action Request", it will get the sensing action from the cloud. After the local planner finish getting the observation from the sensing action, it will publish the topic "Waiting for Observation Request". The cloud should keep looking for the topic on ROS master before the connection is established. However, even the local has already published the topic "Waiting for Observation Request", the cloud planner still cannot find it. Then the system is stuck in the loop. The cloud is keep waiting for the topic while the local is waiting for the subscriber.

We solved the problem by adding a sleeping duration before each spin of ROS. In ROS, socket connections for any topics is monitored. All the callback functions will be stored in a queue and follows the "first in first out" rule. ROS spin is the command that processes the callback functions. ROS will skip the spin procedure if there is no callback function in the queue. Establishing the connection between the publisher and subscriber needs a small period of time. If the connection is not set up, the subscriber cannot get the callback from the publisher, the callback function queue will be empty.

In the proposed publish-subscribe model, we set the subscriber to subscribe to the topic consistently in a loop without any pause. However, it needs time to establish the connection. The cloud planner subscribes to the topic “Waiting For Observation Request” immediately after it sends the sensing action to the local planner. In the meantime, the local planner is getting the observation. The topic is not existed at that time so that the cloud planner will do the subscription continuously without any pause until it finds the topic. When the local planner publishes the topic “Waiting for Observation Request”, the cloud planner may find it. Then both sides are trying to set up a connection. Because the cloud planner is sending the subscription request without any pause, the default waiting time for ROS master to response is limited. However, the time for setting up the connection is longer that the cloud planner’s default waiting time. The cloud planner will ask the ROS master to redo the connection although the connection is nearly finished. In this case, the connection between the local planner and cloud planner will never established. Therefore, the cloud planner cannot get any observation from the local planner.

When sleeping duration is added before ROS spinning, the simulation runs successfully. In order to shorten the communication time, the sleep duration should be set as short as possible. However, if we set the sleep duration time less than 100ms, the problem comes out again. Thus we find out that 100ms was necessary for setting up a publish-subscribe connection. In the proposed model, there are 4 connections in each single process so that it needs to take at least 400ms for the connection establishment. Besides the establishment, there's also communication time cost for sending and receiving the message which takes 70ms per communication. The model can only shorten the time if the number of the particles is extremely huge, the computing time saves much more time than the connection time cost. However, in general, applying the publish-subscribe model to the shorten the running time of the original algorithm is unrealistic.

In the Client-Server model, there is only one connection at the beginning. There is only one service in the communication so that all the data such as sensing action, observation and task action are stored in the same srv message file. There is no need to get connection every time when requesting for different types of data. The only time cost is the communication between both sides, in other words, the cloud planner and the local planner.

5.3 Simulation Results

The simulation is performed on a AWS EC2 instance running Ubuntu 16.04 with type of c5.9xlarge which has 18CPUs and 36 cores. The memory is 72GB. The revised algorithm and experiments are written in C++. OpenMP 3.0 is used to perform parallel computing. The algorithm is compared against a baseline method in which sensing actions are completely chosen randomly following by Gaussian random numbers.

Fig 20 shows the execution time of each action in the model. According to the percentage of the time each task takes, it is necessary to put the active sensing part, which takes 85% of the total time, on the cloud to have shorter running time.

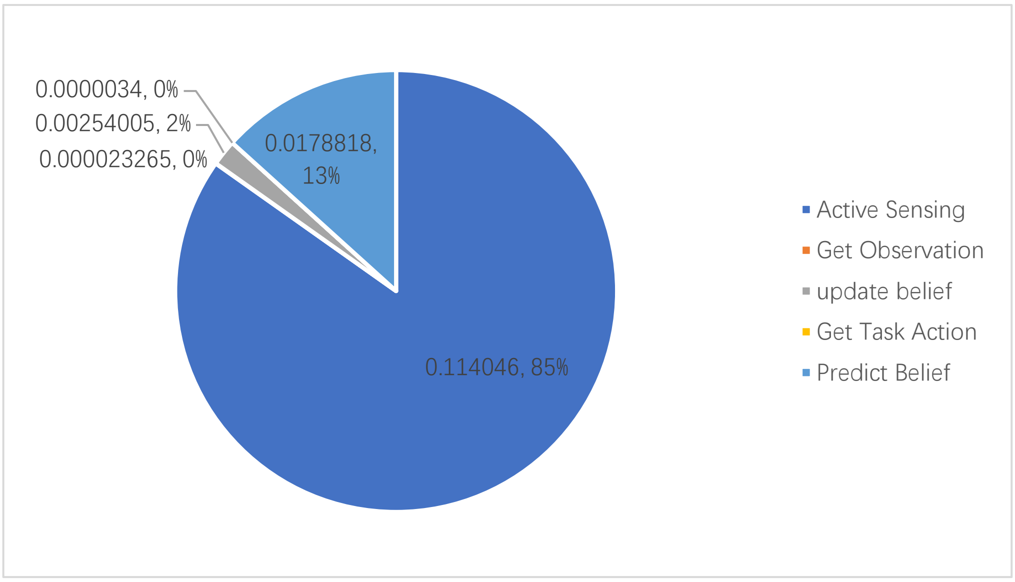

Fig 20 Time spent on each action in seconds, and percentage time of each action

Fig 21 shows the relationship between the execution time and number of particles of the original algorithm with a single core with single thread. We can find out that the execution time increases obviously as the number of the particle increases. The execution time become unacceptable when the number of observations the robot take is 100. The robot does not have enough time to perform the action especially when the robot is in a high-speed movement.

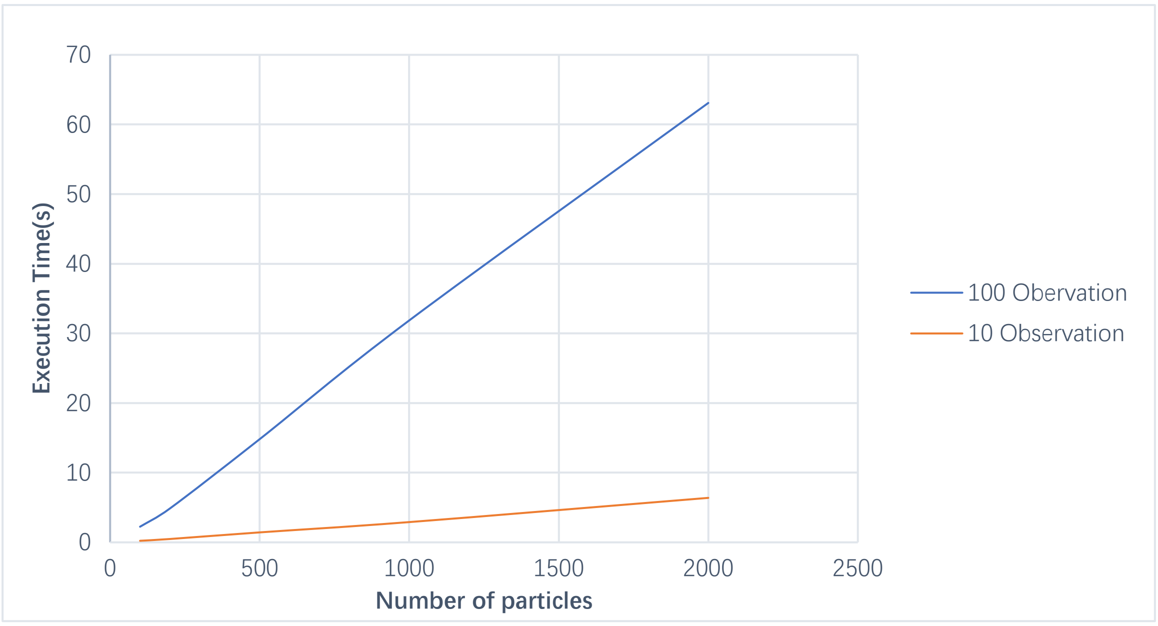

Fig 21 Relationship between number of particles and execution time

To find out the improved performance with AWS, we run the both original and revised algorithm on the cloud. Time for get sensing actions is recorded. Fig 22 compares the execution time of the original algorithm with OpenMP-based algorithm with the number of particles set to 100. We find that the original algorithm has some improvement when the number of cores increase. This phenomenon will be explained in Chapter 6.

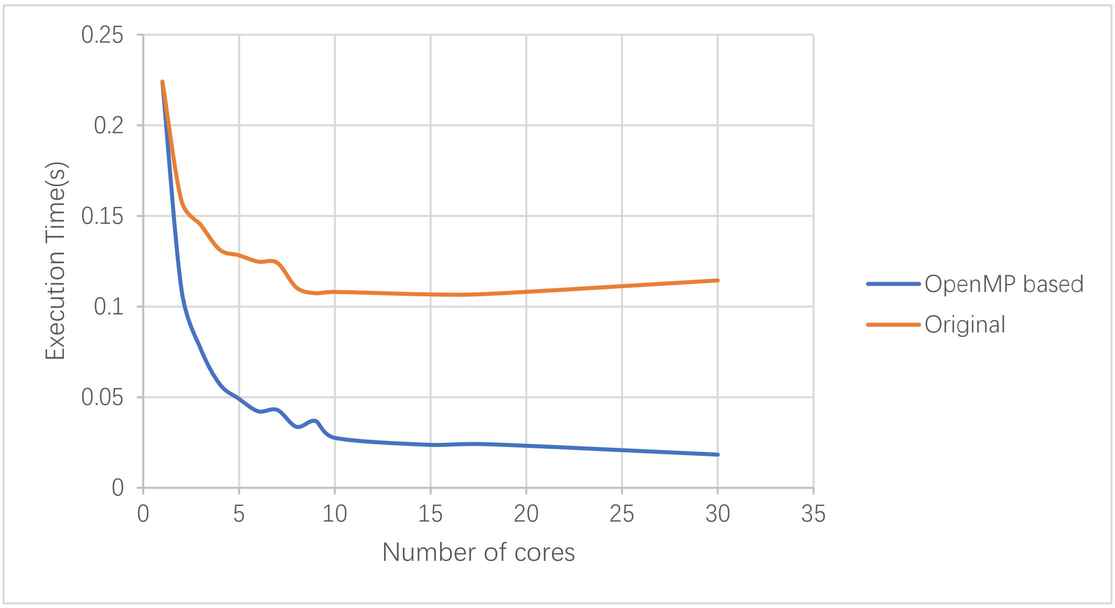

Fig 22 Execution time of the original algorithm and OpenMP based algorithm

To find out the relationship between the time cost of single iteration and the total, we recorded the execution time for one iteration in the loop for both original and OpenMP based algorithm in Fig 23.

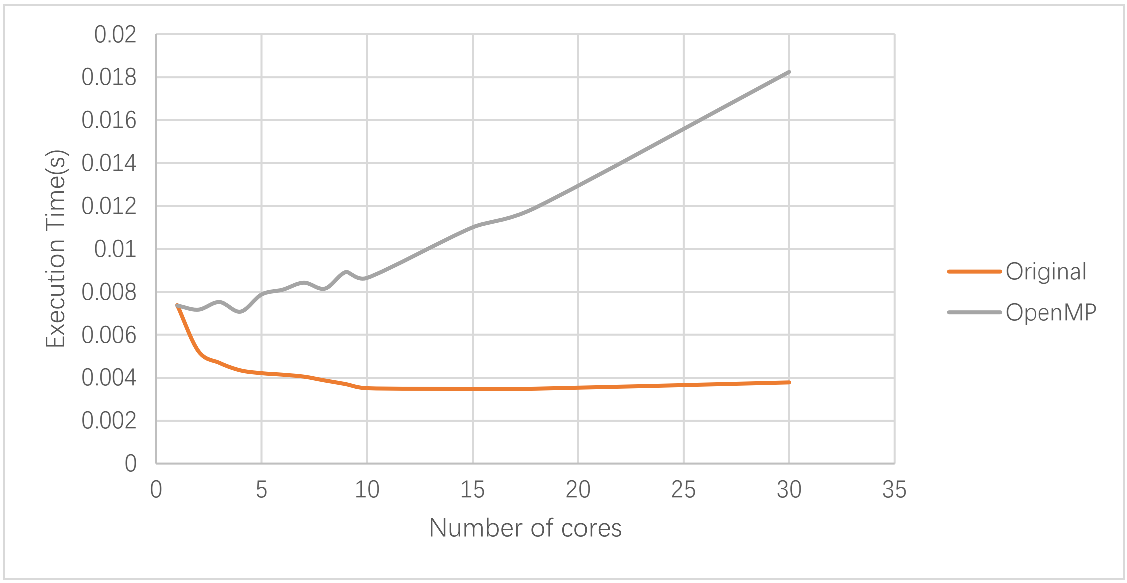

Fig 23 Execution time of the single iteration in original algorithm and OpenMP based algorithm

As we can see in Fig 23, the time cost for single iteration increases when more cores are applied. To find out whether AWS's architecture has an impact on it, we perform the simulation on our local desktop and compare the result with the cloud. The environment of the local desktop is built on Intel i-7 7700k 3.2Ghz with 6 cores and 8GB memory. Fig 24-26 shows the result. Local stands for the local desktop. AWS stands for the EC2 instance.

Fig 24 shows the execution time of the single iteration in the loop when different cores on AWS and local machine is applied. In the figure, local stands for the local machine. The number of particles is fixed to 100. The execution time of single iteration of local machine is more stable than AWS when the number of cores is less than 6. However, when the number of cores exceeds 6, the execution time of single iteration on the local machine increases obviously. It is caused by Hyper-Threading technology which will be discussed in chapter 6. Fig 25,26 shows occurrence of the super linear speedup when the number of cores is set to 2. It will also be discussed in Chapter 6.

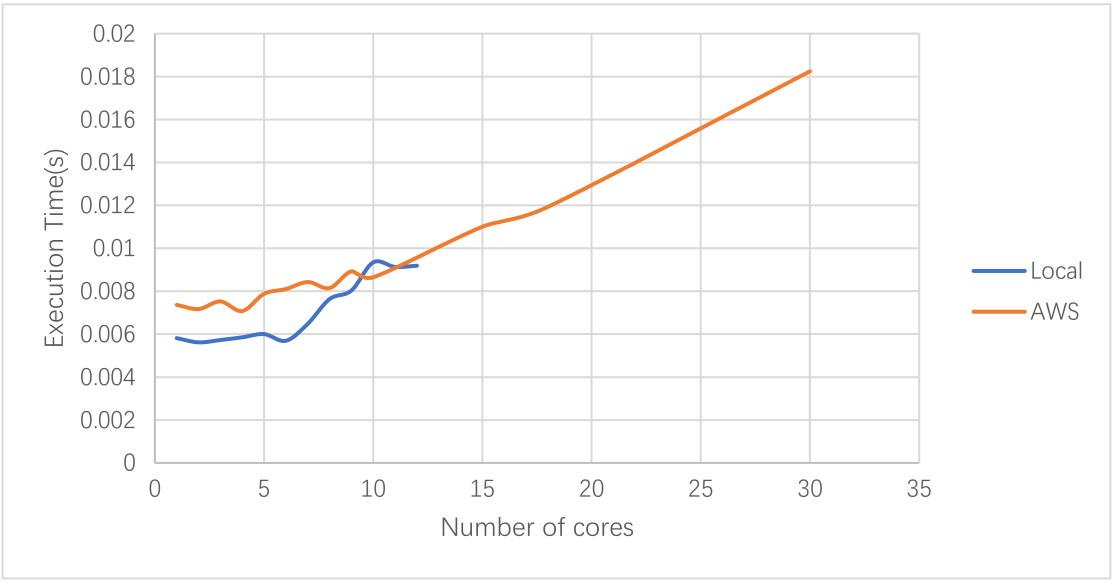

Fig 24 Execution time of single iteration with different cores

Fig 25 shows the relationship between the speedup and the number of cores. Particles are set to 100.

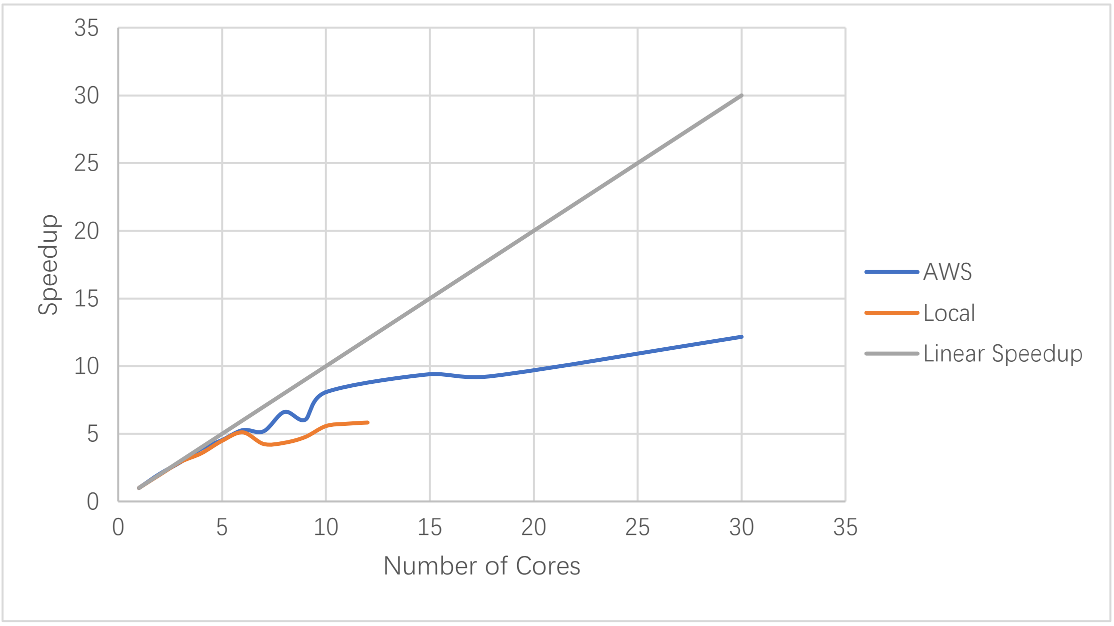

Fig 25 Speedup values with different number of cores

Fig 26 shows the efficiency when different number of cores are applied on both local and the cloud.

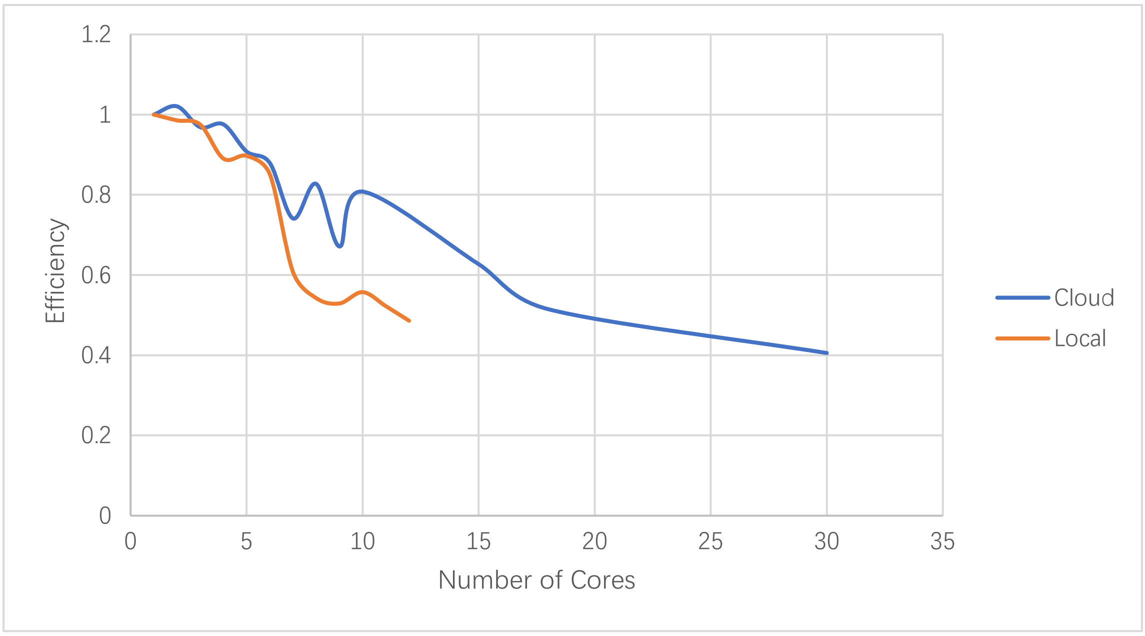

Fig 26 Efficiency with different cores

In order to find out the best performance of the revised algorithm and communication model, we tried different task scheduling method on OpenMP. Fig 27-29 show the performance of each scheduling method. However, it seems that there's no obvious difference on the performance.

Fig 27 shows the execution time of different scheduling method in OpenMP.

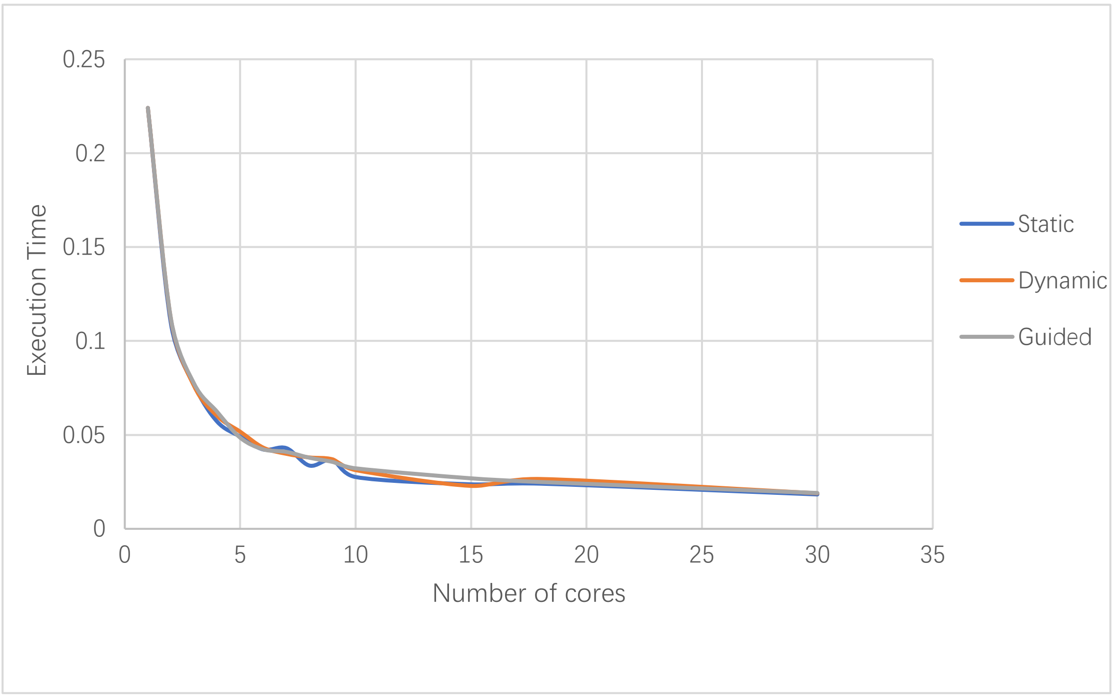

Fig 27 Execution time with different scheduling method

Fig 28 shows the speedup of different scheduling method in OpenMP

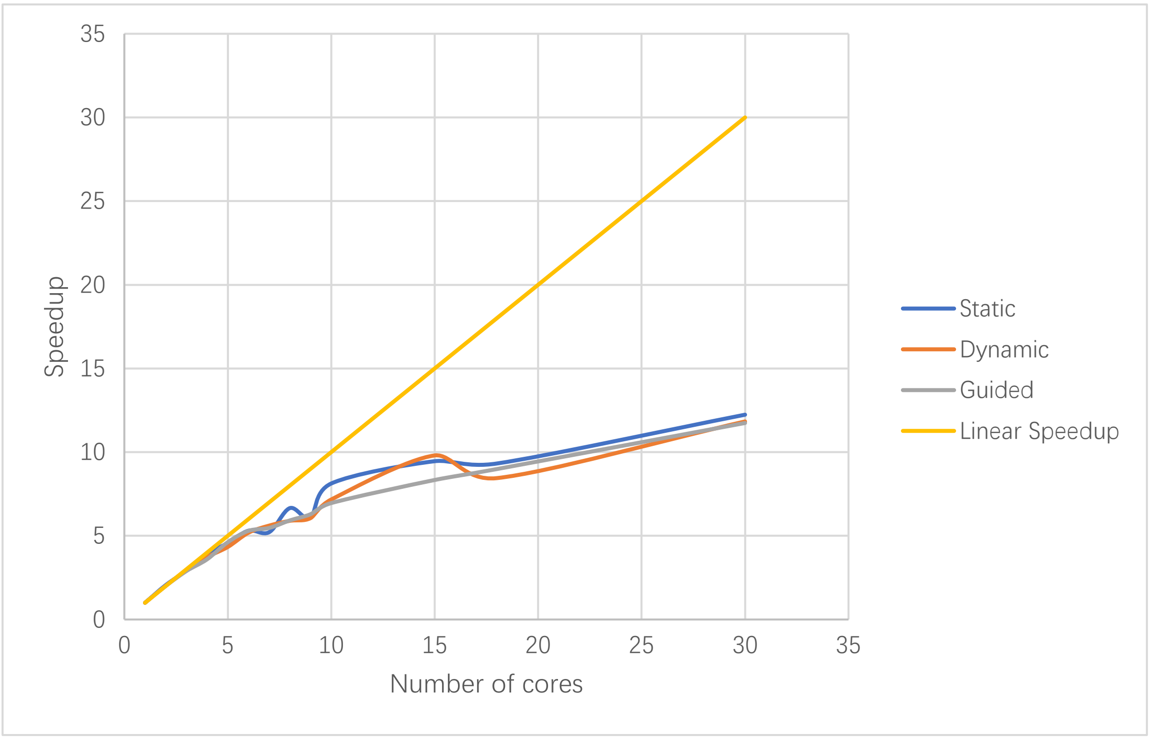

Fig 28 Speedup with Different Scheduling Method

Fig 29 shows the efficiency when different scheduling method is applied.

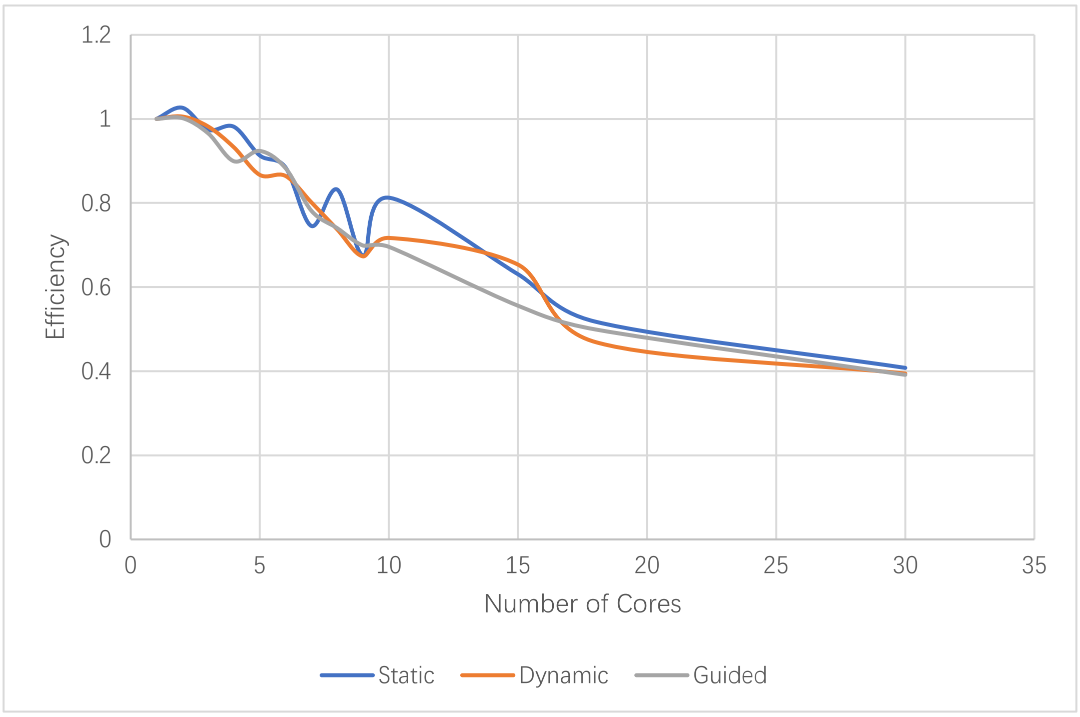

Fig 29 Efficiency with different scheduling method

According to the result we get, static scheduling is selected for the rest of the simulations. To find out whether the number of threads can affect the performance when the core is fixed. We do the simulation in which different number of threads are applied when the number of cores is set to 10. The result is shown in Fig 30 and 31.

Fig 30 shows the execution time under different number of threads when the total CPU is set to 10.

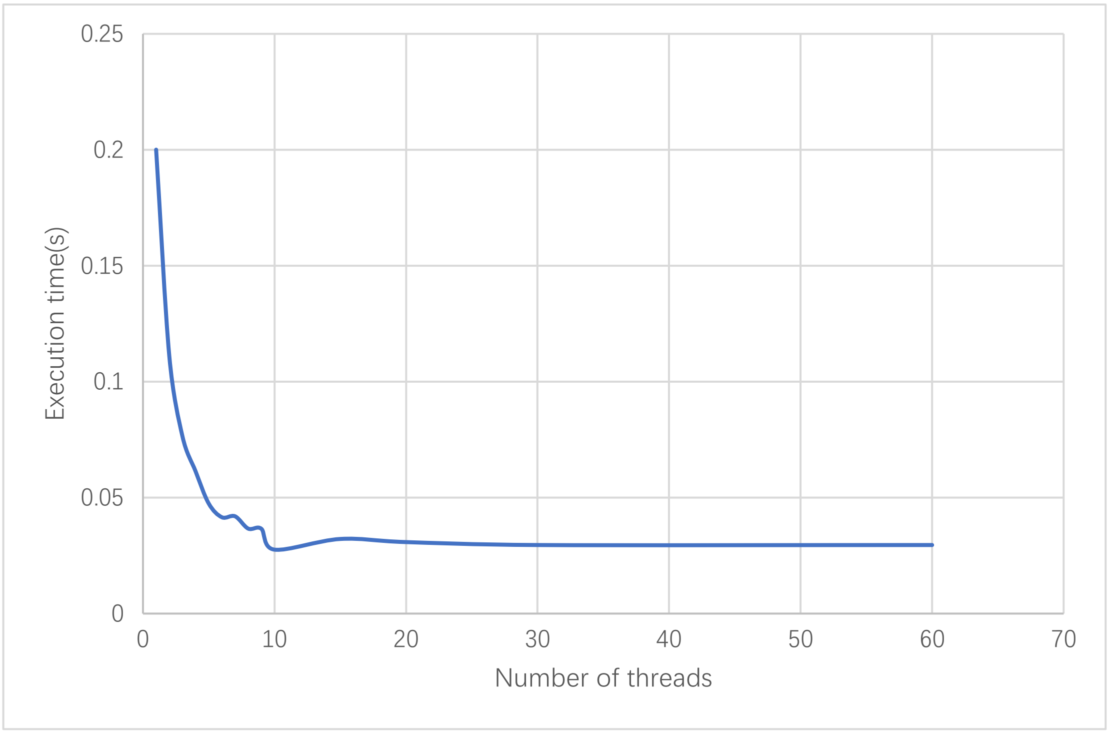

Fig 30 Execution time with different number of threads

Fig 31 shows the speedup in the execution of different number of threads when the CPU number is set to 10.

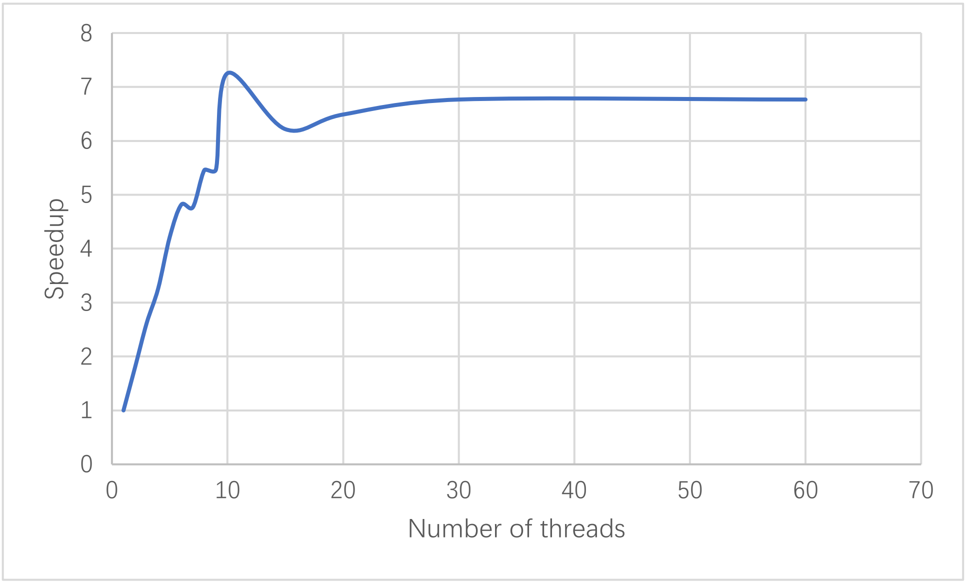

Fig 31 Speedup with different number of threads

Finally, we get the execution time of the total process in Table 3 when the number of cores and threads are set to 10. Static scheduling is chosen for OpenMP.

Table 3 shows the total execution time of different communication model

Number of particles	Single CPU	LAN Client-Server	Cloud Client-Server
100	263	81	171
200	564	228	213
500	1682	530	501
1000	3422	785	1061
2000	7510	1910	1873

Table 2 Total execution time of different communication model (ms)

Chapter 6 Discussion

This chapter will analyze the simulation results and explain the features in the experiment that were not expected. The super linear speedup will also be taken into discussion.

6.1 The impact when multi cores are applied

Fig 22 and Fig 23 shows that the performance gets better when multi-core is applied. For the original sequential algorithm, even simply multi-core processing in the underlying system, the time cost of one iteration of the loop is decreased by one-half.

The reason is because, even though the algorithm is sequential, additional cores are applied to deal with other system processes, and therefore, there are more resources for the algorithm to run.

Another explanation is that, we use the FLANN package, which stands for fast library for approximate nearest neighbors, in our algorithm to search the nearest neighbors to estimate the entropy for each particle. The searching algorithm in FLANN has a parallel structure which can work with OpenMP. In the makefile of the FLANN library there's a default setting for enabling OpenMP. The original algorithm was executed when OpenMP was enabled in FLANN. Therefore, the functions in FLANN can be executed in parallel when multicores are applied. Despite of FLANN, most parts of the original algorithm is not parallelized, and so the speedup can only approximately reach 2. Every single iteration in the loop is executed consecutively so that the speedup of the total process can also be estimated by the speedup of the single iteration. When the performance reaches the maximum, the impact of increasing the number of cores is not obvious. For example, 10 cores have the best performance in the simulation, when we increase the number of cores from 10, the time cost for single iteration and total process do not change much.

For the revised algorithm, the performance of the total process is much better than the original algorithm when multi-core is applied. However, Fig 24 shows that the time cost for a single iteration increases when more cores are applied on AWS. Although the time cost of single iteration increases, since the single iterations are executed in parallel, the performance is still better than the original algorithm.

One possible explanation for this phenomena is that the synchronization and scheduling overheads increases when multi-core is applied. Bull et al have already measured the overhead and the time cost of synchronization by setting a set of benchmarks in 1999 [45]. They had the conclusion that the overhead increases rapidly when the number of threads increase. Oliver et al also performed experiments evaluating the performance of OpenMP 3.0 [46]. Their results illustrated that the speedup of the parallel-based algorithm would decrease when the number of cores increased to a large amount.

The other explanation is that distribution of CPUs on AWS is not uniform, which leads to the different performances of the CPUs. The CPU on the EC2 instance are not the same and may have different performances. To support this explanation, the result is shown in Fig 25. As we can see, the local desktop has better performance than the AWS, which means the performance of CPU on AWS is worse than Intel i-7 7700k. In addition, the execution time of the single iteration on the local machine does not change much when the core count is less than 6 which means each core on the local machine has the same performance. The performance of CPUs on AWS is different. Le et al compared the performance of the Top 5 AWS EC2 instances and discussed about the performance related issues [47]. According to their research, AWS's hardware may be opaque because the system administrators and developers cannot examine any aspect of the hardware that has been allocated to them. Also, AWS offers the access to the hardware in their data center, however the hardware as well as the CPU we use is not own by us. Other customers can also get access to the hardware, which means we are sharing those cloud resources. When we perform our simulation on the cloud, perhaps there are other customers who run their applications on the same CPU. In this situation, we cannot get 100% performance form the EC2 instance. The research also mentioned other factors such as ECU mismatch, stolen CPU, maintenance and service interruptions that may also affect the performance of the EC2 instance.

The third explanation is the Hyper-Threading in Intel Multi-Core processors. HT-technology is designed to simulate the presence of another processor which means there is an extra virtual core that runs with the physical core on the CPU. One physical CPU core with HT-technology is regarded as two logical cores by the operating system. The computing resources of the physical processor is shared to work as two logical cores [49]. This can improve the CPU's performance in some situation, for example, if one core is stuck, the other core can borrow its computing resources. Tian et al proposed a compiler that works with OpenMP and Hyper-Threading [50]. Their experiment runs on a single core CPU with Hyper-Threading, and only gets the maximum speed up of 1.6 (which is less than the desired speed-up of 2). Tau et al also test the performance of their algorithm by running on a Linux cluster and get the result that CPU with hyper-threading can gain 30% performance [51]. Deborah et al also get the results that the gain up of Hypter-Threading is 30% [52]. Saini et al find out that Hyper-Threading cannot have 200% performance because it increases competition for resources in the memory hierarchy [53]. Hyper-threading performance is also affected by the increased communication pressure.

As fig 25 shows, when the number of cores on the local desktop exceeds 6, the execution time of a single iteration increases suddenly. The local desktop has 6 physical cores while Ubuntu 16.04 shows that there are 12 cores total. This is because we enabled the Hyper-Threading on our processor so that there are 6 more virtual core.

However, the virtual core does not have the same performance as the physical core. It only uses the resources that is available to make fully use of the processor's resources. In order to gain the full benefit from Hyper-Threading, the task for each thread should be different enough so that the conflict of the computing resource will be small [48]. However, in our simulation, all the parallel units perform the same computation, and so the work for the physical core and the virtual core are similar. Conflict may occur in the execution so that the virtual core's performance is not as expected. CPU with HT-technology could have better performance than a CPU without it but not twice.

6.2 Different Scheduling and Super Linear Speedup

In order to find out the best performance of the revised algorithm and communication model, we tried different task scheduling method on OpenMP. Fig 27 shows the performance of each scheduling method. However, it seems that there's no obvious difference on the performance. Ayguadé et al also found that the performance of different scheduling method in OpenMP was similar [54]. The difference range is within 3%.

Speedup is used to evaluate the performance when different number of cores is applied. It is defined as the ratio of the time cost of the sequential algorithm to the time cost of the parallel algorithm to solve the same problem on a specific number of processors. The definition is in formula (4):

$$\mathrm{S} = \frac{T_S}{T_P} \tag{4}$$

T_S stands for the time spent on the sequential algorithm and T_P stands for the time spent on the parallel algorithm. The number of the cores is p.

In our simulation, T_s is the execution time when a single core is applied to the original algorithm. T_P is the execution time when p cores are applied to the OpenMP based algorithm.

Efficiency is also another factor that can evaluate the performance and measure the fraction of time for which a processor is utilized when different number of cores are applied. It is defined as the ratio of speedup to the number of cores in formula (5):

$$\mathrm{E} = \frac{S}{p} = \frac{T_S}{pT_P} \tag{5}$$

S stands for the speedup and p is the number of the cores.

According to the result in Fig 26,27,29,30, we find out that when the number of cores increases, the efficiency drops because more communication is needed between the threads and the monitoring and scheduling of the threads take more time. There is a special case when the core number is set to 2, the speedup is greater than 2. This situation is called super linear speedup.

Super linear speedup is unexpected because the speedup is limited with the number of cores or processors according to Gustafson’s Law [55]. Many researchers encountered super linear speedup in their experiment but only mentioned it as a side effect [56]. Ciamulski et al proposed an explanation for the super linear speedup. They believed that there was greater amount of cache memory in the parallel execution compared to sequential [57].

Another explanation is that the total amount of data that needs to be searched in parallel computing is less than the amount in the sequential execution. In other words, the execution time of the single iteration decreases. The super linear speedup only occurs when the number of cores is set to 2 because the communication time between 2 threads is shorter than the time saved by accessing the cache. Ristov et al also gives explains the super linear speedup on the cloud [58]. The architecture of the cloud affects the performance of CPU and leads to super linear speedup.

In fact, the cache only saves a short time when accessing the data. However, the time of thread communication cannot be saved. When the number of cores increases, the time of thread communication costs more than the time the cache saves. Therefore, the efficiency is less than 1 when the number of cores exceeds 2.

6.3 Time Formula Derivation

As Fig 23 shows, we found that the curve is similar to an inverse function. To find out the relationship between the execution time and the number of cores, we draw the trend-line as Fig 32 shows.

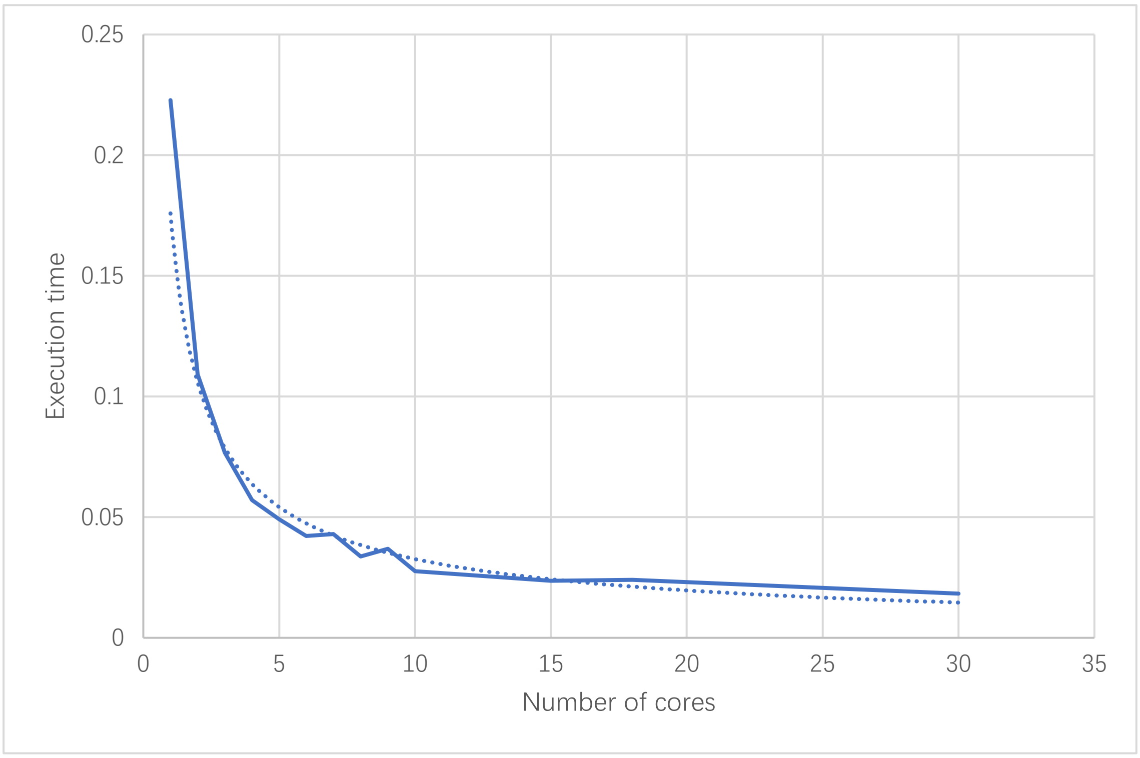

Fig 32 Trendline of the OpenMP based algorithm

According to the trend line in Fig 32, we find that a power function can be used to describe the relationship between the number of cores and the total execution time. In view of the code structure of the proposed algorithm, there are some initializations before the parallel part so that a fixed time should be added in the formula. We proposed the formula (6) as below:

$$t = \frac{\beta}{x^{\gamma}} + \alpha \tag{6}$$

x stands for the number of the cores, α is the fixed time in the function, $\frac{\beta}{x^{\gamma}}$ is the time cost of the parts in parallel. γ should be less than 1 because the time cost of the single iteration increases when the number of core increases. The parameter values are interpolated from the simulation run times when the number of particles is set to 100. The formula with parameters is shown as below:

$$t = \frac{0.196105}{x^{0.94}} + 0.006$$

According to the parameters we got, we can find out that the fixed time in the function is 6ms.

6.4 The impact of different threads when core is fixed

In our simulation, there are 3 sensing actions, each sensing action has 10 observations. Static scheduling is used for parallel computing. In order to decrease the time for thread communication, the total number of the tasks should be divisible by the number of cores. We find out that 10 core is the most suitable case for the simulation when the particle is set to 100.

Some research discusses multi-core and multi-thread performance [59]. In order to find out the best performance setting, we test the performance on different number of threads when the core is set to 10. As Fig 32 and Fig 33 shows, the speedup reaches the peak when the number of thread equals to the number of cores. When the number of thread is greater than the number of cores, a redundancy in threads occurs which makes the communication between the thread take more time.

Table 3 shows the time cost of the entire process. The number of cores is set to 10 and the number of the threads is the same. The speedup can exceed 7. We find that when the number of particles increases, the cloud-based model will have better performance as the communication time overhead is smaller.

Chapter 7 Conclusion and Future Work

In this work, we have modified the original action entropy active sensing model. We set the computation part on the cloud by using the service of AWS. The communication between the cloud and local is made under the structure of ROS. After the failure in the publish and subscribe the model, the communication succeeds by applying the ROS client and server model. The connection is set up with a tunnel between the AWS VPC and the local LAN with OpenVPN. To improve the performance of the action entropy active sensing algorithm, we applied parallel computing to it with OpenMP. Simulations are designed to find out the configuration for the best performance. Factors that have impact on the performance of the parallel computing such as Hyper-threading and super linear speedup are also discussed. Finally, we derive the time cost formula and find out the best configuration.

Future work contains three different directions. The first direction is the physical robot implementation. In our project, we only implement robot simulations. When the algorithm is applied to real word problem, we need to consider the connection, the sensors on the real robot and the time for the robot to perform the action.

The second direction is improving the performance of connection. In our simulation, we did not calculate the packet loss and delay of each port and socket. Different protocols can have different performance when communicating between the local and the cloud. If the best protocol is selected and the packets are fully received, the communication time can be reduced which helps the improvement of the performance of the revised model.

The third direction is to achieve better speedup by applying GPU to the original algorithm. As we mentioned in Chapter 3, there is research that implement both OpenMP parallel computing and GPU based computing. Li et al has already implemented a parallel algorithm based on GPU in the AWS Cloud and reaches the speedup up to 80 compared to the original algorithm on CPU [60]. Guo et al got the speedup of 426 when CUDA was implemented [39]. Sun et al got the speedup of 100 in their parallel GPU based algorithm [41]. Compute Unified Device Architectures is a parallel computing plat form and application programming interface that aims to solve the parallel computing problem. When GPU is applied, the data will be divided among different Kernel functions. The CPU only needs to wait for the results from the GPU. The speedup of action entropy active sensing algorithm may be much better than the result from CPU.

Reference

[1] Greigarn, Tipakorn, Michael S. Branicky, and M. Cenk Çavuşoğlu. "Task-Oriented Active Sensing via Action Entropy Minimization." *IEEE Access* 7 (2019): 135413-135426.

[2] Kehoe, Ben, et al. "A survey of research on cloud robotics and automation." IEEE Transactions on automation science and engineering 12.2 (2015): 398-409.

[3] P. Mell and T. Grance, The NIST definition of cloud computing, National Institute of Standards and Technology, 2009.

[4] National Science Foundation, Enabling a new future for cloud computing, 2014. [Online]. Available: http://nsf.gov/news/news summ.jsp? cntn id=132377

[5] What is RoboEarth? [Online]. Available: http://www.roboearth.org/what-is-roboearth

[6] M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D.Gálvez-López, K. Häussermann, R. Janssen, J. Montiel, A. Perzylo,B. Schießle, M. Tenorth, O. Zweigle, and R. De Molengraft, RoboEarth, IEEE Robot. Autom. Mag., vol. 18, no. 2, pp.69 – 82, Jun. 2011.

[7] Wan, Jiafu, et al. "Cloud robotics: Current status and open issues." *IEEE Access* 4 (2016): 2797-2807.

[8] J. Kuffner, Cloud-enabled robots, in Proc. IEEE-RAS Int. Conf. Humanoid Robot., Nashville, TN, USA, 2010.

[9] “A history of cloud computing.” http://www.computerweekly.com/feature/ A-history-of-cloud-computing, March 2009. Accessed: 2020-01-16.

[10] “Amazon media room: History & timeline.” http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-corporateTimeline, September 2011. Accessed: 2020-01-16.

[11] “Amazon web services blog: Amazon ec2 beta.” http://aws.typepad.com/aws/2006/08/ amazon_ec2_beta.html, August 2006.

Accessed: 2020-01-16.

[12] Mason, Andrew G. (2002). Cisco Secure Virtual Private Network. Cisco Press. p. 7

[13] Sheila Frankel, Karen Kent, Ryan Lewkowski, Angela D. Orebaugh, Ronald W. Ritchey, and Steven R Sharma. Guide to IPsec VPNs. NIST Special Publication, 800-77, 2005.

[14] Bruce Schneier, David Wagner, and Mudge. Cryptanalysis of Microsoft's PPTP Authentication Extensions (MS-CHAPv2). In Secure Networking|CQRE [Secure]'99, pages 192{203. Springer, 1999.

[15] Sheila Frankel, Paul Ho_man, Angela Orebaugh, and Richard Park. Guide to SSL VPNs. NIST Special Publication, 800-113, 2008.

[16] John R. Vacca. Virtual private network security. In Complete Book of Remote Access: Connectivity and Security, pages 251{267. Auerbach Publications, 2002.

[17] Berry Hoekstra, Damir Musulin, and Jan Just Keijser. Comparing TCP performance of tunneled and non-tunneled tra_c using Open-VPN. Master's thesis, Universiteit Van Amsterdam, System & Network Engineering, 2011.

[18] Adeyinka, Olalekan. "Analysis of problems associated with IPSec VPN Technology." *2008 Canadian Conference on Electrical and Computer Engineering*. IEEE, 2008.

[19] Stijn Huyghe. OpenVPN 101: introduction to OpenVPN. https://openvpn.net/papers/openvpn-101.pdf. Accessed: 2020-1-17.

[20] OpenVPN Manual. https://community.openvpn.net/openvpn/wiki/Openvpn23ManPage.Accessed: 2020-1-17.

[21] OpenVPN network protocol overview documentation file. https://github.com/OpenVPN/openvpn/blob/master/doc/doxygen/doc_protocol_overview.h. Accessed: 2020-1-17.

[22] Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." ICRA workshop on open source software. Vol. 3. No. 3.2. 2009.

[23] Chandra, Rohit, et al. Parallel programming in OpenMP. Morgan kaufmann, 2001.

[24] OpenMP Loop Scheduling https://software.intel.com/en-us/articles/openmp-loop-scheduling. Accessed: 2020-1-20

[25] Muja, Marius, and David Lowe. "Flann-fast library for approximate nearest neighbors user manual." *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* (2009).

[26] Wayne W. Schmaedeke. Information-based sensor management. In Optical Engineering and Photonicsin Aerospace Sensing, pages 156{164. International Society for Optics and Photonics, 1993.

[27] James M Manyika and Hugh F Durrant-Whyte. Information-theoretic approach to management in decentralizeddata fusion. In Applications in Optical Science and Engineering, pages 202{213. InternationalSociety for Optics and Photonics, 1992.

[28] S. Thrun, W. Burgard, and D. Fox, Probabilistic Robotics (Intelligent Robotics and Autonomous Agents series), ser. Intelligent robotics and autonomous agents. The MIT Press, aug 2005.

[29] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In Proceedings of the Twelfth International Conference on International Conference on Machine Learning, pages 362-370, 1995.

[30] J. M. Porta, N. Vlassis, M. T. J. Spaan, and P. Poupart, “Point-Based Value Iteration for Continuous POMDPs,” The Journal of Machine Learning Research, vol. 7, pp.2329–2367, Dec 2006

[31] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, J. Internet Services Appl. 1 (1) (2010) 7–18.

[32] G.Yang, H.Li, Y.Dou, H.Tang, “Application and research on Winograd parallel algorithm of matrix multiplication based on OpenMP,” Application Research of Computers, p2435-p2437, July 2012.

[33] Reyes-Ortiz, Jorge Luis, Luca Oneto, and Davide Anguita. "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf." INNS Conference on Big Data. Vol. 8. 2015.

[34] Peng, Ying, and Fang Wang. "Cloud computing model based on MPI and OpenMP." 2010 2nd International Conference on Computer Engineering and Technology. Vol. 7. IEEE, 2010.

[35] Wottrich, Rodolfo, Rodolfo Azevedo, and Guido Araujo. "Cloud-based OpenMP parallelization using a mapreduce runtime." 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing. IEEE, 2014.

[36] Alonso, Pedro, et al. "Neville elimination on multi-and many-core systems: OpenMP, MPI and CUDA." The Journal of Supercomputing 58.2 (2011): 215-225.

[37] Bodin, Francois, and Stephane Bihan. "Heterogeneous multicore parallel programming for graphics processing units." Scientific Programming 17.4 (2009): 325-336.

[38] Yang, Chao-Tung, Chih-Lin Huang, and Cheng-Fang Lin. "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters." Computer Physics Communications 182.1 (2011): 266-269.

[39] Guo, Xing, et al. "Parallel computation of aerial target reflection of background infrared radiation: Performance comparison of OpenMP, OpenACC, and CUDA implementations." IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 9.4 (2016): 1653-1662.

[40] Memeti, Suejb, et al. "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption." Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing. 2017.

[41] D.Sun, H.Tang, Research on Parallel Simplification of Road Surface Point Cloud, Intelligent Computer and Applications, p307-312, Septermber 2019

[42] Greigarn, Tipakorn, and M. Cenk Çavuşoğlu. "Active sensing for continuous state and action spaces via task-action entropy minimization." *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016.

[43] Steven M LaValle. Planning Algorithms. Cambridge University Press, May 2006.

[44] Matthew Mason. The mechanics of manipulation. In IEEE International Conference on Robotics and Automation, volume 2, pages 544-548, 1985.

[45] Bull, J. Mark. "Measuring synchronisation and scheduling overheads in OpenMP." Proceedings of First European Workshop on OpenMP. Vol. 8. 1999.

[46] Olivier, Stephen L., and Jan F. Prins. "Evaluating OpenMP 3.0 run time systems on unbalanced task graphs." International Workshop on OpenMP. Springer, Berlin, Heidelberg, 2009.

[47] Lê-Quôc, Alexis, Mike Fiedler, and Carlo Cabanilla. "The top 5 AWS EC2 performance problems." *Whitepaper. Datadog Inc* (2013).

[48] Burger, Thomas W. "Intel multi-core processors: Quick reference guide." *cachewww. intel. com/cd/00/00/23/19/231912_231912. pdf* (2005).

[49] Marr, Deborah T., et al. "Hyper-Threading Technology Architecture and Microarchitecture." Intel Technology Journal 6.1 (2002).

[50] Tian, Xinmin, et al. "Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance." Intel Technology Journal 6.1 (2002).

[51] Tau Leng, Rizwan Ali, et al. "An empirical study of hyper-threading in high performance computing clusters." Linux HPC Revolution 45 (2002).

[52] Marr, Deborah T., et al. "Hyper-Threading Technology Architecture and Microarchitecture." *Intel Technology Journal* 6.1 (2002).

[53] Saini, Subhash, et al. "The impact of hyper-threading on processor resource utilization in production applications." 2011 18th International Conference on High Performance Computing. IEEE, 2011.

[54] Ayguadé, Eduard, et al. "Is the schedule clause really necessary in OpenMP?." International workshop on OpenMP applications and tools. Springer, Berlin, Heidelberg, 2003.

[55] Gustafson, John L. "Reevaluating Amdahl's law." Communications of the ACM 31.5 (1988): 532-533.

[56] Rufino, José, Ana I. Pereira, and Jan Pidanic. "coPSSA-Constrained parallel stretched simulated annealing." 2015 25th International Conference Radioelektronika (RADIOELEKTRONIKA). IEEE, 2015.

[57] Ciamulski, Tomasz, and Maciej Sypniewski. "Linear and superlinear speedup in parallel FDTD processing." 2007 IEEE Antennas and Propagation Society International Symposium. IEEE, 2007.

[58] Ristov, Sasko, et al. "Superlinear speedup in HPC systems: Why and when?." 2016 Federated Conference on Computer Science and Information Systems (FedCSIS). IEEE, 2016.

[59] Guz, Zvika, et al. "Many-core vs. many-thread machines: Stay away from the valley." *IEEE Computer Architecture Letters* 8.1 (2009): 25-28.

[60] Li, Jianming, Wei Wang, and Xiangpei Hu. "Parallel particle swarm optimization algorithm based on CUDA in the AWS cloud." 2015 Ninth International Conference on Frontier of Computer Science and Technology. IEEE, 2015.