A Study of Uniform Job Assignment Algorithms to Workers in User-PC Computing System

September, 2024

Xudong Zhou

Graduate School of
Natural Science and Technology

(Doctor's Course)
Okayama University

Dissertation submitted to
Graduate School of Natural Science and Technology
of
Okayama University
for
partial fulfillment of the requirements
for the degree of
Doctor of Philosophy.

Written under the supervision of

Professor Nobuo Funabiki

and co-supervised by
Professor Satoshi Denno
and
Professor Yasuyuki Nogami

OKAYAMA UNIVERSITY, September 2024.

To Whom It May Concern

We hereby certify that this is a typical copy of the original doctor thesis of
Xudong Zhou

Signature of the Supervisor	Seal of
Prof. Nobuo Funabiki	Graduate School of Natural Science and Technology

Abstract

Recently, the computing ability of a *personal computer (PC)* has been greatly increased with the faster CPU clock cycle, the greater number of CPU cores, the larger memory size, and the higher storage capacity/access speed. Since a PC can be available with a low cost normally, the collection of PCs that are owned by users in an organization, called *user-PCs* in this thesis, can provide a significantly efficient computing platform with a very small cost for complex computational projects by running them on idling resources of *user-PCs*. Then, the proper assignment of incoming jobs to available *user-PCs* or *scheduling* is very important to use computational resources properly.

To implement this concept, we have studied and devised the *User-PC Computing (UPC)* system as a low-cost and high performance distributed computing platform based on the *master-worker model*. A user can submit a job to the web server through the browser in the UPC system, then the web server will send the job to the *UPC master*. When the *UPC master* receives a job, it will assign the job to an idling *UPC worker*. After the worker completes running the job, the master will receive the result. Finally, the *UPC master* will return the result to the web browser of the user. As a result, the *job assignment algorithm* is critical to achieve the minimization for *makespan* to complete all the demanded jobs in the UPC system.

Some users need to execute a lot of *uniform jobs* that will use the identical program but slightly different input data/parameters. Such *uniform jobs* may include deep learning (machine learning), physics simulations, software testing, computer network simulations, mathematical modeling, and mechanics modeling. These *uniform jobs* share the characteristic of requiring similar CPU time when executed on a specific PC, regardless of the variations in input data. Due to the large number of input data sets, the overall completion time is often extended. For example, in physics or network simulations, it can take several days to find the best result by repeatedly running the program while slightly changing some parameter values. This work can be common in research activities using computer simulations.

As the first contribution of this thesis, I present the static *uniform job* assignment algorithm to workers in the UPC system. To minimize the *makespan* for completing all the jobs in the system, a set of linear equations are derived to actually find the number of jobs assigned to each worker, such that the CPU time to complete the assigned jobs becomes equal between the workers.

For evaluations of the proposal, I consider the *uniform jobs* in the *code testing* application in *Android Programming Learning Assistance System (APLAS)*. This job runs the test code with the source code submitted from a student. Since a lot of students submitted codes, it will take long time to complete all of them. I applied the proposal to six test codes and 578 source codes. The results show that the proposal could reduce *makespan* by up to 13% of that by the FIFO approach, which confirmed the effectiveness of it.

As the second contribution of this thesis, I propose an extension of the first static *uniform job* assignment algorithm. The proposal addresses the scenario where *uniform jobs* of various types are assigned concurrently, deriving modified multiple simultaneous linear equations to consider dif-

ferent *uniform job* applications, including *OpenPose*, *OpenFOAM*, and *APLAS*. Then, the proposal finds the *lower bound* on *makespan* where every worker requires the same CPU time to complete the assigned jobs.

For evaluations, we prepared 41 images of human bodies in *OpenPose*, 32 parameter sets in *OpenFOAM*, and 578 source codes in *APLAS*. These jobs were applied to the proposed algorithm and were assigned to six workers in the UPC system. The results show that the *makespan* was reduced by 5% on average from the results by the first static *uniform job* assignment algorithm. Thus, the effectiveness of the proposal was confirmed.

As the third contribution of this thesis, I further extend the second static *uniform job* assignment algorithm to handle *uniform jobs* whose CPU times are multiples of the CPU time of the *unit job*. The job who has the shortest CPU time is called the *unit job* for convenience. In physics or network simulations, which are typical scenarios, the CPU time is proportional to the inputs of the program such as the number of meshes or the number of iteration steps. In the proposal, the necessary number of *unit job* to satisfy the CPU time are prepared for each input job.

For evaluations, I selected *OpenFOAM* and *Network Simulator 3 (NS-3)* as the *uniform job* applications where the CPU time can be multiple of the *unit job* time. We prepared 176 *OpenFOAM* jobs and 261 *NS-3* jobs for finding the assignments to six workers by the proposal in the UPC system. The results show that the proposed assignment algorithm reduces the *makespan* compared to the second static *uniform job* assignment algorithm and others, thereby confirming its effectiveness.

As the last contribution of this thesis, I present a design and implementation of a stationery product recognition method using the *YOLOv8* model at two stages. In order to reduce the retraining time and improve the accuracy, the first-stage model is applied to recognize the category of the target object from the given image and the second-stage models recognize the product name/type among those in the category.

For evaluations, we prepared 795 images of 45 different stationery products in 9 categories, and trained the total of 10 *YOLO* models by *NVIDIA's RTX-3060 GPU*. Then, I measured the training time, the retraining time, and the recognition accuracy of the proposal. As the experiment results, I show the difference of them between the conventional one model case and the proposed two-stage model case, which confirmed the effectiveness of our proposal.

In future works, I plan to utilize the UPC system to facilitate AI tasks such as training and execution. In this context, the assignment algorithm plays a crucial role in efficiently managing these tasks and utilizing computational resources effectively.

Acknowledgements

I would like to extend my deepest gratitude to those who have supported and guided me throughout my Ph.D. journey.

First and foremost, I am profoundly grateful to my supervisor, Professor Nobuo Funabiki, for his steadfast support, invaluable guidance, and insightful feedback. His expertise and encouragement were crucial in shaping the direction of my research and overcoming numerous challenges.

I am thankful to my Ph.D. co-supervisors, Professor Satoshi Denno and Professor Yasuyuki Nogami, for their constructive criticisms and suggestions which significantly enhanced the quality of my work. I would also like to give deep appreciation to Professor Minoru Kuribayashi of Tohoku University for his advice and support.

I owe a special debt of gratitude to my colleagues and friends at Funabiki Lab for their camaraderie, stimulating discussions, and continuous encouragement. In particular, I would like to thank Dr. Hein Htet and Dr. Ariel Kamoyedji for their technical assistance and moral support.

My heartfelt thanks go to my parent, Mr. Daoyuan Zhou and Ms. Xu Li, for their endless love, patience, and sacrifices. Their belief in my abilities and their emotional support have been my bedrock throughout this demanding journey.

A profound thank you to my girlfriend, Ms. Taiyi Tang, for her love, patience, and encouragement. Your constant support and faith in me have been invaluable, especially during the most challenging times of this journey.

Finally, I would like to acknowledge all the individuals and organizations whose contributions, direct or indirect, have played a part in the completion of this thesis. Thank you all for being part of my academic journey and for helping me reach this significant milestone.

Xudong Zhou
Okayama University, Japan
September 2024

List of Publications

Journal Papers

1. **Xudong Zhou**, Nobuo Funabiki, Hein Htet, Ariel Kamoyedji, Irin Tri Anggraini, Yuanzhi Huo and Yan Watequlis Syaifudin, "A Static Assignment Algorithm of Uniform Jobs to Workers in a User-PC Computing System Using Simultaneous Linear Equations," Algorithms, Vol. 15, No. 10, pp. 369 (2022).

International Conference Papers

2. **Xudong Zhou**, Nobuo Funabiki, Shinji Sugawara, Hein Htet, and Ariel Kamoyedji, "A Static Uniform Job Assignment Algorithm to Workers in User-PC Computing System," Proceedings of the 10th International Conference on Computer and Communications Management (ICCCM 2022), pp. 24–30 (Okayama, Japan, 2022).

3. **Xudong Zhou**, Nobuo Funabiki, Yanhui Jing, Yanqi Xiao and Wen-Chung Kao, "A Design and Implementation of Stationery Product Recognition Method Using Two-Stage YOLO v8 Model," Proceedings of the 12th International Conference on Information and Education Technology (ICIET 2024), pp. 327-332 (Yamaguchi, Japan, 2024).

Other Papers

4. **Xudong Zhou**, Nobuo Funabiki, Lynn Htet Aung and Xiang Xu, "An Extension of Static Worker Assignment Algorithm to Uniform Jobs with Multiple CPU Time in User-PC Computing System," 2023 IEICE Conferences Archives, BS-2-15, (Saitama, Japan, 2023).

List of Figures

1.1 Overview of UPC system. 2

2.1 Usage of Docker in UPC system. 7
2.2 Docker image generation process at master. 8
2.3 Memory usage rate without job control. 10
2.4 Memory usage rate with job control. 10

3.1 Overview of *APLAS* system. 12

4.1 Flowchart of the proposal. 25

6.1 *YOLO* base idea. 40
6.2 Origin image. 42
6.3 Resized image. 42

List of Tables

3.1 PC specifications. 16
3.2 Job specifications. 17
3.3 Constant CPU time to start jobs (sec). 17
3.4 Increasing CPU time at *PC1~PC4* (sec). 17
3.5 Increasing CPU time at *PC5* (sec). 17
3.6 Increasing CPU time at *PC6* (sec). 18
3.7 Maximum *makespan* results (sec). 18
3.8 FCFS *makespan* detail (sec). 18
3.9 T-FCFS *makespan* detail (sec). 19
3.10 Proposal *makespan* detail (sec). 19

4.1 PC specifications. 26
4.2 Job specifications. 27
4.3 Constant CPU time to start jobs (s). 27
4.4 Increasing CPU time at *PC1~PC4* (s). 28
4.5 Increasing CPU time at *PC5* (s). 28
4.6 Increasing CPU time at *PC6* (s). 28
4.7 Maximum *makespan* results (s). 29
4.8 FCFS *makespan* detail (s). 29
4.9 T-FCFS *makespan* detail (s). 30
4.10 Proposal *makespan* detail (s). 30
4.11 Maximum *makespan* results (s) by proposal. 31

5.1 PC Specifications. 36
5.2 Constant CPU time to prepare jobs (sec). 36
5.3 Required CPU time for best throughput (sec). 36
5.4 Total *makespan* (sec). 37
5.5 Number of assigned jobs by previous. 37
5.6 Number of assigned jobs by proposal. 37

6.1 Dataset details. 42
6.2 *YOLOv8* pre-trained model details. 43
6.3 PC specification. 44
6.4 Mean average precision at 50. 44
6.5 Retraining time Comparison 44

Contents

Abstract i

Acknowledgements iv

List of Publications vi

List of Figures viii

List of Tables x

1 Introduction 1
- 1.1 Background 1
- 1.2 Contributions 2
  - 1.2.1 Static Uniform Job Assignment Algorithm 2
  - 1.2.2 Extension for Concurrent Multi-Type Uniform Job Allocation 3
  - 1.2.3 Extension for Uniform Jobs with Multiple CPU Time 3
  - 1.2.4 Stationery Product Recognition Method Using Two-Stage YOLOv8 3
- 1.3 Contents of This Dissertation 3

2 Review of the User-PC Computing System 6
- 2.1 Web Server 6
  - 2.1.1 Basic Functions 6
- 2.2 UPC Master 6
  - 2.2.1 MySQL and Docker 7
  - 2.2.2 Basic Functions 7
  - 2.2.3 Docker Image Generation 7
  - 2.2.4 Worker Management 8
- 2.3 UPC Worker 9
  - 2.3.1 Basic Functions 9
  - 2.3.2 Job Control Function 9
- 2.4 Summary 10

3 Proposal of Static Uniform Job Assignment Algorithm 12
- 3.1 APLAS System 12
- 3.2 Static Uniform Job Assignment Algorithm 13
  - 3.2.1 Conditions for Uniform Job Assignment 13
  - 3.2.2 Problem Formulation 13
  - 3.2.3 Static Uniform Job Assignment Algorithm 14

3.3 Evaluation . . . 16
3.3.1 Experiment Setup . . . 16
3.3.2 Comparative Algorithms . . . 17
3.3.3 Makespan Results . . . 18
3.3.4 Discussion . . . 19
3.4 Summary . . . 19

4 Extension of Static Assignment Algorithm for Concurrent Multi-Type Uniform Job Allocation 21
4.1 Three Uniform Job Applications . . . 21
4.1.1 OpenPose . . . 21
4.1.2 OpenFOAM . . . 22
4.2 Proposal of Static Uniform Job Assignment Algorithm . . . 22
4.2.1 Objective . . . 22
4.2.2 Simultaneous Linear Equations . . . 22
4.2.3 Problem Formulation . . . 23
4.2.4 Conditions for Uniform Job Assignment . . . 24
4.2.5 Static Uniform Job Assignment Algorithm . . . 24
4.3 Evaluation . . . 26
4.3.1 Testbed UPC System . . . 26
4.3.2 Jobs . . . 26
4.3.3 CPU Time . . . 27
4.3.4 Comparative Algorithms . . . 28
4.3.5 Total Makespan Results . . . 29
4.3.6 Individual Makespan Results . . . 29
4.3.7 Discussions . . . 30
4.4 Extension to Multiple Job Types Assignment . . . 30
4.4.1 Algorithm Extension . . . 31
4.4.2 Total Makespan Results . . . 31
4.4.3 Discussions . . . 32
4.5 Summary . . . 32

5 Extension of Static Assignment Algorithm for Uniform Jobs with Multiple CPU Time 34
5.1 OpenFOAM and NS-3 . . . 34
5.2 Proposal of Algorithm Extension . . . 34
5.2.1 Previous Algorithm . . . 34
5.2.2 Extension of Algorithm . . . 35
5.3 Evaluation . . . 35
5.3.1 Experiment Setup . . . 35
5.3.2 Makespan Results . . . 36
5.3.3 Assignments Results . . . 36
5.4 Summary . . . 37

6 Proposal of Stationery Product Recognition Method Using Two-Stage YOLOv8 39
6.1 YOLO Family . . . 39
6.2 Proposal of Stationery Product Recognition Method . . . 41
6.2.1 Dataset for YOLO . . . 41

6.2.2 Two-Stage YOLO Models 41
6.3 Evaluation 43
6.3.1 Experiments Results 43
6.3.2 Discussion 44
6.4 Summary 44

7 Related Works in Literature 46

8 Conclusion 51

References 53

Chapter 1

Introduction

1.1 Background

As machine learning technologies for *artificial intelligence (AI)* become more useful and common in various applications, the importance of low-cost and high-performance computing platforms has increased, since AI requires a lot of computing resources. On the other hand, the performance of *personal computers (PCs)* has been dramatically enhanced with the advancements of LSI technologies. Particularly, the number of CPU cores has significantly increased so that multi-threaded programs can run on them in parallel, thus drastically reducing the required CPU time for job completion. As a result, a collection of PCs that are owned by an organization's members, called *user-PCs* in this thesis, can provide a significantly efficient and very low cost platform for complex computational projects by running the latter on idling resources of *user-PCs*. To implement this concept, we have studied and devised the *User-PC Computing (UPC)* system as a low-cost and high performance distributed computing platform based on the *master-worker model* [1].

The UPC system provides high computational power to the members of an organization by using idling computing resources of their PCs [2]. Moreover, the UPC system is different from the *Volunteer Computing (VC)* system [3] in that it can achieve high dependency by using trusted PCs in the same organization or group. Figure 1.1 is an overview of the UPC system. In the UPC system, a user can submit a job to the UPC web server, using a web browser. Then, the UPC web server sends the job to the *UPC master*, which assigns it to a *UPC worker*, and receives the result upon completion. Lastly, the *UPC master* returns the received result to the user web browser.

The UPC system allows various application programs to run on various worker PC environments using *Docker* [4]. *Docker* is a popular software tool that has been designed to make it easier to create, deploy, and run an application program on various platforms using the *container technology* [5]. A *Docker container image* is a lightweight, standalone, and executable package containing all the software that is needed to run the application program. It includes the source code, the runtime environment, the system tools, the system libraries, and the setting parameters.

The UPC system work flow consists of seven steps: 1) a user submits *computing projects (jobs)* from their Web browser to the *UPC master* via the UPC Web server, 2) the master generates a *Docker image* for each job, 3) the master finds an assigned *UPC worker* for each job using a job-worker assignment algorithm, 4) the master transmits the jobs *Docker images* to the assigned workers, 5) each assigned UPC worker computes its assigned job in a *Docker container* and transmits the result to the master upon completion, 6) the master receives all jobs results from workers, and 7) the master returns the project result to the user upon all jobs results reception.

Previously, my group has proposed the algorithm of assigning *non-uniform jobs* to workers in

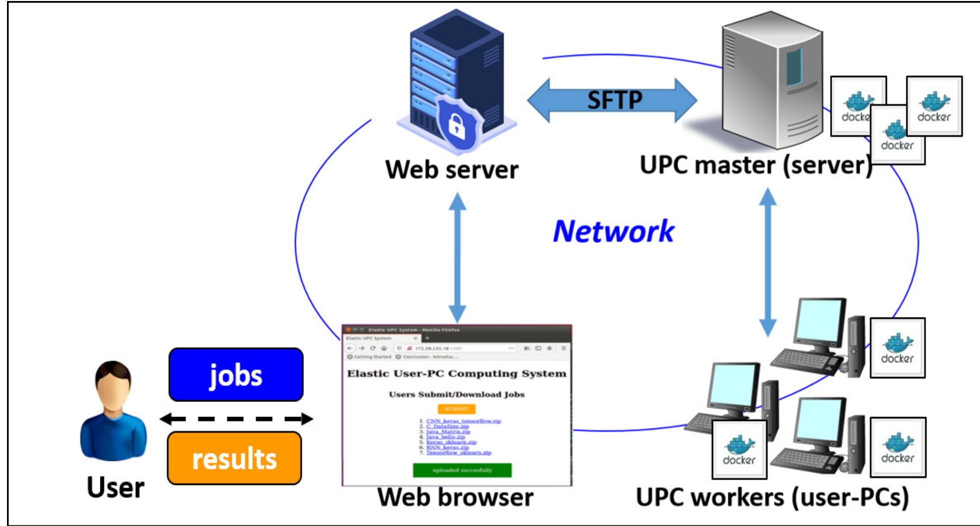

Figure 1.1: Overview of UPC system.

the UPC system [6]. In *non-uniform jobs*, the programs may be much different from each other, including the developed programming languages, the number of threads, and the requiring data. The execution time for each *non-uniform job* is highly different from the others. The previous algorithm can find the job-worker assignment through two stages sequentially, of which are heuristic due to the nature of the *NP-hardness* and cannot guarantee the optimality of the solution.

Some applications need to execute a lot of *uniform jobs* that use the identical program but with slightly different input data/parameters. The applications include deep learning (machine learning), physics simulations, software testing, computer network simulations, mathematical modeling, and mechanics modeling. These applications share the characteristic of requiring similar CPU time when executed on a specific PC, regardless of the variations in input data. Due to the large number of input data sets, the overall completion time is often extended. For example, in physics or network simulations, it can take several days to find the best result by repeatedly running the program while slightly changing some parameter values. This work can be common in research activities using computer simulations. As a result, the *uniform job assignment algorithm* is critical to achieve the minimization for *makespan* to complete all the demanded *uniform jobs* in the UPC system.

1.2 Contributions

In this thesis, I propose several *uniform job* assignment algorithms for the User-PC computing system and an *YOLOv8* based stationery product recognition method.

1.2.1 Static Uniform Job Assignment Algorithm

I first present the static *uniform job* assignment algorithm for the UPC system. To minimize the *makespan* for completing all the jobs in the system, a set of linear equations are derived to actually find the number of jobs assigned to each worker, such that the CPU time to complete the assigned jobs becomes equal between the workers. I evaluated the proposal using *uniform jobs* in the code testing application of the *Android Programming Learning Assistance System (APLAS)*,

which tests the source codes submitted by students. Applying the algorithm to six test codes and 578 source codes show that it reduced the *makespan* by up to 13% compared to the FIFO approach, demonstrating its effectiveness.

1.2.2 Extension for Concurrent Multi-Type Uniform Job Allocation

Next, I introduce the extension of the first static *uniform job* assignment algorithm. The proposal addresses the scenario where *uniform jobs* of various types are assigned concurrently, deriving modified multiple simultaneous linear equations to consider different *uniform job* applications, including *OpenPose*, *OpenFOAM*, and *APLAS*. Then, the proposal finds the *lower bound* on *makespan* where every worker requires the same CPU time to complete the assigned jobs. For evaluation, 41 images in *OpenPose*, 32 parameter sets in *OpenFOAM*, and 578 source codes in *APLAS* were used, and these jobs were assigned to six workers in the UPC system using the proposed algorithm. The results demonstrate the *makespan* was reduced by the average of 5% compared to the results from the first static *uniform job* assignment algorithm, confirming the proposal's effectiveness.

1.2.3 Extension for Uniform Jobs with Multiple CPU Time

Third, I further extend the second static *uniform job* assignment algorithm to handle *uniform jobs* whose CPU times are multiples of the CPU time of the *unit job*. The job who has the shortest CPU time is called the *unit job* for convenience. In physics or network simulations, which are typical scenarios, the CPU time is proportional to the inputs of the program such as the number of meshes or the number of iteration steps. In the proposal, the necessary number of *unit job* to satisfy the CPU time are prepared for each input job. For evaluation, we prepared 176 *OpenFOAM* jobs and 261 *NS-3* jobs to be assigned to six workers in the UPC system. The results show that the proposed assignment algorithm reduced the *makespan* compared to the second static *uniform job* assignment algorithm and others, demonstrating its effectiveness.

1.2.4 Stationery Product Recognition Method Using Two-Stage YOLOv8

Finally, I present a design and implementation of a stationery product recognition method using the *YOLOv8* model at two stages. In order to reduce the retraining time and improve the accuracy, the first-stage model is applied to recognize the category of the target object from the given image and the second-stage models recognize the product name/type among those in the category. For evaluations, we prepared 795 images of 45 different stationery products in 9 categories, and trained the total of 10 *YOLO* models by *NVIDIA's RTX-3060 GPU*. Then, I measured the training time, the retraining time, and the recognition accuracy. As the experiment results, I show the difference of them between the conventional one model case and the proposed two-stage model case, which confirmed the effectiveness of our proposal.

1.3 Contents of This Dissertation

The remaining part of this thesis is organized as follows: Chapter 2 reviews the the UPC system platform using *Docker*. Chapter 3 presents a static *uniform job* assignment algorithm for the UPC system and its evaluation through experiments in code testing application. Chapter 4 presents an

extension of the first static assignment algorithm for concurrent multi-type *uniform job* allocation and its evaluation through experiments in three different *uniform job* applications. Chapter 5 presents an extension of the second contribution to *uniform jobs* with multiple CPU time in UPC system and its evaluation through experiments. Chapter 6 presents a stationery product recognition method using two-stages *YOLOv8* model and its evaluation through experiments in accuracy and training time. Chapter 7 reviews relevant works in literature. Finally, Chapter 8 concludes this thesis with some future works.

Chapter 2

Review of the User-PC Computing System

In this chapter, I present the design and implementation of the UPC system platform using *Docker* [1]. The UPC system is composed of the three components, the *web server*, the *UPC master*, and *UPC workers*, shown in Figure 1.1. The implementations of the basic functions in each component will be discussed.

2.1 Web Server

The web server is implemented by using *Node.js* [7]. *Node.js* is an open source server environment and can run on various platforms including *Linux*, *Windows*, and *Mac OS*. It offers the running environment of *JavaScript* programs on the server [7]. Thus, the following three basic functions are implemented by *JavaScript* programs.

2.1.1 Basic Functions

In the web server, the following three functions are implemented with different threads.

- The *job acceptance thread* accepts the jobs submitted from the web browser. One job usually consists of the source codes, the required platforms, and the library lists.
- The *job transmission thread* transfers the submitted jobs to the UPC master using *SSH File Transfer Protocol (SFTP)* [8].
- The *result reception thread* receives the results of the jobs from the UPC master and stores them so that user can download them.

In our implementation, the *Linux OS* is adopted. The built-in module in *Node.js* is used to listen to the server ports and give the responses to the UPC master. The browser page programs are implemented using HTML5, CSS, and JavaScript.

2.2 UPC Master

The programs in the UPC master are implemented using *Python*. The *Python* multi-threaded module supports powerful and high-level threads [9]. Multiple workers are connected with the UPC master, where one thread in the server program is allocated to each worker.

2.2.1 MySQL and Docker

MySQL [10] is adopted as the database system to manage the data of the UPC system.

The *Docker* container technology [11] is used to provide the flexibility and portability for running various jobs on different worker platforms. It builds the *Docker* image to offer the software environment for running each job, including source codes, libraries, middle ware, and parameters, so that the job can run on any worker PC without considering the installed software.

2.2.2 Basic Functions

In the UPC master, the following four basic functions are implemented with different threads.

- The *job management thread* receives the request for a new job from the web server by detecting the newly updated files using SFTP. Then, it prepares a new job by unzipping, inserting and modifying the *Docker file template*, and builds and saves the complete job running environment.

- The *worker management thread* receives the joining request from a UPC worker. When the master receives the request, it creates a new thread for the new worker, collects the information on the worker, and stores them at the master' database.

- The *job transmission thread* sends a job in the job queue to the assigned worker. It is repeated until the job queue is empty.

- The *result uploading thread* sends the result from the worker to the web server using SFTP.

2.2.3 Docker Image Generation

The UPC master accepts the jobs from the web server. Then, for each job, it prepares the *Docker file* that contains the list of the instructions to build the *Docker image* that bundles the environments and the applications, and executes it as a *Docker container*, shown in Figure 2.1. In our implementation, the *Docker file* is automatically created by analyzing the list of the requirements for the job from the user and the extensions of source codes.

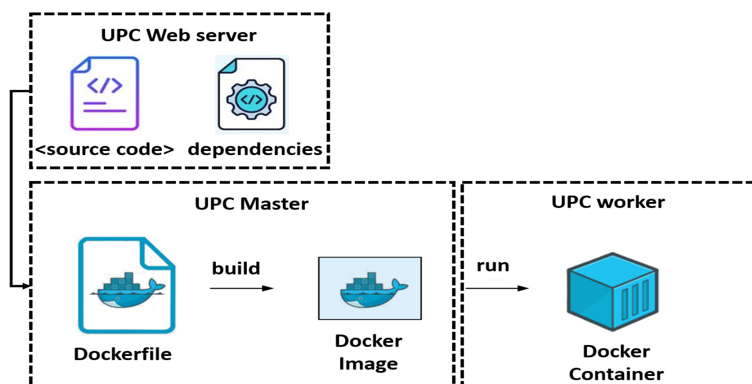

Figure 2.1: Usage of Docker in UPC system.

Figure 2.2 shows the details of the process. The UPC master performs the following steps to generate the Docker Image for each submitted job.

1. It unzips the job, examines the program type, and explores the requirement list.

2. It compares and checks the information obtain in step 1 with the log data under the temporary information directory that stores the previously built *Docker* image information.

3. It refers the previous built *Docker* image if the running environment, libraries, and dependencies are almost similar with the current job's requirements.

4. If not, it refers the base image of the previously built *Docker* image when only the running environment is same.

5. Otherwise, it generates a new *Docker* image for the current job by following the instructions of the generated *Docker file*.

6. It accesses to *Public Remote Repository* to download and install the necessary images, libraries, and platforms, and chooses the small and light package to reduce the image size to a minimum.

7. It saves them as a *Docker* image when successfully finished, and adds it in the correspondence job list.

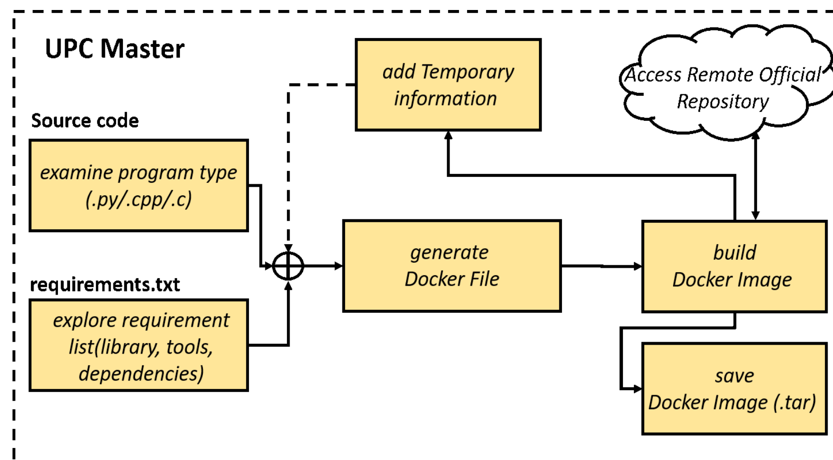

Figure 2.2: Docker image generation process at master.

2.2.4 Worker Management

When a PC joins the UPC system as a new worker, the UPC master collects the static performance-related information of the PC, such as the memory size, the CPU clock rate, the number of cores, and the hard disk size using *psutil* [12]. The master also periodically collects the dynamic performance information of the PC, such as the percentage of the current resource usage and the available resource status. The UPC master keeps all the information in the database. Thus, if the worker cannot keep running the job because of the resource usage shortage, the UPC master can send the stop alert of the running job to the worker, and the resume alert when resources are available to use.

2.3 UPC Worker

The programs of the UPC worker are also implemented using *Python*. The *Docker* container technology is installed to run the *Docker* image for each job on the worker assigned by the UPC master.

2.3.1 Basic Functions

The following five basic functions of the UPC worker are implemented with different threads:

- The *connection initiation thread* finds the address and the port of the UPC master from the socket. Then, the worker is connected to the master by sending the necessary information.
- The *job reception thread* receives the *Docker* image for the job with the *.tar* file and temporarily allocates it in the disk space of the worker.
- The *job execution thread* starts to load and run the received *Docker* image as a container.
- The *job restoring thread* saves the current running states of the jobs in the hard disk and sends the state to the master when the worker runs out all the available resources.
- The *result transmission thread* transfers the result of the job when successfully completing it.

2.3.2 Job Control Function

In the UPC system, any running job on a worker must not disturb the use of the PC by the owner. Thus, the job control function is implemented to stop the running container job and free the memory when the memory usage rate exceeds the given threshold, where 90% is selected from our experiment results [4]. The suspended job would be reloaded to the memory for resuming the job when it falls below the threshold. We discuss the implementation of worker PC memory control on *Linux* or *Windows* operating system.

First, I discuss the implementation for *Linux*. ***'kill'*** command is used to stop the job. Then, ***'kill -STOP #ProcessName'*** command is used to free the memory. If the job can run there again, ***'kill -CONT #ProcessName'*** command is used to resume the job.

Next, I discuss the implementation for *Windows*. ***'taskkill'*** command is used to stop the job. Then, ***'Stop-Process -Name #ProcessName'*** command is used to free the memory. If the job can run there again, ***'Cont-Process -Name #ProcessName'*** command is used to resume the job.

Figure 2.3 shows the change of the memory usage rate of the *Convolutional Neural Network (CNN)* job program. The PC does not work properly at the fourth run. When it exceeds 90%, the PC is hung up and needs to be rebooted, where all the running processes are lost. Therefore, the memory usage rate for the UPC job must be carefully controlled to avoid the problem.

Figure 2.4 shows the change of the memory usage rate when the same *CNN* job program runs on the PC five times. Every time the rate exceeds the given threshold 90%, the job is automatically stopped and about 36% of the memory is released to keep running daily processes by the PC owner.

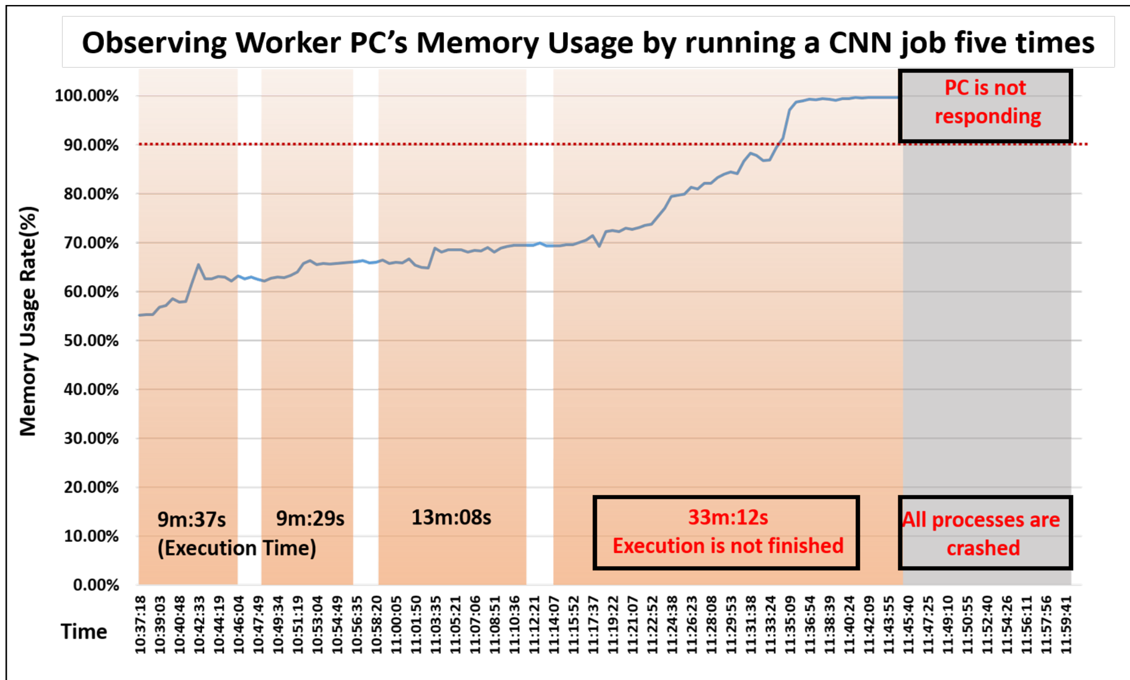

Figure 2.3: Memory usage rate without job control.

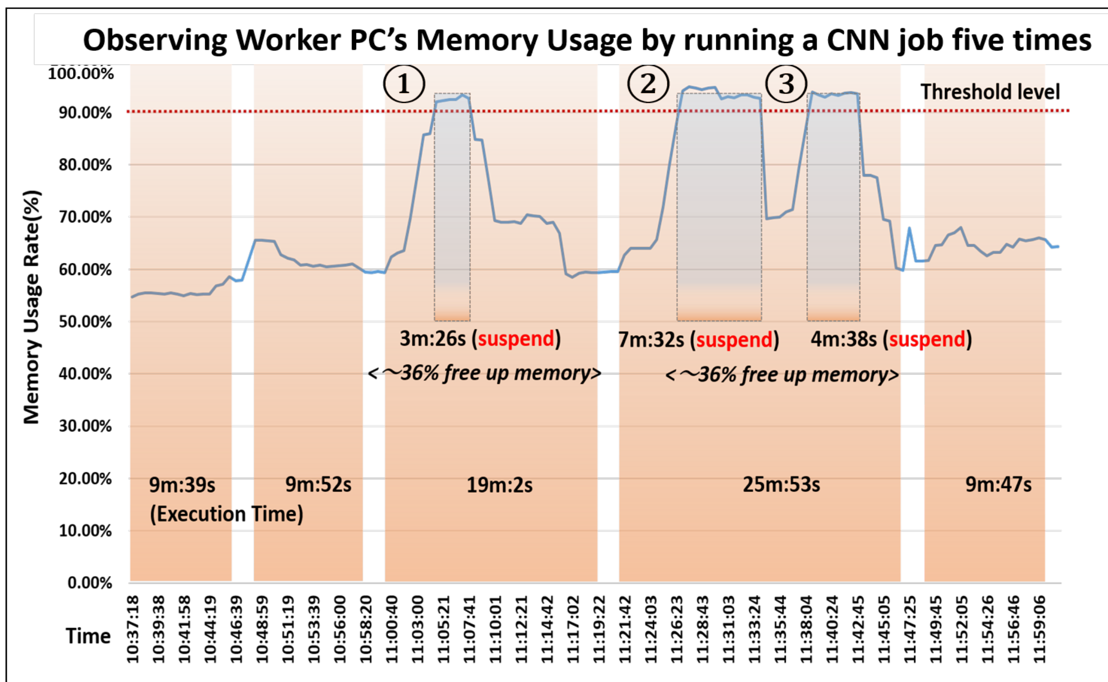

Figure 2.4: Memory usage rate with job control.

2.4 Summary

In this chapter, I presented the design and implementation of UPC system platform using *Docker*. I discussed the job control function and it can be avoided losing the PC owner's processes due to the running of the UPC jobs while the memory usage is high. In the next chapter, I will present the static *uniform job* assignment algorithm for the UPC system and its evaluation.

Chapter 3

Proposal of Static Uniform Job Assignment Algorithm

In this chapter, I present the static *uniform job* assignment algorithm to the workers in the UPC system. To minimize the *makespan* for completing all the jobs in the system, a set of *linear equations* are derived to find the number of jobs to be assigned to each worker, such that the CPU time to complete the assigned jobs becomes equal between the workers. Unlike the previous method in our group [6], the proposed one can find the global optimum solution.

3.1 APLAS System

In this section, I review the *APLAS* system in Figure 3.1. Some practical applications need to execute a lot of *uniform jobs* using the same program but the slightly different input data. For such jobs, the required total CPU time at a worker can be linear to the number of the assigned jobs, because the execution time of any job can be the same. The *software testing* in *Android programming learning assistance system (APLAS)* [13] is the typical application to require *uniform jobs*. *APLAS* is a web application system that has been developed in our group to assist teaching and studying *Android programming* using *Java* and *XML* by teachers and students in the courses. To confirm the validity in satisfying the required specifications of answer source codes to exercise assignments from students, *APLAS* has the function of testing them using *JUnit* and *Robolectric*. Therefore, the jobs execute the same programs with different source codes as the input data. Besides, each job needs a rather long CPU time. To execute the jobs in the UPC system is significant.

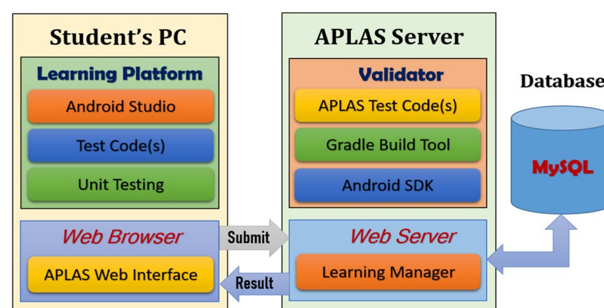

Figure 3.1: Overview of *APLAS* system.

APLAS consists of the *client part* and the *server part*. The *client part* includes the *web browser* to submit jobs and receive their results, and the *learning platform* for students to make answer source codes in *Java* and *XML* to the assignments in *APLAS* using *Android Studio*. The *server part* includes the *validator* to evaluate the correctness of the submitted answers in the background process, and the *web application* to offer the web page interface accessible by the students.

3.2 Static Uniform Job Assignment Algorithm

In this section, I present the static *uniform job* assignment algorithm in the UPC system.

3.2.1 Conditions for Uniform Job Assignment

First, the following conditions are assumed for the uniform job assignment in the UPC system.

- Several job types may exist for uniform jobs, where different job types may need the different CPU time, the memory, and the number of CPU cores.
- The jobs for the same type are requested at one timing, where the jobs for different types will be requested at the different timing.
- Each job is executed on one worker until it is completed.
- Each worker may have the different performance specification from the others.
- Each worker may have the different number of running jobs in parallel using multi-threads for the best throughput.
- The CPU time to run the certain number of jobs in parallel is given for each worker and job type.

3.2.2 Problem Formulation

Next, the problem formulation is described for the static *uniform job* assignment algorithm in the UPC system.

Variables

The following variables are defined for the problem to be solved:

- t: particular job type,
- w: particular worker,
- x_w^t: # of the assigned jobs to worker w for type t,
- m_w^t: *makespan* at worker w to complete all the assigned jobs for type t, and
- d_w: # of running jobs in parallel using multi-threads at worker w.

Given Constants

The following constants are given as the inputs to this problem:

- T: set of job types,
- W: set of workers,
- N^t: total # of jobs for type t,
- D_w: # of jobs for the best throughput at worker w for any type,
- C^t_w: CPU time at worker w to prepare running jobs for type t, and
- $R^t_{w,d}$: CPU time at worker w to run d jobs for type t in parallel.

It is noted that D_w is constant for any job type, because it depends on *JUnit* and *Robolectric* that are common in every job type. C^t_w represents the CPU time to initiate *Gradle Wrapper* daemon and generate *shadow objects* [14] that are necessary to run the validator. $R^t_{w,d}$ is measured at any worker while increasing the number of running jobs in parallel from 1 until D_w.

Objective

The minimization of the maximum *makespan* among the workers in W is given as the objective of this problem:

$$minimize\{max(m^t_w)\} \; for \; t \in T, w \in W \tag{3.1}$$

The *makespan* m^t_w at worker w for type t is given by the summation of the CPU time for preparation and for execution.

Constraints

- The total number of the assigned jobs to workers must be equal to N^t for any type t.

$$\sum_{w \in W} x^t_w = N^t \; (t \in T) \tag{3.2}$$

- Any worker cannot run d jobs in parallel when d is larger than the D_w (let d_w for worker w) due to the PC specifications.

$$d_w \leq D_w \tag{3.3}$$

3.2.3 Static Uniform Job Assignment Algorithm

Finally, it is observed that when *makespan* of every worker is the same, the objective of the problem can be achieved. If the number of assigned jobs to a worker can take any real number, the maximum *makespan* can be reduced by moving some jobs at the bottleneck worker that determines this maximum *makespan* to other workers. Only when every worker has the same *makespan*, the maximum *makespan* cannot be reduced. Thus, the *linear equations* should be derived to find the optimal assignment such that the CPU time required to complete the assigned jobs becomes equal between the workers.

The CPU time of execution is different by the number of running jobs in parallel in each worker. To increase the job completion throughput, D_w jobs for type t should run at worker w as much as possible, since it will give the best throughput. Based on this observation, I present the three-step static *uniform job* assignment algorithm.

First Step

The following *linear equations* are derived assuming that the best CPU time to solve one job at worker w is given by $R^t_{w,D_w}/D_w$ and any value for x^t_w is acceptable:

$$C^t_i + \frac{R^t_{i,D_i}}{D_i} \times x^t_i = C^t_j + \frac{R^t_{j,D_j}}{D_j} \times x^t_j \qquad (3.4)$$
$$for\ i \neq j, i \in W, j \in W, t \in T.$$

By solving the equations in (3.2) and (3.4), the solution $\hat{x^t_w}$ is obtained.

Second Step

The solution in the first step becomes feasible only when $\hat{x^t_w}$ is a multiple of D_w for type t. Unfortunately, $\hat{x^t_w}$ does not satisfy the condition, in general. Therefore, in the second step, as the closest number to satisfy the condition, the following $\tilde{x^t_w}$ jobs will be assigned to the worker (worker w), where $\lfloor y \rfloor$ gives the largest integer equal to or smaller than y:

$$\tilde{x^t_w} = \lfloor \frac{\hat{x^t_w}}{D_w} \rfloor \times D_w \qquad (3.5)$$

Then, the number of the remaining jobs (let r^t for type t) is calculated by:

$$r^t = N^t - \sum_{w \in W} \tilde{x^t_w} \qquad (3.6)$$

Besides, the estimated *makespan* for each worker (let em^t_w for worker w and type t) after the assignments is calculated by:

$$em^t_w = C^t_w + R^t_{w,D_w} \times \frac{\tilde{x^t_w}}{D_w} \qquad (3.7)$$

As the cost function to evaluate the solution quality, the maximum estimated *makespan* among the workers (let EM^t for type t) is calculated by:

$$EM^t = \{max(em^t_w)\}\ for\ w \in W \qquad (3.8)$$

Third Step

In the third step, the remaining jobs (r_t) will be assigned to workers such that the increase of the maximum estimated *makespan* EM^t is minimized. Here, to utilize parallel job computations using multi-threads as much as possible, the simultaneous assignment of multiple jobs to one worker is always considered.

1. Find the worker whose $\hat{em}^t_w$ is smallest among the workers (let worker w).

$$\hat{em}^t_w = em^t_w + R^t_{w,D_w} \tag{3.9}$$

2. Assign Δx^t_w jobs to worker w.

$$\Delta x^t_w = \begin{cases} D_w, & r_t > D_w \\ r_t, & r_t \leq D_w \end{cases} \tag{3.10}$$

3. Update the remaining jobs (r_t), the number of assigned jobs and *makespan* of worker w by:

$$\begin{aligned} x^t_w &= x^t_w + \Delta x^t_w, \\ em^t_w &= em^t_w + R^t_{w,\Delta x^t_w}, \\ r_t &= r_t - \Delta x^t_w \end{aligned} \tag{3.11}$$

4. If the remaining jobs equals to zero, terminate the procedure.

5. Go to 1.

3.3 Evaluation

In this section, I evaluate the proposal through experiments using the testbed UPC system.

3.3.1 Experiment Setup

Table 3.1 shows the PC specifications in the testbed UPC system. One master and six workers are used here.

Table 3.1: PC specifications.

PC	# of cores	CPU model	Clock rate	Memory size	Best throughput(D_w)
PC1	4	Core i3	1.70 GHz	2 GB	1
PC2	4	Core i5	2.60 GHz	2 GB	1
PC3	4	Core i5	2.60 GHz	2 GB	1
PC4	8	Core i7	3.40 GHz	4 GB	2
PC5	16	Core i9	3.60 GHz	8 GB	5
PC6	20	Core i9	3.70 GHz	8 GB	6

Table 3.2 shows the job specifications in the experiments. A total of 578 jobs with six types will be assigned to workers by the proposal and executed in the UPC system.

Table 3.3 shows the constant CPU time required to start running jobs on each worker for each of the six job types. Tables 3.4-3.6 show the increasing CPU time when the number of jobs is increased by one until the number for the best throughput for each type. It is noted that the preliminary experiments found the number of simultaneously running jobs for the highest throughput at each worker. *PC1*, *PC2*, and *PC3* can run only one job in parallel due to the low specifications. For *PC4*, it is two, for *PC5*, it is five, and for em PC6, it is six.

Table 3.2: Job specifications.

Job type	# of jobs	Ave. job size (KB)	Ave. LOC	Ave. peak mem. use (GB)
BassixAppX1	97	548	1,288	1.80
BassixAppX2	125	623	1,499	1.82
ColorGame	114	177	1,834	1.94
SoccerMatch	88	381	2,632	2.39
AnimalTour	71	31,048	4,625	4.21
MyLibrary	83	409	4,850	2.51
total/ave.	578	5,531	2,788	2.445

Table 3.3: Constant CPU time to start jobs (sec).

Job type	PC1	PC2	PC3	PC4	PC5	PC6
BassixAppX1	9	6	6	5	4	4
BassixAppX2	9	6	6	5	4	4
ColorGame	9	6	6	5	4	4
SoccerMatch	10	6	6	6	5	4
AnimalTour	18	16	16	13	9	8
MyLibrary	11	7	7	6	5	4

Table 3.4: Increasing CPU time at *PC1~PC4* (sec).

Job type	PC1	PC2	PC3	PC4: 1 job	PC4: 2 jobs
BassixAppX1	58	37	37	25	32
BassixAppX2	38	24	24	15	21
ColorGame	60	35	35	25	31
SoccerMatch	128	71	71	46	56
AnimalTour	301	58	58	37	46
MyLibrary	119	43	43	27	34

Table 3.5: Increasing CPU time at *PC5* (sec).

Job type	1 job	2 jobs	3 jobs	4 jobs	5 jobs
BassixAppX1	18	21	25	27	31
BassixAppX2	11	13	16	19	22
ColorGame	16	19	22	26	30
SoccerMatch	31	37	43	55	62
AnimalTour	25	29	50	67	79
MyLibrary	17	20	32	41	47

3.3.2 Comparative Algorithms

For performance comparisons, I implemented the three existing algorithms of assigning *non-uniform jobs* to workers.

The first one is the *First Come First Serve (FCFS)* algorithm. It assigns each job to the first available worker, starting from the worker with the highest specification until the lowest. It limits

Table 3.6: Increasing CPU time at *PC6* (sec).

Job type	1 job	2 jobs	3 jobs	4 jobs	5 jobs	6 jobs
BassixAppX1	16	17	20	23	27	31
BassixAppX2	9	10	12	15	18	21
ColorGame	15	17	19	22	24	28
SoccerMatch	27	31	36	44	54	61
AnimalTour	23	26	30	35	38	44
MyLibrary	16	18	21	27	33	39

that any worker executes only one job at a time.

The second one is the *best throughput based FCFS (T-FCFS)* algorithm. The difference from *FCFS* is that each worker may execute multiple jobs simultaneously for the best throughput.

The third one is the *job scheduling algorithm considering CPU core utilization (CORE)* in [6]. It classifies the jobs by the number of threads and the workers by the number of cores into two groups, and assign the jobs to the workers in the same group using a heuristic local search method.

3.3.3 Makespan Results

Table 3.7 compares the maximum *makespan* results when the testbed UPC system run the jobs by following the assignments by the four algorithms. Tables 3.8-3.10 show *makespan* or the total CPU time of each individual worker by the assignment of the algorithms except for *CORE*.

Table 3.7: Maximum *makespan* results (sec).

Job type	FCFS	T-FCFS	CORE	Proposal
BassixAppX1	536	268	N/A	221
BassixAppX2	470	235	N/A	184
ColorGame	621	276	N/A	233
SoccerMatch	828	414	N/A	370
AnimalTour	666	319	N/A	289
MyLibrary	520	260	N/A	238
total	3,641	1,772	3,965	1,535

Table 3.8: FCFS *makespan* detail (sec).

Job type	PC1	PC2	PC3	PC4	PC5	PC6
BassixAppX1	536	516	516	510	506	500
BassixAppX2	470	450	450	440	435	442
ColorGame	621	574	574	570	560	570
SoccerMatch	828	770	770	780	792	775
AnimalTour	638	666	666	650	612	620
MyLibrary	520	500	500	462	462	480

Table 3.9: T-FCFS *makespan* detail (sec).

Job type	PC1	PC2	PC3	PC4	PC5	PC6
BassixAppX1	268	215	215	222	210	241
BassixAppX2	235	210	210	208	208	214
ColorGame	276	246	246	252	258	256
SoccerMatch	414	385	385	372	377	390
AnimalTour	319	296	296	295	298	312
MyLibrary	260	250	250	240	260	246

Table 3.10: Proposal *makespan* detail (sec).

Job type	PC1	PC2	PC3	PC4	PC5	PC6
BassixAppX1	183	191	191	197	190	221
BassixAppX2	161	174	174	173	180	184
ColorGame	189	216	216	222	233	228
SoccerMatch	266	361	361	342	358	370
AnimalTour	0	248	248	289	246	272
MyLibrary	130	222	222	210	234	238

3.3.4 Discussion

When the results by *FCFS* and *T-FCFS* are compared in Table 3.7, the latter results become less than half of the former results. This difference comes from multiple simultaneous job executions for the best throughput at *PC4* to *PC6*. Thus they are important to reduce *makespan*.

The results by *CORE* are worst among the four algorithms, although it outperformed for *non-uniform jobs*. *CORE* is not suitable for assigning a lot of *uniform jobs*.

The results by the proposal are the best among them. The proposal efficiently assigns the given jobs to the workers considering the performances of them.

It is noted that in Table 3.10, *makespan* at *PC1* is smaller than that at other workers. Especially, no job was assigned at *AnimalTour*. At this worker, as shown in Table 3.4, the increasing CPU time with the number of assigned jobs is much larger than that at other workers. If another job is assigned to this worker, the maximum *makespan* of the system will be increased, which should be avoided.

3.4 Summary

In this chapter, I proposed the static *uniform job* assignment algorithm to workers in the UPC system. *Linear equations* are derived to find the assignment minimizing the maximum *makespan* among the workers, such that the CPU time to complete the assigned jobs becomes equal between the workers. For evaluations, the *software testing* program in *Android programming learning assistance system (APLAS)* was selected, where a lot of source codes from students will be tested. The experiment results for 578 jobs on six workers in the testbed UPC system confirmed the effectiveness of the proposal. In the next chapter, I will present the static *uniform job* assignment algorithm and its extension in the UPC system using simultaneous linear equations.

Chapter 4

Extension of Static Assignment Algorithm for Concurrent Multi-Type Uniform Job Allocation

In this chapter, I present the static assignment *uniform job* algorithm and its extension in the UPC system. The proposal addresses the scenario where *uniform jobs* of various types are assigned concurrently, deriving modified multiple simultaneous linear equations to consider different uniform-job applications, including *OpenPose*, *OpenFOAM*, and *APLAS*. Then, the proposal finds the *lower bound* on *makespan* where every worker requires the same CPU time to complete the assigned jobs.

4.1 Three Uniform Job Applications

In this section, I review the *OpenPose* and *OpenFOAM*. *APLAS* was reviewed in chapter 3.1.

4.1.1 OpenPose

Fisrt, I review *OpenPose*. It has been developed by researchers at Carnegie Mellon University and is an popular open-source software for real-time human pose estimation [15]. It extracts the feature points, called *keypoints*, of the human body in the given image using *Convolutional Neural Network (CNN)*. The *keypoints* represent the important joints in a human body, the contours of eyes, lips in the face, fingertips, and joints in the hands and feet. Using the *keypoints*, the shapes of a body, face, hands, and feet can be described. Since it has been developed based on CNN, the CPU time is very long when computed on a conventional PC.

OpenPose is used in our group for developing the *exercise and performance learning assistant system (EPLAS)* to assist practicing exercises or learning performances by themselves at home [16]. *EPLAS* offers video content of *Yoga* poses by instructors whose performances should be followed by users. During the practice, it automatically takes photos of important scenes of the user. Then, it extracts the keypoints of the human body using *OpenPose* to rate the poses in the photos by comparing the coordinates of them between the user and the instructor.

4.1.2 OpenFOAM

Then, I review *OpenFOAM*. It is an open-source software for the *computational fluid dynamics (CFD)* simulations and has been developed primarily by *OpenCFD Ltd.* It has an extensive range of features to solve anything from complex fluid flows involving chemical reactions, turbulence, and heat transfer, to acoustics, solid mechanics, and electromagnetics [17]. Furthermore, the optimal parameter selection is critical for the high accuracy of the results, and it needs a lot of iterations of selecting parameters in *OpenFOAM* and running it with the parameter values. We applied the *parameter optimization method* for *OpenFOAM* [18]; it needs to run *OpenFOAM* with a lot of different parameters.

Meanwhile, it is also applied for developing the *air conditioning guidance system* [19] in our research. The estimation or prediction of the distributions of the temperature or humidity inside a room using this simulation model is necessary to properly control the air conditioner. By estimating the room environment changes under various actions, it will be possible to decide when the air conditioner is turned on or off. Even the timing to open or close windows in the room can be selected. To estimate or predict the distributions in a room together with sensors, the CFD simulation using *OpenFOAM* has been investigated. Then, the optimization of the parameters in *OpenFOAM* is critical in order to fit the simulation results well with the corresponding measured ones.

4.2 Proposal of Static Uniform Job Assignment Algorithm

In this section, I present the static *uniform job* assignment algorithm to workers in the UPC system.

4.2.1 Objective

To design the algorithm, it is observed that when the *makespan* of every worker becomes equal, the objective of the problem on the *makespan* minimization can be achieved. Otherwise, the maximum *makespan* can be reduced by moving some jobs at the bottleneck worker which determines this maximum *makespan* to other workers, if the number of assigned jobs to any worker can take a real number. Only when every worker has the same *makespan*, the maximum *makespan* cannot be reduced.

$$minimize\{max(m_w^t)\} \; for \; t \in T, w \in W \tag{4.1}$$

The minimization of the maximum *makespan* among all the workers is given as the objective of the problem, where *makespan* m_w^t at worker w for type t is given by the summation of the CPU time for preparation and execution.

4.2.2 Simultaneous Linear Equations

In this paper, the following *simultaneous linear equations* have been derived to find the optimal job-worker assignment, such that the estimated CPU time required to complete the assigned jobs becomes equal among all the workers. The solutions of the *simultaneous linear equations* will be the *lower bound* on *makespan*. Since the solutions become real numbers in general, the integer

number of assigned jobs to each worker should be introduced to them.

$$C_i^t + \frac{R_{i,D_i}^t}{D_i} \times x_i^t = C_j^t + \frac{R_{j,D_j}^t}{D_j} \times x_j^t \\ for\ i \neq j, i \in W, j \in W, t \in T. \tag{4.2}$$

To satisfy the objective of the equal CPU time among the workers, $R_{w,D_w}^t / D_w$ gives the best CPU time to solve one job at worker w by running D_w jobs.

4.2.3 Problem Formulation

To present the static *uniform jobs* assignment algorithm to workers in the UPC system, the problem to be solved is formulated here.

Variables

The following variables are defined for the problem to be solved:

- t: Particular job type;
- w: Particular worker;
- x_w^t: # of the assigned jobs to worker w for type t;
- m_w^t: *Makespan* at worker w to complete all the assigned jobs for type t;
- d_w: # of running jobs in parallel using multi-threads at worker w.

Constants

The following constants are given as the inputs to this problem:

- T: Set of job types;
- W: Set of workers;
- N^t: Total # of jobs for type t;
- D_w: # of jobs for the best throughput at worker w for any type;
- C_w^t: CPU time at worker w to prepare job executions for type t;
- $R_{w,d}^t$: CPU time at worker w to execute d jobs for type t in parallel.

Here, D_w represents the number of simultaneously running jobs for job type t at worker w, which maximizes the number of completed jobs per unit time. This is constant for any job type in each application, because it depends on the common program in the application for every job type.

C_w^t represents the CPU time required to initiate the execution of the program at worker w. For example, in the *code testing* application, it represents the CPU time to initiate the *Gradle Wrapper* daemon and generate *shadow objects* that are necessary to run the *code testing function*.

$R_{w,d}^t$ can be measured using any worker by running jobs for job type t while increasing the number of running jobs in parallel from 1 until D_w.

Constraints

The following two constraints must be satisfied in the problem:

- The total number of the assigned jobs to workers must be equal to N^t for any type t.

$$\sum_{w \in W} x_w^t = N^t \ (t \in T) \tag{4.3}$$

- Any worker cannot run d jobs in parallel when d is larger than the D_w (let d_w for worker w) due to the PC specifications.

$$d_w \leq D_w \tag{4.4}$$

4.2.4 Conditions for Uniform Job Assignment

For the *uniform job* assignment to workers in the UPC system, the following conditions are assumed:

- Several job types may exist for *uniform jobs* in each application, where different job types may need the different CPU time, memory size, and number of CPU cores due to the differences in data;
- Each job is fully executed on one worker until it is completed;
- Each worker may have different performance specifications from the others;
- Each worker may have a different number of running jobs in parallel, using multi-threads for the best throughput;
- The CPU time to run the certain number of jobs in parallel is given for each worker and job type.

4.2.5 Static Uniform Job Assignment Algorithm

Here, I note that the CPU time may be different depending on the number of running jobs in parallel in each worker that has multiple cores. To reduce the CPU time by increasing the job completion throughput, D_w jobs of type t should run at worker w as much as possible, since it will give the best throughput. Based on this observation, I present the three-step static *uniform job* assignment algorithm. Figure 4.1 shows the flowchart of the proposal.

First Step

By solving the *simultaneous linear equations* composed of (4.2) and (4.3), the optimal number of assigned jobs of type t to worker w, $\hat{x}_w^t$, is obtained, assuming that any real value is acceptable for it.

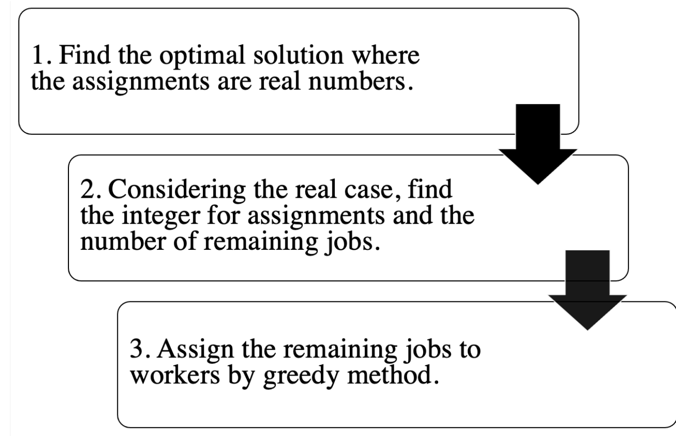

Figure 4.1: Flowchart of the proposal.

Second Step

The solution in the first step becomes feasible only when $\hat{x_w^t}$ is a multiple of D_w for type t. Unfortunately, $\hat{x_w^t}$ does not satisfy the condition, in general. Therefore, in the second step, as the closest integer number to satisfy the condition, the following $\tilde{x_w^t}$ jobs will be assigned to the worker (worker w), where $\lfloor y \rfloor$ gives the largest integer equal to or smaller than y:

$$\tilde{x_w^t} = \lfloor \frac{\hat{x_w^t}}{D_w} \rfloor \times D_w \tag{4.5}$$

Then, the number of the remaining jobs (let r^t for type t) is calculated by:

$$r^t = N^t - \sum_{w \in W} \tilde{x_w^t} \tag{4.6}$$

Besides, the estimated *makespan* for each worker (let em_w^t for worker w and job type t) after the job assignment is calculated by:

$$em_w^t = C_w^t + R_{w,D_w}^t \times \frac{\tilde{x_w^t}}{D_w} \tag{4.7}$$

Therefore, after completing the procedures for all the job types, the estimated *makespan* for each worker is calculated by:

$$EM_w = \sum_{t \in T} em_w^t \tag{4.8}$$

As the objective of the algorithm, the maximum estimated *makespan* among the workers is calculated by:

$$EM = \{max(EM_w)\} \; for \; w \in W \tag{4.9}$$

Third Step

In the third step, the remaining jobs (r_t) in the second step will be assigned to workers in a greedy way, such that the increase in the maximum estimated *makespan* *EM* is minimized. It is noted that the remaining jobs may exist for any job type. Here, to utilize the parallel job computation using multiple threads on multiple cores for each worker as much as possible, the simultaneous assignment of multiple jobs to one worker is always considered.

1. Find the worker whose $E\hat{M}_w$ is smallest among the workers (let worker w).

$$E\hat{M}_w = EM_w + R^t_{w,D_w} \tag{4.10}$$

2. Assign Δx^t_w jobs to worker w.

$$\Delta x^t_w = \begin{cases} D_w, & r_t > D_w \\ r_t, & r_t \leq D_w \end{cases} \tag{4.11}$$

3. Update the number of the remaining jobs (r_t), and the number of assigned jobs and *makespan* of the worker w by:

$$\begin{aligned} x^t_w &= x^t_w + \Delta x^t_w, \\ EM_w &= EM_w + R^t_{w,\Delta x^t_w}, \\ r_t &= r_t - \Delta x^t_w \end{aligned} \tag{4.12}$$

4. If the number of the remaining jobs becomes zero ($r_t = 0$), terminate the procedure.

5. Go to 1.

4.3 Evaluation

In this section, I evaluate the proposal through extensive experiments which are running jobs in three applications on the testbed UPC system.

4.3.1 Testbed UPC System

Table 4.1 shows the PC specifications in the testbed UPC system. One master and six workers are used here.

Table 4.1: PC specifications.

PC	# of Cores	CPU Model	Clock Rate	Memory Size
master	4	Core i5	3.20 GHz	8 GB
PC1	4	Core i3	1.70 GHz	2 GB
PC2	4	Core i5	2.60 GHz	2 GB
PC3	4	Core i5	2.60 GHz	2 GB
PC4	8	Core i7	3.40 GHz	4 GB
PC5	16	Core i9	3.60 GHz	8 GB
PC6	20	Core i9	3.70 GHz	8 GB

4.3.2 Jobs

Table 4.2 shows the specifications of the jobs for the eight job types in the experiments. For the *code testing* application in *APLAS*, six job types are prepared, where each job type represents one assignment to students in *APLAS*. These job types run the same programs of *JUnit* and *Robolectric*,

but accept many different data of answer source codes and test codes. For the other applications, only one job type is considered.

Table 4.2: Job specifications.

Job Type	**# of Jobs**	**Ave. Job Size (KB)**	**Ave. LOC**	**Ave. Peak Mem. Use (GB)**
BassixAppX1	97	548	1288	1.80
BassixAppX2	125	623	1499	1.82
ColorGame	114	177	1834	1.94
SoccerMatch	88	381	2632	2.39
AnimalTour	71	31,048	4625	4.21
MyLibrary	83	409	4850	2.51
OpenPose	41	62	N/A	2.69
OpenFOAM	32	27	N/A	0.035
total/ave.	651	4159	N/A	2.17

4.3.3 CPU Time

Table 4.3 shows the constant CPU time required to start running the jobs on each worker for each of the six job types. Tables 4.4–4.6 show the increasing CPU time when the number of jobs is increased by one until the number for the best throughput for each type.

Through preliminary experiments, I found the number of simultaneously running jobs for the highest throughput for each worker. For *code testing* in *APLAS*, *PC1*, *PC2*, and *PC3* can run only one job in parallel due to the low specifications. This number is two for *PC4*, five for *PC5*, and six for *PC6*. For *OpenPose*, any worker can only execute one job because it uses a lot of threads to compute CNN. For *OpenFOAM*, for each worker, the CPU time is constant at any number of simultaneously running jobs until it reaches the number of cores in the worker.

Table 4.3: Constant CPU time to start jobs (s).

Job Type	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**	**PC6**
BassixAppX1	9	6	6	5	4	4
BassixAppX2	9	6	6	5	4	4
ColorGame	9	6	6	5	4	4
SoccerMatch	10	6	6	6	5	4
AnimalTour	18	16	16	13	9	8
MyLibrary	11	7	7	6	5	4
OpenPose	10	9	9	8	7	7
OpenFOAM	5	5	5	4	3	3

Table 4.4: Increasing CPU time at *PC1~PC4* (s).

Job Type	PC1	PC2	PC3	PC4: 1 Job	PC4: 2 Jobs
BassixAppX1	58	37	37	25	32
BassixAppX2	38	24	24	15	21
ColorGame	60	35	35	25	31
SoccerMatch	128	71	71	46	56
AnimalTour	301	58	58	37	46
MyLibrary	119	43	43	27	34
OpenPose	70	35	35	26	N/A
OpenFOAM	415	206	206	170	170

Table 4.5: Increasing CPU time at *PC5* (s).

Job Type	1 Job	2 Jobs	3 Jobs	4 Jobs	5 Jobs
BassixAppX1	18	21	25	27	31
BassixAppX2	11	13	16	19	22
ColorGame	16	19	22	26	30
SoccerMatch	31	37	43	55	62
AnimalTour	25	29	50	67	79
MyLibrary	17	20	32	41	47
OpenPose	22	N/A	N/A	N/A	N/A
OpenFOAM	128	128	128	128	128

Table 4.6: Increasing CPU time at *PC6* (s).

Job Type	1 Job	2 Jobs	3 Jobs	4 Jobs	5 Jobs	6 Jobs
BassixAppX1	16	17	20	23	27	31
BassixAppX2	9	10	12	15	18	21
ColorGame	15	17	19	22	24	28
SoccerMatch	27	31	36	44	54	61
AnimalTour	23	26	30	35	38	44
MyLibrary	16	18	21	27	33	39
OpenPose	21	N/A	N/A	N/A	N/A	N/A
OpenFOAM	106	106	106	106	106	106

4.3.4 Comparative Algorithms

For performance comparisons, I implemented two simple algorithms to assign *non-uniform jobs* to workers.

The first one is the *First-Come-First-Serve (FCFS)* algorithm. It assigns each job to the first available worker, starting from the worker with the highest specification until the one with the lowest. It limits the worker to executing only one job at a time.

The second is the *best throughput-based FCFS (T-FCFS)* algorithm. The difference between T-FCFS and *FCFS* is that each worker may execute multiple jobs simultaneously until the best throughput.

4.3.5 Total Makespan Results

Table 4.7 compares the maximum *makespan* results for each job type when the testbed UPC system runs the jobs by following the assignments found by the algorithms. Furthermore, it shows the *lower bound (LB)* on the maximum *makespan* found at *First Step* of the proposed algorithm for the reference of them.

Table 4.7: Maximum *makespan* results (s).

Job Type	FCFS	T-FCFS	Proposal	LB
BassixAppX1	536	268	221	203.04
BassixAppX2	470	235	184	178.67
ColorGame	621	276	233	224.04
SoccerMatch	828	414	370	356.03
AnimalTour	666	319	289	262.82
MyLibrary	520	260	238	227.07
OpenPose	272	272	220	209.97
OpenFOAM	1044	131	131	81.55
Total	4957	2175	1886	1743.19

The results indicate that for any job type, the maximum *makespan* result by the proposal is better than the results by the two compared algorithms and is close to the lower bound. Thus, the effectiveness of the proposal is confirmed. It is noted that the results by *FCFS* are far larger than the ones by the others because FCFS does not consider simultaneous multiple job executions for a worker.

4.3.6 Individual Makespan Results

For reference, Tables 4.8–4.10 show *makespan* or the total CPU time of each worker and the largest CPU time difference between the workers and the three algorithms. For *OpenFOAM*, no job was assigned to *PC1–PC4*, because all of the 32 jobs can be executed simultaneously at *PC5* and *PC6*. The largest CPU time difference by the proposal is smaller than the ones by the others, except for *ColorGame*, *SoccerMatch*, *AnimalTour*, and *MyLibrary*, where in Table 4.4, the increasing CPU time of *PC1* is much larger than other workers, and the far smaller number of jobs was assigned. Therefore, the proposal can balance well the job assignments among the workers.

Table 4.8: FCFS *makespan* detail (s).

Job Type	PC1	PC2	PC3	PC4	PC5	PC6	Diff.
BassixAppX1	536	516	516	510	506	500	36
BassixAppX2	470	450	450	440	435	442	35
ColorGame	621	574	574	570	560	570	61
SoccerMatch	828	770	770	780	792	775	58
AnimalTour	638	666	666	650	612	620	54
MyLibrary	520	500	500	462	462	480	58
OpenPose	240	264	264	272	261	252	32
OpenFOAM	840	844	844	1044	917	981	204

Table 4.9: T-FCFS *makespan* detail (s).

Job Type	PC1	PC2	PC3	PC4	PC5	PC6	Diff.
BassixAppX1	268	215	215	222	210	241	58
BassixAppX2	235	210	210	208	208	214	27
ColorGame	276	246	246	252	258	256	30
SoccerMatch	414	385	385	372	377	390	42
AnimalTour	319	296	296	295	298	312	24
MyLibrary	260	250	250	240	260	246	20
OpenPose	240	264	264	272	261	252	32
OpenFOAM	0	0	0	0	131	109	131

Table 4.10: Proposal *makespan* detail (s).

Job Type	PC1	PC2	PC3	PC4	PC5	PC6	Diff.
BassixAppX1	183	191	191	197	190	221	38
BassixAppX2	161	174	174	173	180	184	23
ColorGame	189	216	216	222	233	228	44
SoccerMatch	266	361	361	342	358	370	104
AnimalTour	0	248	248	289	246	272	289
MyLibrary	130	222	222	210	234	238	108
OpenPose	220	219	219	216	205	196	24
OpenFOAM	0	0	0	0	131	109	131

4.3.7 Discussions

The results in Table 4.7 show improvements of maximum *makespan* results by the proposed algorithm if compared with *T-FCFS*. However, some differences can be observed against the *lower bound*.

The current algorithm can find the assignment of some remaining jobs to workers, and assign an integer number of jobs to any worker in a greedy way, after the real number solutions are obtained by solving the *simultaneous linear equations*. A greedy method is usually difficult to give a near-optimum solution, since it only considers the local optimality under the current assignment.

To improve the solution quality, a local search method using iterations has often been adopted for solving combinatorial optimization problems, including this study. Therefore, I will study the use of a local search method for the remaining job assignment in the proposed algorithm.

4.4 Extension to Multiple Job Types Assignment

In this section, I extend the proposed algorithm to the case when jobs for multiple job types are assigned together.

4.4.1 Algorithm Extension

In *First Step* of the proposed algorithm, the linear equations are modified in this extension to consider the CPU time to complete all the jobs for the plural job types assigned to each worker:

$$\sum_{t\in T}(C_i^t + \frac{R_{i,D_i}^t}{D_i} \times x_i^t) = \sum_{t\in T}(C_j^t + \frac{R_{j,D_j}^t}{D_j} \times x_j^t) \quad for\ i \neq j, i \in W, j \in W. \tag{4.13}$$

The number of variables to be solved is $|W||T|$, where $|W|$ represents the number of workers and $|T|$ represents the number of job types, respectively. Thus, $|W||T|$ linear equations are necessary to solve them. In the original algorithm, for each job type, $(|W|-1)$ linear equations are derived for the CPU time equality and one equation is for the job number. Thus, $|W||T|$ equations can be introduced.

However, in this extension, the total number of linear equations for the CPU time equality is reduced to $(|W|-1)$ because all the job types need to be considered together here. Therefore, to solve the linear equations uniquely, the following $(|W|-1)(|T|-1)$ linear equations will be introduced by considering the total CPU time for $(|T|-1)$ job types together for $(|T|-1)$ combinations of $(|T|-1)$ job types, in addition to the total CPU time for $|T|$ job types together in (4.13):

$$\sum_{t\in T-\{u\}}(C_i^t + \frac{R_{i,D_i}^t}{D_i} \times x_i^t) = \sum_{t\in T-\{u\}}(C_j^t + \frac{R_{j,D_j}^t}{D_j} \times x_j^t) \quad for\ i \neq j, i \in W, j \in W, u \in T. \tag{4.14}$$

where $T-\{u\}$ represents the set of the job types in T except for job type u.

The $(|T|-1)$ combinations of $(|T|-1)$ job types are selected by excluding the combination where the following estimated total CPU time to execute all the jobs in the remaining job types on *PC6* is smallest:

$$\sum_{t\in T-\{u\}}(C_6^t + \frac{R_{6,D_6}^t}{D_6} \times N^t) \tag{4.15}$$

Then, in *Second Step* and *Third Step*, the estimated *makespan* for each worker and the maximum estimated *makespan* among the workers are modified to consider all the given job types together.

4.4.2 Total Makespan Results

Table 4.11 shows the maximum *makespan* results when the testbed UPC system runs the jobs by following the assignments by the extended algorithm. When compared with the result by the original algorithm, it is reduced by 5%, and becomes closer to the *lower bound*. The difference between the result and the lower bound is very small. Thus, this extension is effective when plural job types are requested at the UPC system together.

Table 4.11: Maximum *makespan* results (s) by proposal.

Original	Extended	LB
1886	1799	1743.19

4.4.3 Discussions

The result in Table 4.11 confirms some reduction in the total *makespan* result by the extended algorithm. However, there is still a difference when compared to the *lower bound*. Thus, it is necessary to further improve the algorithm.

One idea for this improvement in the extended algorithm will not be to limit the exclusion of one job type combination—where the estimated total CPU time to execute all jobs in the remaining job types on *PC6* is the smallest—and to generate the linear equations for the CPU time equality. Instead, every combination will be excluded one by one to obtain the result for each combination exclusion. Then, the best one will be selected among them.

4.5 Summary

In this chapter, I proposed the static *uniform job* assignment algorithm and its extension for different *uniform job* types in the UPC system. The proposal addresses the scenario where *uniform jobs* of various types are assigned concurrently, deriving modified multiple simultaneous linear equations to consider different *uniform job* applications, including *OpenPose*, *OpenFOAM*, and *APLAS*. Then the proposal finds the *lower bound* on *makespan* where every worker requires the same CPU time to complete the assigned jobs. For an evaluation, the 651 *uniform jobs* in three applications, *OpenPose*, *OpenFOAM*, and *code testing* in *APLAS*, were considered to run on six workers in the testbed UPC system, and the *makespan* was compared with the results by two simple algorithms and the lower bounds. The comparisons confirmed the effectiveness of the proposal. In the next chapter, I will present an extension of static assignment algorithm for *uniform jobs* with multiple CPU time in the UPC system.

Chapter 5

Extension of Static Assignment Algorithm for Uniform Jobs with Multiple CPU Time

In this chapter, I present an extension of the second static *uniform job* assignment algorithm, in order to handle *uniform jobs* whose CPU times are multiples of the CPU time of the *unit job*. The job who has the shortest CPU time is called the *unit job* for convenience. In physics or network simulations, which are typical scenarios, the CPU time is proportional to the inputs of the program such as the number of meshes or the number of iteration steps. In the proposal, the necessary number of *unit jobs* to satisfy the CPU time are prepared for each input job.

5.1 OpenFOAM and NS-3

In this section, I review the *Network Simulator 3 (NS-3)*. *OpenFOAM* was reviewed in chapter 4.1.2.

NS-3 is an open-source discrete-event network simulator developed primarily for research and educational purposes in the field of networking. It allows researchers to model and simulate various network protocols, internet systems, and wired and wireless networks. *NS-3* provides a comprehensive set of tools and libraries for simulating network topologies, protocols, and traffic patterns. It supports a wide range of network protocols, including IP, TCP, UDP, and various routing protocols. One of the key features of *NS-3* is its use of C++ for core simulation models, combined with Python bindings to facilitate easier scripting and configuration. This dual-language approach provides both high performance and easy use, enabling detailed and complex simulations to be built and executed efficiently. These make it an invaluable tool for researchers and educators to evaluate and test new networking ideas or analyze the performance of existing protocols under different conditions [20].

5.2 Proposal of Algorithm Extension

In this section, I review the previous algorithm and present the extension.

5.2.1 Previous Algorithm

In the previous algorithm, first, the following *simultaneous linear equations* are solved to find the job-worker assignment such that the estimated CPU time for completing the assigned jobs at every

worker becomes equal. This real number solution will be the *lower bound* on *makespan*. Next, the real number of assigned jobs is converted to an integer number close to it.

$$\sum_{w \in W} x_w^t = N^t \ (t \in T). \tag{5.1}$$

$$C_i^t + \frac{R_{i,D_i}^t}{D_i} \times x_i^t = C_j^t + \frac{R_{j,D_j}^t}{D_j} \times x_j^t \quad for\ i \neq j, i \in W, j \in W, t \in T, \tag{5.2}$$

where N^t is the total # of jobs for type t, C_w^t is the CPU time at worker w to prepare job executions for type t, $R_{w,d}^t$ is the CPU time at worker w to execute d jobs for type t in parallel, D_w is number of jobs for the best throughput at worker w for any type, x_w^t is the number of the assigned jobs to worker w for type t, T is the set of job types, and W is the set of workers.

5.2.2 Extension of Algorithm

The algorithm is extended to the case when each *uniform job* can have a multiple of the CPU time for the *unit job* or the *unit time*. Let k be the multiple of the *unit time* for a job that requires *k unit time*. Then, *k unit jobs* with the *unit time* will be assigned to a worker for this job. Therefore, the total # of jobs to be assigned by the algorithm is replaced by:

$$N^t = \sum_{k \in K^t} k \times N_k^t \tag{5.3}$$

where K^t is the set of CPU time multiples and N_k^t is the # of jobs requiring *k unit time* for type t.

Next, the real number solution $\hat{x_{w,k}^t}$ of the *simultaneous linear equations* is converted to the integer number by using the greedy algorithm presented in the chapter 4.2.5 . Here, k is used instead of D_w, and the job type requiring the longest CPU time is first assigned to the fastest worker.

5.3 Evaluation

In this section, I evaluate the proposal through experiments using the testbed UPC system.

5.3.1 Experiment Setup

Table 5.1 shows the specifications of the six worker PCs in experiments. In *OpenFOAM* and *NS-3* applications, the program runs with three different physics or network parameter sets. Table 5.2 shows the constant CPU time required to start execute jobs on each worker. Table 5.3 shows the number of jobs, and the CPU time of the best throughput for each job type at each worker PC. Although the large CPU time may not be an exact multiple of the shortest time, the difference is very small.

Table 5.1: PC Specifications.

PC	# of cores	CPU	clock rate	memory size	best throughput
PC1	4	Core i3	1.70 GHz	2 GB	1
PC2	4	Core i5	2.60 GHz	2 GB	1
PC3	4	Core i5	2.60 GHz	2 GB	1
PC4	8	Core i7	3.40 GHz	4 GB	2
PC5	16	Core i9	3.60 GHz	8 GB	5
PC6	20	Core i9	3.70 GHz	8 GB	6

Table 5.2: Constant CPU time to prepare jobs (sec).

job type	PC1	PC2	PC3	PC4	PC5	PC6
OpenFOAM	10	9	9	7	5	4
NS-3	9	8	8	6	4	3

5.3.2 Makespan Results

Table 5.4 shows maximum *makespan* results. The *Thread-based First Come First Serve (T-FCFS)* and the previous algorithms are used for comparisons. *T-FCFS* is applied to each job type one by one. The previous algorithm and the proposal are applied to all the job types at once. The results indicate that the total *makespan* by the proposal is smaller than them. Thus, the effectiveness of the proposal is confirmed.

5.3.3 Assignments Results

Tables 5.5 and 5.6 show the number of assigned jobs to each worker by the previous algorithm and the proposal respectively. The previous algorithm assigns the jobs in each job type to all the workers because it basically finds the assignment by the job type. The proposal assigns more heavy jobs to faster workers.

Table 5.3: Required CPU time for best throughput (sec).

job type	# of jobs	PC1	PC2	PC3	PC4	PC5	PC6
OpenFOAM-1	55	415	206	206	170	128	106
OpenFOAM-2	78	274	134	134	113	87	71
OpenFOAM-3	43	138	68	68	56	42	35
NS-3-A	64	389	201	201	183	132	103
NS-3-B	102	194	100	100	91	66	51
NS-3-C	95	97	50	50	45	33	25

Table 5.4: Total *makespan* (sec).

<table>
<tr><th colspan="2">job type</th><th>T-FCFS</th><th>previous</th><th>proposal</th></tr>
<tr><td rowspan="3">OpenFOAM</td><td>-1</td><td>532</td><td rowspan="7">2288</td><td rowspan="7">2238</td></tr>
<tr><td>-2</td><td>525</td></tr>
<tr><td>-3</td><td>156</td></tr>
<tr><td rowspan="3">NS-3</td><td>-A</td><td>636</td></tr>
<tr><td>-B</td><td>486</td></tr>
<tr><td>-C</td><td>232</td></tr>
<tr><td colspan="2">total</td><td>2567</td></tr>
</table>

Table 5.5: Number of assigned jobs by previous.

job type	PC1	PC2	PC3	PC4	PC5	PC6
OpenFOAM-1	1	2	2	6	20	24
OpenFOAM-2	1	3	3	8	25	38
OpenFOAM-3	1	2	2	8	15	15
NS-3-A	1	2	2	4	20	35
NS-3-B	2	6	6	8	35	45
NS-3-C	3	3	3	14	30	42

Table 5.6: Number of assigned jobs by proposal.

job type	PC1	PC2	PC3	PC4	PC5	PC6
OpenFOAM-1	1	0	0	0	0	54
OpenFOAM-2	1	0	0	16	55	6
OpenFOAM-3	1	15	15	4	2	6
NS-3-A	0	0	0	4	0	60
NS-3-B	0	0	0	0	90	12
NS-3-C	11	22	22	34	0	6

5.4 Summary

In this chapter, I proposed an extension of the second static assignment algorithm in this thesis to *uniform jobs* whose CPU time are multiple of the shortest one in the UPC system. This job is called the *unit job* for convenience. For evaluations, I prepared 176 *OpenFOAM* jobs and 261 *NS-3* jobs to assign to six workers in the UPC system. The results showed that the proposed assignment algorithm reduced the *makespan* compared to the second static assignment algorithm and others, demonstrating its effectiveness. In the next chapter, I will present a design and implementation of a stationery product recognition method using the latest *YOLOv8* model at two stages.

Chapter 6

Proposal of Stationery Product Recognition Method Using Two-Stage YOLOv8

In this chapter, I present a design and implementation of a stationery product recognition method using the *YOLOv8* model at two stages. In order to reduce the retraining time and improve the accuracy, the first-stage model is applied to recognize the category of the target object from the given image and the second-stage models recognize the product name/type among those in the category.

6.1 YOLO Family

In this section, I review the history of *YOLO* Family. Originally, the *YOLO model* was proposed by Joseph Redmon and Ali Farhadi in 2015 [21], as a real-time object detection system based on *CNN (Convolutional Neural Network)*. Subsequently, in December 2016, they unveiled *YOLOv2*, which not only enhanced the accuracy but also increased its computing speed [22]. In 2018, they presented *YOLOv3*, which demonstrates further advancements in the object detection performance [23]. However, Joseph Redmon exited the computer vision research due to concerns about military applications and privacy violations. Yet, *YOLO* researches in the field continued, resulting in the ongoing development of a substantial *YOLO* family. This section outlines the brief history of the *YOLO* family from *YOLOv1* to the latest *YOLOv8* for tracing its evolution over time.

YOLOv1 Unlike the previous method, *YOLOv1* simultaneously identifies all the bounding boxes by partitioning the input image into a $S \times S$ grid and making predictions for B bounding boxes with confidence scores pertaining to C classes within each grid cell. The result is the tensor of dimensions $S \times S \times (B \times 5 + C)$, and this output can undergo the non-maximum suppression to eliminate redundant detection. Figure 6.1 shows the basic idea of the original *YOLO*. *YOLOv1* achieved the *mAP* of 63.4% on the *PASCAL VOC2007* dataset [21].

YOLOv2 In contrast to other region proposal-based approaches like *Fast R-CNN*, *YOLOv1* exhibits a higher positioning error and a reduced recall rate. Hence, *YOLOv2*, which released in 2016, primarily focuses on improving the recall rate and the positioning accuracy by batch normalizations, anchor boxes, and dimension clusters. The most important improvement of the architecture was the famous *DarkNet-19*, which contained 19 convolutional layers. *YOLOv2* achieved *mAP* of 78.6% on the *PASCAL VOC2007* dataset [22].

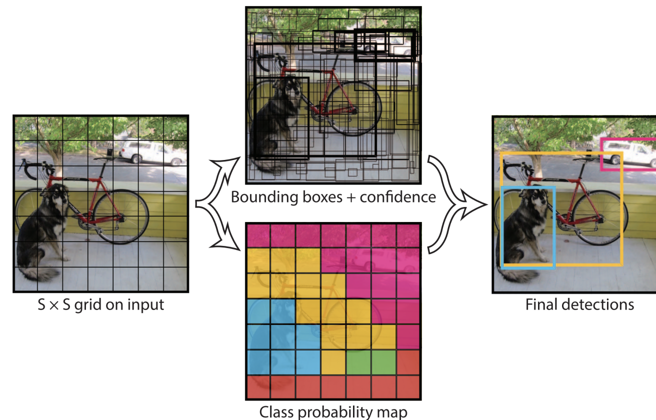

Figure 6.1: *YOLO* base idea.

YOLOv3 *YOLOv3*, which was introduced in 2018, was further improved with the more efficient backbone network, multiple anchors, and the spatial pyramid pooling. This model is excelled in detecting small and densely packed objects, thanks to key enhancements such as replacing *Softmax Loss* with *Logistic Loss*, using nine anchors for improved *IoU*, and employing three detections instead of one. The substitution of *Darknet-19* with *Darknet-53*, which further boosted the object detection accuracy. Starting from *YOLOv3*, the benchmark testing dataset was changed to *MS COCO* dataset and *YOLOv3-spp* achieved *mAP(50)* of 60.6% when the testing image size is 608×608 pixels [24].

YOLOv4 After the left of Joseph Redmon, *YOLOv4* was released in April 2020 by Alexey Bochkovskiy and others. *YOLOv4* kept all the good points from *YOLOv3*, fast, open source, *DarkNet*, and others. By experimenting latest bag-of-freebies and bag-of-specials methods such as mosaic data augmentations, and the modified *SAM YOLOv4* had the capability to finish training and detection in a single GPU and achieved *mAP(50)* of 65.7% on *MS COCO* dataset when testing image size is 608×608 pixels [25].

YOLOv5 In June 2020, two months after the release of *YOLOv4*, *YOLOv5* was lunched by Glen Jocher from *Ultralytics*. The main difference between *YOLOv5* and *YOLOv4* is that *YOLOv5* is implemented by *PyTorch* instead of *DarkNet*. *YOLOv5* became the most popular open-source project in the objective detection filed due to its user-friendly parameter adjusting, formats exporting, and lifecycle deployments. It is mainly maintained by *Ultralytics* and other contributors on *GitHub* and the latest version is *YOLOv5-7.0*. *Ultralytics* proposed multiple scaled models, where the largest model, *YOLOv5x*, can achieve *mAP(50)* of 68.9% and *mAP(50-95)* of 50.7% on *MS COCO* dataset when testing image size is 640×640 pixels [26].

YOLOv6 *YOLOv6* was published by Meituan in 2022. It updates the design of network, including a new backbone, neck and heap. It provides multiple scaled models for Meituan's industrial applications and the largest model, *YOLOv6-L*, achieved *mAP(50)* of 70% on *MS COCO* dataset when the testing image size is 640×640 pixels [27].

YOLOv7 The authors of *YOLOv4* published *YOLOv7* in 2022. *YOLOv7* significantly improved the speed and accuracy from *YOLOv4* by applying the state-of-the-art bag-of-freebies methods

and updating the model network. *YOLOv7* also provides multiple scaled models. Comparing with *YOLOv4*, 43% of the parameters are reduced with no accuracy lose. The largest model *YOLOv7-X* achieved *mAP(50)* of 71.2% on *MS COCO* dataset when testing image size is 640×640 pixels. Meanwhile, *YOLOv7* added additional tasks such as pose estimation on the *MS COCO* key points dataset [28].

YOLOv8 The company releasing *YOLOv5*, which is *Ultralytics*, published *YOLOv8* in early 2023. Based on the user-friendly life-cycle development of *YOLOv5*, *YOLOv8* provides completed functions in the computer version field, encompassing detection, classification, segmentation, pose estimation, and tracking. The differences between *YOLOv8* and *YOLOv5* include advanced network architectures, anchor-free split *Ultralytics* head, and optimized accuracy-speed tradeoff. It also provides multiple scaled models, where the largest model *YOLOv8x* achieved *mAP(50-95)* of 53.9% (*YOLOv5* is 50.7%) when testing image size is 640×640 pixels on *MS COCO* dataset [29].

6.2 Proposal of Stationery Product Recognition Method

In this section, I present the design and implementation of the *stationery product recognition method* using two-stage *YOLOv8* models.

6.2.1 Dataset for YOLO

The *YOLO* training dataset comprises two key components: images, which are typically stored in the JPEG format, and corresponding labels, which are stored in the TXT format. Each label provides annotations of the objects contained within the associated image, establishing a direct correlation between two files. The TXT format is given by ***class x-center y-center width height***. The ***class*** represents the object class number starting from 0. The ***x-center y-center width height*** represents the bounding box coordinate of the object that must be normalized between 0 and 1.

In order to build a reasonable custom dataset, the pro-processing for images is indispensable. Fundamentally, machine learning models perform quicker training for smaller images. When the input image becomes twice as large, which means four times the number of pixels, it leads to significant increase of training time. Moreover, to build a stationery image dataset in this paper, I collected images in different ways. Some were taken by smartphones, or by cameras, or by downloaded from online open datasets. Since the sizes of these images are different, the pro-resizing was applied in the pre-processing. Considering the PC specification, I resized images to 640×640 pixels, which resulted in reasonable training time. For rectangular images, I ensured their existing aspect ratio by adding black padding to them after resizing. Figures 6.2 and 6.3 are an example origin image and its resized image. *Roboflow* [30], an online tool, was used for preparing this dataset. Table 6.1 shows the details of this stationery dataset. Five products were considered for each of nine categories. For each product, 15-20 images were prepared.

6.2.2 Two-Stage YOLO Models

To reduce the retraining time and to improve the accuracy, one *YOLO* model called *category model* recognizes the category of the target object in the given image at the first stage. Then, at the second stage, the *YOLO* model for that category called *product model* recognizes the product. Thus, a total

Figure 6.2: Origin image.

Figure 6.3: Resized image.

Table 6.1: Dataset details.

Category	Products	Images
Ball-pen	5	92
Clip	5	80
Correction-tape	5	98
Dry-cell	5	83
Eraser	5	84
Flash-drive	5	80
Glue-stick	5	99
Pencil	5	94
Stapler	5	85
Total	45	795

of ten models are trained, where one model is for the first stage, and nine models for the second stage.

CLI and Parameters

Considering the specifications of the PC to run the proposal, the following CLI and parameters are used for the transfer training in *YOLOv8*:

```
$ yolo detect train data=data.yaml imgsz=640 batch=16 workers=8\
epochs=300 patience=0 model=yolov8s.pt
```

where

- data: the data file path.
- imgsz: the integer value for image size.
- batch: the number of images per batch.
- workers: worker thread count for data loading.
- epochs: the training duration, measured in the number of epochs.
- patience: epochs to wait for the absence of noticeable progress before implementing early training termination.
- model: the pre-trained model file path.

The values of *data* are different for training different stage models. I used 640 to *imgsz*, 16 to *batch*, and 8 to *workers*. Although the default value of *epochs* is 100, I set it to 300 to fully use the dataset and the power of the PC. Meanwhile, I set *patience* to 0 to disable the early training termination in *YOLO*. *Ultralytics* provides pre-trained *YOLOv8* models on *COCO* dataset. They are called *yolov8n*, *yolov8s*, *yolov8m*, *yolov8l*, and *yolov8x*. Table 6.2 shows the differences between them [29]. Therefore, instead of training the *YOLO* model from scratch, I adopted *yolov8s* to the initial parameters and applied transfer learning, which is more efficient.

Table 6.2: *YOLOv8* pre-trained model details.

Model	**Size (pixels)**	**mAP 50-95**	**Speed CPU ONNX (ms)**	**Speed A100 TensorRT (ms)**	**Params (M)**
YOLOv8n	640	37.3	80.4	0.99	3.2
YOLOv8s	640	44.9	128.4	1.20	11.2
YOLOv8m	640	50.2	234.7	1.83	25.9
YOLOv8l	640	52.9	375.2	2.39	43.7
YOLOv8x	640	53.9	479.1	3.53	68.2

6.3 Evaluation

In this section, I show the evaluation results of the proposal.

6.3.1 Experiments Results

For running the *YOLO* models, I used a PC that is equipped with *NVIDIA RTX 3060* graphics card. Table 6.3 shows the specification of the PC. Table 6.4 shows the mean average precision at 50 (*mAP50*) between the proposal and the conventional *All-in-One* model. Table 6.5 shows the retraining time when one product is added in the *stapler* category.

Table 6.3: PC specification.

CPU	GPU	Mem. size	GPU Mem. Size
i9 10900K	RTX 3060	16 GB	12 GB

Table 6.4: Mean average precision at 50.

Category	Proposal	All-in-One
Ball-pen	99.5%	97.4%
Clip	99.5%	99.5%
Correction-tape	99.5%	99.5%
Dry-cell	99.5%	99.5%
Eraser	99.5%	99.5%
Flash-drive	96.62%	88.67%
Glue-stick	99.5%	99.5%
Pencil	94.35%	95.48%
Stapler	97.21%	99.5%
Average	98.35%	97.61%

Table 6.5: Retraining time Comparison

Method	Retraining Time(m)
Proposal (stapler)	10.2
All-in-One	82.9

6.3.2 Discussion

Table 6.4 shows that the average *mAP(50)* of the proposal is 98.35% which is 0.71% higher than the *all-in-one* model. Nevertheless, the results of the proposal in the categories of pencil and stapler are 94.35% and 97.21%, which are lower than those of *all-in-one*. Table 6.5 shows that the retraining time by the proposal becomes less than one seventh of the result by *all-in-one*. This difference comes from the retraining dataset size between them. Along with increases of categories and products, the retraining time will much increase, which should be noticed.

6.4 Summary

In this chapter, I proposed the design and implementation of the stationery product recognition method using *YOLOv8* models at two stages. The first-stage model recognizes the category of the object and the second-stage models recognize the product from the category. For evaluations, the custom dataset with 795 images of 45 stationery products in 9 categories were built. The experiment results confirmed the effectiveness of the proposal. In the next chapter, I will introduce the related works to this study.

Chapter 7

Related Works in Literature

In this chapter, I review related works in literature to this study, including studies of job assignment algorithms and YOLO models.

In [31], Lin proposed several linear programming models and algorithms for identical jobs (*uniform jobs*) on parallel uniform machines for individual minimizations of several different performance measures. The proposed linear programming models provide structured insights of the studied problems and provide an easy way to tackle the scheduling problems.

In [32], Mallek et al. addressed the problem of scheduling identical jobs (*uniform jobs*) on a set of parallel uniform machines. The jobs are subjected to conflicting constraints modeled by an undirected graph G, in which adjacent jobs are not allowed to be processed on the same machine. The minimization of the maximum *makespan* in the schedule is known to be *NP-hard*. To solve the general case of this problem, they proposed mixed-integer linear programming formulations alongside lower bounds and heuristic approaches.

In [33], Bansal et al. proposed the two-stage *Efficient Refinery Scheduling Algorithm (ERSA)* for distributed computing systems. In the first stage, it assigns a task according to the min–max heuristic. In the second stage, it improves the scheduling by using the refinery scheduling heuristic that balances the loads across the machines and reduces *makespan*.

In [34], Murugesan et al. proposed a multi-source task scheduler to map the tasks to the distributed resources in a cloud. The scheduler has three phases: the task aggregation, the task selection, and the task sequencing. By using the ILP formulation, this scheduler minimizes *makespan* while satisfying the budget allotted by the cloud user based on the divisible load theory.

In [35], Garg et al. proposed the *adaptive workflow scheduling (AWS)* for grid computing using the dynamic resources based on the rescheduling method. The AWS has three stages of the initial static scheduling, the resource monitoring, and the rescheduling, to minimize *makespan* using the directed acyclic graph workflow model for grid computing. It deals with the heterogeneous dynamic grid environment, where the availability of computing nodes and link bandwidths are inevitable due to existences of loads.

In [36], Gawali et al. proposed the two-stage *Standard Deviation-Based Modified Cuckoo Optimization Algorithm (SDMCOA)* for the scheduling of distributed computing systems. In the first stage, it calculates the sample initial population among all the available number of task populations. In the second stage, the modified COA immigrates and lays the tasks.

In [37], Bittencourt et al. reviewed existing scheduling problems in cloud computing and distributed systems. The emergence of distributed systems brought new challenges on scheduling in computer systems, including clusters, grids, and clouds. They defined a taxonomy for task scheduling in cloud computing, namely, pre-cloud schedulers and cloud schedulers, and classified

existing scheduling algorithms in the taxonomy. They introduced future directions for scheduling research in cloud computing.

In [38], Attiya et al. presented a modified *Harris hawks optimization (HHO)* algorithm based on the *simulated annealing (SA)* for scheduling the jobs in a cloud environment. In this approach, SA is employed as a local search algorithm to improve the convergence rate and the solution quality generated by the standard HHO algorithm. HHO is a novel population-based, nature-inspired optimization paradigm proposed by Heidari et al. [39]. The main inspiration of HHO is the cooperative behavior and the chasing style of Harris' hawks in nature. In the HHO model, several hawks explore prey, respectively, and simultaneously after attacking the target from different directions to surprise it.

In [40], Al-Maytami et al. presented a novel scheduling algorithm using *Directed Acyclic Graph (DAG)* based on the *Prediction of Tasks Computation Time algorithm (PTCT)* to estimate the preeminent scheduling algorithm for prominent cloud data. The proposed algorithm provides a significant improvement with respect to *makespan* and reduces the computational complexity via employing *Principal Components Analysis (PCA)* and reducing the *Expected-Time-to-Compute (ETC)* matrix.

In [41], Panda et al. proposed an *energy-efficient task scheduling algorithm (ETSA)* to address the demerits associated with the task consolidation and scheduling. The proposed algorithm *ETSA* takes into account the completion time and the total utilization of a task on the resources, and follows a normalization procedure to make a scheduling decision. The *ETSA* provides an elegant trade-off between energy efficiency and *makespan*, more so than the existing algorithms.

In [42], Terven et al. offered a comprehensive review of evolution from the original *YOLO* to *YOLOv8* and other versions. They introduced the brief history and analyzed the advancements and contributions of each version in *YOLO* family. They started by providing an overview of the standard metrics and post-processing techniques. Following them, they explored the notable changes in the network architecture and training strategies unique to each model. To sum up, they brought together essential learnings acquired from *YOLO's* development, presenting a viewpoint on its future and highlighting possible research paths to enhance real-time object detection systems.

In [43], Jiang et al. delivered a concise overview of *YOLO* and its subsequent advanced versions. Through abundant analysis, they shared numerous observations and insightful findings. The results emphasized differences and similarities among different *YOLO* versions and between *YOLO* and *CNNs*. The main point is that continuous enhancements are underway for *YOLO*. Moreover, the article outlines the developmental path of *YOLO*, summarizes methods for the target recognition and the feature selection. Additionally, significant contributions are made to the current literature on *YOLO* and alternative object detection methods.

In [44], Talebi et al. investigated the impact of image size on the accuracy of tasks in the computer vision. Traditional resizers, like bi-linear and bi-cubic, are found to limit task performances. They proposed learned resizers as replacements, emphasizing their ability to substantially improve task performances rather than visual qualities. These resizers are jointly trained with baseline vision models, focusing on the *ImageNet* classification task with four different models. The experiments demonstrated consistent improvement in task metrics and proved usefulness for fine-tuning classification baselines for other vision tasks.

In [45], Lu et al. focused on leveraging *Unmanned Aerial Vehicles (UAVs)* in intelligent transportation systems, specifically, addressing the challenge of the vehicle detection in an aerial image. They propose a vehicle detection method based on *YOLOv3*. The authors processed three public aerial image datasets to create a tailored dataset for *YOLO* training. Experimental results indicate the model's strong performance on unknown aerial images, especially in detecting small, rotating,

compact, and dense objects, meeting real-time requirements effectively.

In [46], Abhinand et al. proposed a safety alert system for Indian metro stations to address the disregard for safety rules, particularly the prohibition of crossing the yellow line. The system utilizes *YOLOv3* object detection models from stationary surveillance cameras, employing pre-trained models for simplicity. The motion detection involves several steps, including behavior understanding and activity recognition. The yellow line detection uses a color detection method on the first frame due to the static camera. When an object crosses the yellow line, the system triggers an emergency alert through sound signals for authorities and passengers, ensuring a real-time and accurate response to safety violations.

In [47], Niharika et al. explored using object recognition models, specifically comparing three *CNNs* (*Faster Region-Based-CNN*, *YOLOv3*, and *YOLOv4*), to automatically assign the mobile eye-tracking data to real objects in a student lab course. The aim is to simplify the time-consuming analysis of eye-tracking data, with *YOLOv4*, combined with optical flow estimation, proving to be the most efficient and accurate. This automatic assignment facilitates real-time system responses to the user's gaze. They also acknowledge and discusses various problems associated with using object detection for mobile eye-tracking data.

In [48], Zhou et al. proposed a safety helmet detection method based on *YOLOv5* for recognizing a prevalent issue of safety helmet non-compliance due to insufficient awareness. They built a custom dataset containing $6,045$ images, trained four *YOLOv5* models with different parameters, comparing and analyzing them. Experimental results indicate that their average detection speed satisfies real-time detection requirements, and the *mAP* reached 94.7% by adopted pre-trained *YOLOv5x* weights, validating the effectiveness of it.

In [49], Liu et al. delivered an enhanced *YOLOv7* network which utilizes an *ACmixBlock* module to improve the accuracy and speed for the underwater target detection. An additional *ResNet-ACmix* module and a *Global Attention Mechanism (GAM)* are integrated to enhance feature extractions and reduce computations, while the *K-means* algorithm is replaced by the *K-means++* one to obtain anchor boxes. Experimental results indicate that the mean average precision (*mAP*) of the proposal outperforms the comparisons such as the original *YOLOv7*, demonstrating its effectiveness as an underwater target detection method.

In [50], Talaat et al. proposed a *Smart Fire Detection System (SFDS)* based on the *YOLOv8* algorithm, which aims to reduce false alarms, enhance real-time detection, and offer cost-effective solutions for smart cities. The proposal incorporates fog-cloud services and IoT computing for data collections, coupled with an enhanced *YOLOv8* algorithm for real-time responsiveness. Experimental results indicate the state-of-the-art performance of precision, presenting a promising detection system for practical applications.

In [51], Vats et al. focused on integrating artificial intelligence (AI) and computer vision (CV) for automatic checkout in retail industries to solve occlusions, motion blur, and similarity of items. The presented solution *RetailCounter* combines video inpainting with the detection, tracking, and selection modules to accurately recognize, localize, track, and count products in front of a camera. *RetailCounter* operates on a detect-then-track paradigm, incorporates automatic ROI identifications, and removes unwanted objects. *RetailCounter* achieved the fourth place ranking in track 4 of *2023 AI City Challenge*, which proved its competitive performance.

In [52], Luo et al. proposed a small-object detection algorithm called *DC-YOLOv8* for scenarios where traditional camera sensors relying on human observation face the challenges like eye fatigue and cognitive limitations. The innovations include a new down-sampling method to preserve context features, an improved feature fusion network, and a new network structure for enhanced detection accuracy. The algorithm outperforms existing ones (*YOLOX*, *YOLOR*, *YOLOv3*,

scaled YOLOv5, *YOLOv7-Tiny*, and *YOLOv8*) according to experiments on three datasets (*Visdron*, *Tinyperson*, *PASCAL VOC2007*). Their results show higher map, precision, and recall ratios for the proposed algorithm *DC-YOLOv8* across various scenarios.

Chapter 8

Conclusion

This thesis presented the studies of *uniform job* assignment algorithms to workers in the user-PC computing system and the stationery product recognition method using the *YOLOv8* model.

I first presented the static *uniform job* assignment algorithm for the UPC system. To minimize the *makespan* for completing all the jobs in the system, a set of linear equations are derived to actually find the number of jobs assigned to each worker, such that the CPU time to complete the assigned jobs becomes equal between the workers. The results showed that it reduced the *makespan* by up to 13% compared to the FIFO approach.

Next, I presented an extension of this first static *uniform job* assignment algorithm. The proposal addresses the scenario where *uniform jobs* of various types are assigned concurrently, deriving modified multiple simultaneous linear equations to consider different *uniform job* applications. The results demonstrated the *makespan* was reduced by an average of 5% compared to the first static *uniform job* assignment algorithm.

Third, I further extended the second static *uniform job* assignment algorithm to handle *uniform jobs* whose CPU times are multiples of the CPU time of the *unit job*. The job who has the shortest CPU time is called the *unit job* for convenience. The results showed that the proposed assignment algorithm reduced the *makespan* compared to my second static *uniform job* assignment algorithm and others.

Finally, I presented a design and implementation of the stationery product recognition method using the *YOLOv8* model at two stages. In order to reduce the retraining time and improve the accuracy, the first-stage model is applied to recognize the category of the target object from the given image and the second-stage models recognize the product name/type among those in the category. The results showed the difference between the conventional one model case and the proposed two-stage model case, which confirmed the effectiveness of it.

In future studies, I plan to utilize the UPC system to facilitate AI tasks in training and executions. In this context, the assignment algorithm plays a crucial role in efficiently managing these tasks and utilizing computational resources effectively.

Bibliography

[1] H. Htet, N. Funabiki, A. Kamoyedji, M. Kuribayashi, F. Akhter, W.-C. Kao, "An implementation of user-PC computing system using Docker container," Int. J. Future Comput. Commun., vol. 9, no. 4, pp. 66-73, Dec. 2020.

[2] N. Funabiki, K. S. Lwin, Y. Aoyagi, M. Kuribayashi, W.-C. Kao, "A user-PC computing system as ultralow-cost computation platform for small groups," Appl. Theo. Comput. Tech., vol. 2, no. 3, pp. 10-24, Mar. 2017.

[3] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer com-puting system," Future Gen. Com. Sys., vol. 18, no. 4, pp. 561-572, Aug. 2002.

[4] H. Htet, N. Funabiki, A. Kamoyedji, M. Kuribayashi, "Design and implementation of improved user-PC computing system," IEICE Tech. Report, NS2020-28, vol. 120, no. 69, pp. 37-42, Jun. 2020.

[5] A. Mouat, *Using Docker: developing and deploying software with containers*, O'Reilly Media, Inc., Dec. 2015.

[6] A. Kamoyedji, N. Funabiki, H. Htet, M. Kuribayashi, "A proposal of job-worker assignment algorithm considering CPU core utilization for user-PC computing system," Int. J. Future Comput. Commun., 2022, 11, 40–46.

[7] D. Herron, *Node.js web development*, 5th Ed., Packt Pub., Jul. 2020.

[8] "SFTP," Available online: `https://www.ssh.com/ssh/sftp`, (Accessed 13 Apr., 2022).

[9] A. Ratan, E. Chou, P. Kathiravelu, M. O. Faruque Sarker, *Python network programming: conquer all your networking challenges with the powerful python language*, Packt Pub., Jan. 2019.

[10] B. Schwartz, P. Zaitsev, V. Tkachenko, *High performance MySQL: optimization, backups, and replication*, 3rd Ed., O'Reilly Media, Mar. 2012.

[11] R. McKendrick, *Monitoring Docker*, Packt Pub., Dec. 2015.

[12] G. Rodola, "Efficient I/O with zero-copy & psutil," Available online: `https://gmpy.dev/static/efficient-io-with-zerocopy-syscalls.pdf`, (Accessed 13 April 2022).

[13] Y. W. Syaifudin, N. Funabiki, M. Kuribayashi, W.-C. Kao, "A proposal of Android programming learning assistant system with implementation of basic application learning," Int. J. Web Info. Sys., Oct. 2019.

[14] Y. W. Syaifudin, N. Funabiki, M. Mentari, H. E. Dien, I. Mu'aasyiqiin, M. Kuribayashi, W.-C. Kao, "A web-based online platform of distribution, collection, and validation for assignments in Android programming learning assistance system," Eng. Letter., vol. 29, no. 3, pp. 1178-1193, August 2021.

[15] "OpenPose," Available online: Cmu-perceptual-computing-lab.github.io/openpose/web/html/doc/index.html (accessed on 15 August 2022).

[16] I.T. Anggraini, A. Basuki, N. Funabiki, X. Lu, C.-P. Fan, Y.-C. Hsu, C.-H. Lin, "A proposal of exercise and performance learning assistant system for self-practice at home," Adv. Sci. Technol. Eng. Syst. J. 2020, 5, 1196–1203.

[17] "OpenFOAM," Available online: www.openfoam.com (accessed on 15 August 2022).

[18] Y. Zhao, K. Kojima, Y.-Z. Huo, N. Funabiki, "CFD parameter optimization for air-conditioning guidance system using small room experimental model," in Proc. 4th Global Conf. Life Sci. Technol., Osaka, Japan, Mar. 2022, pp. 252-253.

[19] S. Huda, N. Funabiki, M. Kuribayashi, R. W. Sudibyo, N. Ishihara, W.-C. Kao, "A proposal of air-conditioning guidance system using discomfort index," in Proc. Broadband Wireless Comput., Commun. Appl., Cham, Switzerland, 2020, pp. 154-165.

[20] "Network Simulator 3," Available online: https://www.nsnam.org (accessed on 22 November 2022).

[21] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, "You only look once: unified, real-time object detection," in Proc. Conf. Comput. Vision. Patt. Recog., pp. 779-788, 2016.

[22] R. Joseph, A. Farhadi, "YOLO9000: better, faster, stronger," in Proc. Conf. Comput. Vision. Patt. Recog., pp. 7263-7271, 2017.

[23] R. Joseph, A. Farhadi, "Yolov3: an incremental improvement," arXiv preprint arXiv:1804.02767, 2018.

[24] "YOLOv3," Available online: https://pjreddie.com/darknet/yolo/ (accessed on 11 October 2023).

[25] A. Bochkovskiy, C-Y. Wang, H-Y. M. Liao, "Yolov4: optimal speed and accuracy of object detection," arXiv preprint arXiv:2004.10934, 2020.

[26] "YOLOv5," Available online: https://github.com/ultralytics/yolov5 (accessed on 11 October 2023).

[27] C. Li, L. Li, H. Jiang, K. Weng, Y. Geng, L. Li, Z. Ke, Q. Li, M. Cheng, W. Nie, Y. Li, B. Zhang, Y. Liang, L. Zhou, X. Xu, X. Chu, X. Wei, X. Wei, "YOLOv6: a single-stage object detection framework for industrial applications," arXiv preprint arXiv:2209.02976, 2022.

[28] C-Y. Wang, A. Bochkovskiy, H-Y. M. Liao, "YOLOv7: trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," in Proc. Conf. Comput. Vision. Patt. Recog., pp. 7464-7475, 2023.

[29] "YOLOv8," Available online: https://docs.ultralytics.com/models/yolov8/ (accessed on 11 October 2023).

[30] "Roboflow," Available online: https://roboflow.com/ (accessed on 11 October 2023).

[31] Y, Lin, "Fast LP models and algorithms for identical jobs on uniform parallel machines," Appl. Math. Model., 2013, 37, 3436–3448.

[32] A. Mallek, M. Bendraouche, M. Boudhar, "Scheduling identical jobs on uniform machines with a conflict graph," Comput. Oper. Res., 2019, 111, 357–366.

[33] S. Bansal, C. Hota, "Efficient refinery scheduling heuristic in heterogeneous computing systems," J. Adv. Inform. Technol., 2011, 2, 159–164.

[34] G. Murugesan, C. Chellappan, "Multi-source task scheduling in grid computing environment using linear programming," Int. J. Comput. Sci. Eng., 2014, 9, 80–85.

[35] R. Garg, A. Singh, "Adaptive workflow scheduling in grid computing based on dynamic resource availability," Eng. Sci. Technol., 2015, 18, 256–269.

[36] M.B. Gawali, S.K. Shinde, "Standard deviation based modified Cuckoo optimization algorithm for task scheduling to efficient resource allocation in cloud computing," J. Adv. Inform. Technol., 2017, 8, 210–218.

[37] L.F. Bittencourt, A. Goldman, E.R.M. Madeira, N.L.S. da Fonseca, R. Sakellariou, "Scheduling in distributed systems: A cloud computing perspective," Comput. Sci. Rev., 2018, 30, 31–54.

[38] I. Attiya, M.A. Elaziz, S. Xiong, "Job scheduling in cloud computing using a modified Harris Hawks optimization and simulated annealing algorithm," Comput. Intelli. Neuro., 2020, 3504642.

[39] A.A. Heidari, S. Mirjalili, H. Faris, I. Aljarah, M. Mafarja, H. Chen, "Harris Hawks optimization: Algorithm and applications," Future Gen. Comput. Syst., 2019, 97, 849–872.

[40] B.A. Al-Maytami, P.F.A. Hussain, T. Baker, P. Liatsis, "A Task Scheduling Algorithm With Improved Makespan Based on Prediction of Tasks Computation Time algorithm for Cloud Computing," IEEE Access, 2019, 7, 160916–160926.

[41] S.K. Panda, P.K. Jana, "An energy-efficient task scheduling algorithm for heterogeneous cloud computing systems," Clust. Comput., 2019, 22, 509–527.

[42] T. Juan, D. Cordova-Esparza, "A comprehensive review of YOLO: From YOLOv1 to YOLOv8 and beyond," arXiv preprint arXiv:2304.00501, 2023.

[43] P. Jiang, D. Ergu, F. Liu, Y. Cai, B. Ma, "A review of Yolo algorithm developments," Procedia Comput. Sci., vol. 199, pp. 1066-1073, 2022.

[44] H. Talebi, P. Milanfar, "Learning to resize images for computer vision tasks," in Proc. Comput. Vision. Patt. Recog., pp. 497-506, 2021.

[45] J. Lu, C. Ma, L. Li, X. Xing, Y. Zhang, Z. Wang, J. Xu, "A vehicle detection method for aerial image based on YOLO," J. Comput. Commun., vol. 6, no. 11, 98-107, 2018.

[46] A. Abhinand, J. Mulerikkal, A. Antony, P. A. Aparna, A. C. Jaison, “Detection of moving objects in a metro rail CCTV video using YOLO object detection models,” in Proc. ICDMAI, vol. 1, pp. 183-195, 2021.

[47] N. Kumari, V. Ruf, S. Mukhametov, A. Schmidt, J. Kuhn, S. Küchemann, “Mobile eye-tracking data analysis using object detection via YOLO v4,” Sensors, no. 22, 2021.

[48] F. Zhou, H. Zhao, Z. Nie, “Safety helmet detection based on YOLOv5,” in Proc. ICPECA, pp. 6-11, 2021.

[49] K. Liu , Q. Sun, D. Sun, P. Lin, M. Yang, N. Wang, “Underwater target detection based on improved YOLOv7,” J. Marine Sci. Eng., vol. 11, no. 3, 2023.

[50] F. M. Talaat, H. ZainEldin, “An improved fire detection approach based on YOLO-v8 for smart cities,” Neural Comput. Appl., vol. 35, pp. 20,939-20,954, 2023.

[51] V. Arpita, D. C. Anastasiu, “Enhancing retail checkout through video inpainting, YOLOv8 detection, and deepSort tracking,” in Proc. Conf. Comput. Vision. Patt. Recog., pp. 5,529-5,536, 2023.

[52] L. Haitong, X. Duan, J. Guo, H. Liu, J. Gu, L. Bi, H. Chen, “DC-YOLOv8: small-size object detection algorithm based on camera sensor,” Electronics, vol. 12, no. 10, 2023.