A Proposal of Job-Worker Assignment Algorithm Considering CPU Core Utilization for User-PC Computing System

Ariel Kamoyedji, Nobuo Funabiki, Hein Htet, and Minoru Kuribayashi

***Abstract*—As a low-cost high-performance *master-worker-model*-based computing platform for group members, we have studied the *User-PC computing system (UPC)*. The *UPC master* assigns queuing jobs from users to *UPC workers* that use idling computing resources of members' *personal computers (PCs)*. In this paper, we propose a *job-worker assignment algorithm* to minimize the *makespan*, considering the number of job threads and the number of CPU cores. For evaluation of the algorithm, we conducted experiments running 72 jobs on the UPC system with six workers that have various numbers of threads and CPU cores. The schedules by the algorithm could significantly reduce the *makespan* compared to other algorithms.**

***Index Terms*—UPC, distributed computing, CPU core, thread, job scheduling, local search, optimization.**

I. Introduction

As machine learning technologies for artificial intelligence (AI) become useful and common in various applications, the importance of low-cost and high-performance computing platforms has increased. On the other hand, the performance of the *personal computer (PC)* has been dramatically enhanced with the advancements of LSI technologies. Particularly, the number of CPU cores has significantly increased so that multithreaded programs can run in parallel and drastically reduce the required CPU time for job completion.

As a *master-worker model* based very low-cost and high-performance computing platform for members of a group such as a university laboratory or a company section, we have studied the *User-PC computing system (UPC)* [1]-[3]. In this system, 1) a user of the UPC system submits computing jobs to the *UPC master* through the *UPC web server*. 2) The master assigns the jobs to the proper *UPC workers*. 3) Each worker computes the assigned jobs and returns the results to the master. 4) The user accesses the results through the web server. To improve the response performance, jobs should be assigned to workers in such a way that the *makespan* or the latency for computing all the requested jobs be minimized.

In this paper, we propose a *job-worker assignment algorithm* for the UPC system. To reduce the CPU time by efficiently using parallel processing, it considers the number of threads used by a job during execution and the number of CPU cores in a worker. First, in the *static assignment*, this algorithm assigns each available job to a worker to minimize the *makespan*. Then, in the *dynamic assignment*, it assigns each newly arrived job to a worker when a worker becomes idle.

For evaluation, we conducted experiments running 72 jobs on the UPC system with six workers that have different number of threads and CPU cores. Two extreme cases are examined to investigate the static assignment and the dynamic assignment individually. In the first case, all the jobs are available. In the second one, jobs join the system dynamically according to a *Poisson process*. The proposed algorithm could significantly reduce the *makespan* compared to other algorithms.

II. Related Works

In this section, we overview related works in the literature. Within our survey, no work has considered the number of CPU cores and/or the number of job threads.

In [4], Xu *et al.* proposed the *Deadline Preference Dispatch Scheduling (DPDS)* algorithm as a dynamic scheduling algorithm that considers the deadline constraint priority. They also proposed the *Improved Dispatch Constraint Scheduling (IDCS)* algorithm that uses a risk prediction model to reduce the waste of computing resources and maximize the number of completed tasks.

In [5], Amalarathinam *et al*. proposed *the Dual Objective Dynamic Scheduling Algorithm (DoDySA)* that allocates the tasks based on the *Earliest Starting Time (EST)* and the *Earliest Finishing Time (EFT)* with the two objectives of maximizing the processor utilization and minimizing the makespan. The experiments results showed that *DoDySA* outperforms the others.

In [6], Bhatia discussed task scheduling algorithms for grid computing in literature. They are categorized into heuristic approach ones and nature inspired ones. Their main goal is to minimize the execution time of each job or to improve the processing capacity of the available resources.

In [7], Ernemann *et al.* applied economic models to the scheduling problem, and came up with a market-economic method that performs quite well. Their proposal is a decentralized one where several geographical domains are defined. Each domain is equipped with a local scheduling instance called *MetaManager*.

In [8], Xie *et al.* proposed the dynamic scheduling algorithm with security awareness called *EDF_OPTS*. It can achieve the high quality of security for real-time tasks while improving resource utilizations. It is an optimized version of the *Earliest Deadline First (EDF)* scheduling algorithm that maintains high guarantee ratios while maximizing security values, by adjusting security levels of accepted tasks.

In [9], Wang *et al.* proposed the *DPK (Dynamic priority and 0-1Knapsack)* algorithm to tackle the scheduling problem for multiple *DAG (Directed Acyclic Graph)-structure* hard

Manuscript received on January 3, 2022; revised May 1, 2022.
The authors are with Department of Information and Communication Systems, Okayama University, Japan (e-mail: funabiki@okayama-u.ac.jp).

doi: 10.18178/ijfcc.2022.11.2.586

real-time applications. The latter is a non-preemptive two-level algorithm that is based on the dynamic job priority adjustment. It resorts to the 0-1 Knapsack algorithm to keep CPUs busy during idling periods by assigning jobs to them.

In [10], Pooranian *et al*. proposed the *Group Leaders Optimization Algorithm (GLOA)*, which was inspired by the effect of leaders in social groups. To make the algorithm converge quickly, they divided the problem space into several smaller parts called groups. Each group is searched in parallel to increase the speed. Each separate space can be searched by its leader, who tries to find a solution by checking whether it's the closest member to the local and global minimum.

In [11], Seol *et al.* proposed the power-aware scheduling algorithm as an improved version of the *Cycle Conserving Earliest Deadline First (CC-EDF)* algorithm. This algorithm can be applied directly to static systems of the pinwheel task model and is more effective than other static schemes when the power-saving is concerned. Their simulation results showed that the algorithm reduces energy consumptions by 10 to 80% over existing ones.

In [12], Garg *et al*. proposed the *Adaptive Workflow Scheduling (AWS)* algorithm as a decentralized scheduling algorithm operating in three phases, the resource discovery and monitoring, the static task scheduling, and rescheduling, based on dynamic resource availability and using a directed acyclic graph workflow model. AWS uses an adaptive scheduling strategy that aims at minimizing the makespan of the workflow application.

III. Job-worker Assignment Problem Formulation

In this section, we formulate the job-worker assignment problem for the UPC system.

A. Symbols

We define the variables and symbols in the formulation.

- *Wk*: the set of available workers,
- *wk*: a worker in *Wk* that is characterized by the number of the CPU cores, the memory size, and the disk space,
- *Jb*: the set of given jobs to process,
- *jb*: a job in *Jb* that is characterized by the number of threads and the required memory and disk spaces,
- $c_{jb,wk}$: the computation cost associated with the processing of job *jb* on worker *wk* ,
- $\theta_{jb,wk}$: the processing time associated with the processing of job *jb* on worker *wk* , and
- *f(jb,wk)*: an assignment function with value 1 if job *jb* is assigned to worker *wk* and 0 otherwise.

B. Assumptions on Job-Worker Assignments

We make the assumptions on job-worker assignments:

- any worker can process one job at a time to avoid tasks swapping,
- any worker can be distinct from others in terms of number of CPU cores,
- any job can be assigned to any worker that can process it,
- all the queuing jobs can be assigned to workers at the same time, and
- any future job arrival cannot be predicted.

C. Problem Formulation

The job-worker assignment problem for the UPC system can be formulated as a combinatorial optimization problem. The goal is to minimize the *makespan*, subject to the constraints related to available resources on workers. This assignment problem is *NP-hard* [13].

1) *Objective*: To minimize the following function **F**:

$$F = \sum_{wk\in Wk}\sum_{jb\in Jb} f(jb,wk)(\theta_{jb,wk}-1) \tag{1}$$

2) *Constraints*

- The total number of assigned jobs must be less than or equal to the total number of available jobs:

$$\sum_{wk\in Wk}\sum_{jb\in Jb} f(jb,wk) \le |Jb| \tag{2}$$

- The total number of job assignments is greater than or equal to the total number of available workers if there are more jobs than workers:

$$\sum_{wk\in Wk}\sum_{jb\in Jb} f(jb,wk) \ge |Wk|, if |Wk| \le |Jb| \tag{3}$$

- A job can be assigned once to a worker that can process it:

$$\sum_{wk\in Wk} f(jb,wk) = 1, \forall\, jb \in Jb \tag{4}$$

- A worker can be assigned at most all available jobs:

$$\sum_{jb\in Jb} f(jb,wk) \le |Jb|, \forall\, wk \in Wk \tag{5}$$

- The resource requirement of any job on any worker do never exceed the usage limit specified by the user:

$$f(jb,wk)c_{jb,wk} \le lim_w, \forall(jb,wk) \in (Jb * Wk) \tag{6}$$

- *f* is a two-variable binary function:

$$f(jb,wk) \in \{0,1\}, \forall(jb,wk) \in (Jb * Wk) \tag{7}$$

- The job processing time must be a positive real value:

$$\theta_{jb,wk} \in R_+^* \tag{8}$$

D. Problem Complexity

Given a set of jobs *Jb* and a set of workers *Wk*, the total number of possible job-worker assignments *N* is given by:

$$N = \sum_{i=1}^{k} \left\{ {n \atop i} \right\} P_i^k \tag{9}$$

where $\left\{ {n \atop i} \right\}$ represents the number of ways to partition a set of n objects into i non-empty subsets [14]. It is given by:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!}\sum_{i=0}^{k}(-1)^k \binom{k}{i}(k-i)^n \tag{10}$$

For instance, for a set of ten jobs and four workers, *N* = 171,889,200, which means that even for a small size problem, it is impossible to find the optimal schedule (job-worker assignments) by analyzing all possible combinations. Thus, an approximation or heuristic algorithm is necessary to tackle the job scheduling problem in the UPC system.

IV. Job-Worker Assignment Algorithm

This section presents the job-worker assignment algorithm composed of the greedy *initial stage* and the local search *improvement stage* for the UPC system.

A. Initial Stage by Greedy Method

The initial stage of the proposed algorithm generates a feasible solution to the problem from scratch, using a *greedy* *method*. To efficiently utilize the CPU cores in workers, this stage groups workers and jobs into several classes according to the number of available cores for workers or required threads for jobs. Then, it greedily sets up job-worker assignments in each class, independently. In this paper, the number of classes is set to two since all our workers have less than 20 cores.

1. *Algorithm Procedure*: The greedy method procedure for the initial stage is given as follows:
   1) Each job in the given job set *Jb* is assigned to either of the two job classes depending on their required number of threads during execution. Actually, a job goes to class 1 if it requires up to four threads during execution and goes to class 2 otherwise.
   2) Each worker in the given worker set *Wk* is assigned to either of the two worker classes depending on their available CPU core number. Actually, a worker goes to class 1 if it has up to four cores and goes to class 2 otherwise.
   3) In each job class, jobs are assigned to workers by using the following greedy method:
      a) Workers are sorted in ascending order of job processing time.
      b) Using the given approximate execution time $t_{jb,wk}$ for each job *jb* in the corresponding job class on each worker *wk* belonging to the current worker class, compute for each job a discriminator $\delta_j = \sum_{wk=1}^{|Wk|} \frac{t_{jb,wk}}{t_{jb,wk+1}}$ and sort jobs in the class in ascending order of δ values. Jobs with the lowest values of δ are to be assigned first for any given job class.
      c) For each job in the current job class, find the available worker in the current worker class that runs the current job within the shortest amount of time and assign the latter to it.

2. *Pseudo Code*: The pseudo code is given in Algorithm 1.

Algorithm 1 Greedy Method for Initial Stage

```
Input: a set of jobs Jb and a set of workers Wk.
Output: job-worker mapping.
workerClasses[2][n] ← ϕ, jobClasses[2][n] ← ϕ
for each worker wk ∈ Wk do
    if current worker wk has ≤ 4 threads then
        add wk to workerClasses[0]
    else
        add wk to workerClasses[1]
    end if
end for
for each job jb ∈ Jb do
    if current job jb uses ≤ 4 threads to execute then
        add jb to jobClasses[0]
    else
        add jb to jobClasses[1]
    end if
end for
for i = 0 to 1 do
    Sort workers in workerClasses[i] in ascending order of
    job processing time (CPU performance).
    for each job jb ∈ jobClasses[i] do
        δj ← 0
    for wk = 0 to count(workerClasses[i]) - 1 do
        δj ← δj + (t_jb, wk / t_jb, wk +1)
    end for
end for
Sort jobs in jobClasses[i] in ascending order of δ values.
    for each job jb ∈ jobClasses[i], from the lowest values
    of δ to the highest do
        Find the available worker in workerClasses[i] that
        processes jb in the shortest amount of time and assign
        jb to it.
    end for
end for
return the generated job-worker mapping
```

B. Improvement Stage by Local Search Method

The initial solution generated at the initial stage is improved by using a randomized multi-start local search method and additionally, hill climbing method is used to escape from local minima.

1. *Algorithm Procedure*: The local search procedure for the improvement stage is described as follows:
   1) Consider the output of the initial stage as the initial solution and define 4 neighborhood solution generation functions and a solution evaluation function as follows:
      - The first function implements a local search method that moves jobs from the bottleneck worker (worker with the highest *makespan*) to any other available worker.
      - The second function implements a local search method that moves jobs randomly from any worker to any other available one.
      - The third function implements a local search method that swaps jobs from the bottleneck worker with any other jobs being processed on any other workers.
      - The fourth function implements a local search method that randomly swaps jobs from any worker with any other jobs being processed on any other workers.
      - The evaluation function takes a job-worker mapping as first parameter, compares it with the mapping passed in as second parameter and then returns the best of both (mapping with the shortest *makespan*).
   2) Randomly select one of the aforementioned functions and run it on the initial solution in order to generate a new initial solution for the current iteration. Then set the newly generated initial solution to be the best job-worker mapping so far.
   3) Randomly select one of the aforementioned functions and run it on the best job-worker mapping (mapping with the shortest *makespan*) so far, to improve it.
   4) In case a better mapping is found, replace the former best mapping with it and repeat the previous step and the current one several times, depending on the number of jobs. In our experiments, we repeated these steps 50 times for 24 jobs, 100 times for 48 jobs and 150 times for 72 jobs.
   5) Compare the best overall mapping resulting from the previous step with the best overall mapping so far and update the latter if it's no more the best one.
   6) Repeat the four previous steps several times depending on the number of jobs, and return the overall best mapping found. In our experiments, we repeated these steps 50 times for 24 jobs, 100 times for 48 jobs and 150 times for 72 jobs.

2. *Pseudo Code*: The *pseudo* code is given in Algorithm 2.

Algorithm 2 Local Search for Improvement Stage

```
Input: The initial job-worker mapping.
Output: The best job-worker mapping found.
        mapping firstLocalSearchMethod (mapping)
        mapping secondLocalSearchMethod (mapping)
        mapping thirdLocalSearchMethod (mapping)
        mapping fourthLocalSearchMethod (mapping)
        mapping evaluateMapping (mapping, mapping)
        bestOverallMapping←
            initialMapping
    for i = 1 to 50 do
            Randomly select one of the previously mentioned
            local search methods.
            currentIterationInitialMapping←
                selectedLocalSearchMethod (initialMapping)
            bestTemporaryMapping←
                currentIterationInitialMapping
        for j = 1 to 50 do
                Randomly select one of the aforementioned 4 local
                search methods.
                newMapping←
                    selectedLocalSearchMethod
                    (bestTemporaryMapping)
                bestTemporaryMapping←
```

```
                evaluateMapping(newMapping,
                bestTemporaryMapping)
            end for
        bestOverallMapping ←
            evaluateMapping(bestTemporaryMapping,
            bestOverallMapping)
    end for
    return bestOverallMapping
```

3. *Time Complexity*: In the greedy algorithm, the construction of worker classes takes $O(|Wk|)$ while that of job classes takes $O(|Jb|)$. Then, assuming all sorting operations are being carried out using quick sort algorithm, let WC be the set of worker classes, WC[i] be any worker class and JC[i] be any job class. Given that the number of worker classes is ≥ 2, the maximum number of workers in a worker class is |Wk| - 1 and the second part of the greedy algorithm (comprising δj calculation for each job in each class, two sorting operations and jobs assignment to workers) takes:

$$O(|WC|*(Max(|WC[i]|)log(Max(|WC[i]|))+Max(|WC[i]|)*Max(|JC[i]|)+Max(|JC[i]|)log(Max(|JC[i]|))+Max(|WC[i]|)*Max(|JC[i]|)))=O(|WC|*(Max(|WC[i]|)log(Max(|WC[i]|))+2Max(|WC[i]|)*Max(|JC[i]|)+Max(|JC[i]|)log(Max(|JC[i]|))))=O(|WC|*((|Wk|-1)log(|Wk|-1)+(|Jb|-1)log(|Jb|-1)+2(|Wk|-1)*(|Jb|-1)))=O(|WC|*(|Wk|log|Wk|+|Jb|log|Jb|+|Wk||Jb|))=O(|WC||Wk||Jb|)$$

The greedy algorithm thus takes no more than: $O(|WC||Wk||Jb|) + O(|Wk|) + O(|Jb|) = \boldsymbol{O(|WC||Wk||Jb|)}$

In the heuristic algorithm, each of the four local search methods takes $O(Max(|WC[i]|) * Max(|JC[i]|))$ and the mapping evaluation algorithm takes $O(|Jb|^2)$. Then, assuming that we run the selected local search algorithm α times and the multi-start local search algorithm β times and that $|Jb| > |Wk|$, the heuristic algorithm takes:

$$O(\alpha(\beta(Max(|WC[i]|)*Max(|JC[i]|)+|Jb|^2)+|Jb|^2))=O(\alpha(\beta((|Wk|-1)(|Jb|-1)+|Jb|^2)+|Jb|^2))=O(\alpha(\beta(|Wk||Jb|+|Jb|^2)+|Jb|^2))=O(\alpha(\beta|Jb|^2+|Jb|^2))=O(\alpha|Jb|^2)=O(|Jb|^2)$$

The heuristic algorithm thus takes no more than $\boldsymbol{O(|Jb|^2)}$.

In conclusion, our scheduling method takes: $O(|WC||Wk||Jb|) + O(|Jb|^2) = O(|WC||Jb|^2)$ since $|Jb| > |Wk|$ = $\boldsymbol{O(|Jb|^2)}$ since $|WC|$ = CONST.

C. Dynamic Job-Worker Assignment

The dynamic job-worker assignment algorithm repeatedly calls the static one whenever there are both idling workers and uncompleted jobs in the system.

1) *Pseudo Code*: The pseudo code is given in Algorithm 3.

Algorithm 3 Dynamic Job Scheduling Algorithm

```
  Input: a queue of waiting jobs Jb with average arrival
rate λ and a set of workers Wk with average processing
rate μ.
  Output: None.
function getIdlingWorkerSet(workerSet)
  idlingWorkerSet ← ∅
  for each worker wk ∈ workerSet do
    if current worker wk is idling then
      Add the current worker to idlingWorkerSet.
    endif
  endfor
  return idlingWorkerSet
end function
    Wk, Jb, timer.initialize(), startTime ← 0,
    resultingMapping ← ∅, makespan = 0
    startTime ← timer.getCurrentTime()
    while true do
        idlingWorkerSet ← getIdlingWorkerSet(Wk)
        if idlingWorkerSet , ∅ then
            resultingMapping ← Static Job Scheduling
            Method (Wk, Jb)
            Assign jobs to idling workers according to
            resultingMapping.
            Update the queue of waiting jobs Jb.
        end if
        if there are no new job arrivals then
            makespan=timer.getCurrentTime()-startTime
            return makespan
        end if
        sleep for a short while
    end while
```

V. Evaluation

In this section, we evaluate the proposed algorithm by assigning and running the 24 jobs in Table II on a UPC system with the six workers in Table I. Table III shows the measured standard CPU time for each job running on each of the six workers. The algorithm was run on a PC with an Intel(R) Core (TM) i5-5200U CPU @ 2.20GHz processor, two CPU Cores, four threads, 8.00 GB memory and a 64-bit Windows 10 Education.

TABLE I: Workers Specifications

worker #	Core number	type of CPU	Clock rate	Memory size
master	4	icore5	3.20 GHz	8 GB
worker1	4	icore3	1.70 GHz	2 GB
worker2	4	icore5	2.60 GHz	2 GB
worker3	4	icore5	2.60 GHz	2 GB
worker4	8	icore7	3.40 GHz	4 GB
worker5	16	icore9	3.60 GHz	8 GB
worker6	20	icore9	3.70 GHz	8 GB

A. Evaluation Setup

To evaluate the algorithm with an increasing number of jobs, each of the 24 jobs was executed once (= 24 jobs in total), twice (= 48 jobs in total), and three times (= 72 jobs in total). In our evaluation, jobs join the system in the same order as in Table II. As performance index for evaluation, the makespan is calculated as the difference between the first job processing start time and the last job completion time.

B. Results for Static Job-Worker Assignment

First, we evaluate the static job-worker assignment results yielded by the proposed algorithm, through comparisons with reference algorithms.

1) *Reference Algorithms for Comparison*: To evaluate the effectiveness of the static scheduling algorithm, we solved the problems using three baseline algorithms we have devised, namely: *First Come First Served (FCFS), Memory consumption-based priority scheduling (M-Priority),* and *CPU thread usage-based priority scheduling (T-Priority).* The *FCFS* algorithm assigns queuing jobs to workers on a first-come first-served basis. The *M-Priority* algorithm assigns jobs in descending order of memory consumption. Actually, the more memory a job consumes, the more powerful the worker it will be processed on. The *T-Priority* algorithm assigns jobs in descending order of number of threads used during execution. That is, jobs using the highest number of threads are assigned first to the most powerful workers. All the reference algorithms as well as the static scheduling algorithm were implemented in *Java*.

TABLE II: Jobs Specifications

job #	job name	# of threads	disk usage
program1	Network Simulator (NS)	1	0.392 GB
program2	Optimization Algorithm (OA)	1	1.5 GB
program3	DCGAN	17	1.9 GB
program4	RNN	17	1.9 GB
program5	CNN	17	1.9 GB
program6	FFmpeg	18	2.8 GB
program7	Converter	1	1.1 GB
program8	Palabos	2	6.7 GB

program9	Flow	4	0.438 GB
program10	blockchain mining	1	920 MB
program11	COVID detection	23	2.95 GB
program12	COVID outbreak prediction	4	1.84 GB
program13	multimedia content resizing	18	2.8 GB
program14	multimedia content format changing	18	2.8 GB
program15	OpenFOAM 5W	1	1.4 GB
program16	OpenFOAM 10W	1	1.4 GB
program17	OpenFOAM 15W	1	1.4 GB
program18	OpenFOAM 20W	1	1.4 GB
program19	OpenFOAM 25W	1	1.4 GB
program20	OpenFOAM 30W	1	1.4 GB
program21	OpenFOAM 35W	1	1.4 GB
program22	OpenFOAM 40W	1	1.4 GB
program23	OpenFOAM 45W	1	1.4 GB
program24	OpenFOAM 50W	1	1.4 GB

2) *Makespan Results:* Table IV compares the *makespan* results yielded by the four algorithms for 24, 48 and 72 jobs. *improvement* in this table indicates the *makespan* difference between our proposed algorithm and the best of the three reference algorithms. The results clearly show that the proposed algorithm outperforms the reference algorithms. For reference, Table V shows the total CPU time required to process all the jobs on the workers.

TABLE III: JOBS STANDARD PROCESSING TIME

job #	worker1	worker2&3	worker4	worker5	worker6
program1	02:14:46	01:06:44	00:54:21	00:39:40	00:36:09
program2	00:41:43	00:26:38	00:20:27	00:13:53	00:13:10
program3	01:37:14	01:09:28	00:22:44	00:12:45	00:11:15
program4	00:17:43	00:12:01	00:07:03	00:05:37	00:04:49
program5	00:26:04	00:22:22	00:07:07	00:05:24	00:04:47
program6	00:46:37	00:31:49	00:13:23	00:07:59	00:06:54
program7	00:17:09	00:11:34	00:05:41	00:04:53	00:04:15
program8	00:12:51	00:08:38	00:05:24	00:04:29	00:03:19
program9	00:25:20	00:14:45	00:11:08	00:08:32	00:07:55
program10	00:36:28	00:09:09	00:07:53	00:05:59	00:04:14
program11	00:39:35	00:24:53	00:10:42	00:04:08	00:03:16
program12	00:13:12	00:06:35	00:05:09	00:03:55	00:03:01
program13	00:34:50	00:19:47	00:12:33	00:07:48	00:07:10
program14	00:36:33	00:24:56	00:10:55	00:05:45	00:04:19
program15	00:12:23	00:07:19	00:06:51	00:05:04	00:04:28
program16	00:29:58	00:19:18	00:16:58	00:13:08	00:11:27
program17	00:45:57	00:30:44	00:25:59	00:20:31	00:17:54
program18	00:55:44	00:36:54	00:32:25	00:25:17	00:22:13
program19	01:20:36	00:52:29	00:46:40	00:36:55	00:32:20
program20	01:37:25	01:05:44	00:56:27	00:45:18	00:39:42
program21	01:44:18	01:12:44	01:06:06	00:51:19	00:44:56
program22	01:52:04	01:26:35	01:14:53	01:00:08	00:52:44
program23	02:14:32	01:30:28	01:19:17	01:03:01	00:55:19
program24	02:22:38	01:32:22	01:23:02	01:07:02	00:58:40

TABLE IV: MAKESPAN RESULTS (H:M:S) FOR STATIC ASSIGNMENT

	24 jobs	48jobs	72 jobs
FCFS	02:58:36	04:55:19	09:03:37
M-Priority	02:37:28	06:07:47	07:37:18
T-Priority	02:47:21	05:02:03	08:20:53
Proposal	02:06:58	04:08:41	06:10:31
improvement	00:30:30 (20%)	00:46:38 (16%)	01:26:47 (19%)

TABLE V: TOTAL CPU TIME RESULTS (H:M:S) FOR STATIC ASSIGNMENT

	24 jobs	48 jobs	72 jobs
FCFS	13:50:34	26:37:15	42:28:37
M-Priority	13:35:17	27:52:08	41:01:32
T-Priority	13:47:05	26:22:36	41:30:02
Proposal	12:29:57	24:43:19	36:54:30

C. Results for Dynamic Job-Worker Assignment

Next, we evaluate the dynamic job-worker assignment results yielded by the proposed algorithm, through comparisons with reference algorithms. Here, we assume that new job arrivals in the UPC system occur according to a Poisson distribution and jobs are inserted into a non-preemptive queue [15] as they join the system. The average job arrival rate λ is set to 1 *job/500s*. Then, when workers become idle, queuing jobs are assigned and transmitted to workers by the algorithm.

1) *Reference Algorithms for Comparison*: For performance comparisons, additionally, two algorithms called *First Come First Served (FCFS)* and *Scheduling Upon Arrival (SUA)* are implemented and applied to the same set-up. *FCFS* randomly assigns the first arriving job to the first available worker. *SUA* assigns newly arrived jobs to workers as soon as they joined the system, using the previously described static job scheduling algorithm. Consequently, if a job is assigned to a currently busy worker, it has to wait until the worker becomes free, to be processed. Algorithms were implemented in *Java*.

2) *Results and Analysis*: Table VI compares the *makespan* results yielded by the three algorithms for 24, 48 and 72 jobs. Here, improvement means the *makespan* difference between the proposed algorithm and the best of the two reference algorithms. The results show that the proposed algorithm outperforms the reference algorithms. From the data in Table III, we could roughly estimate the average worker service rate in two steps. First, we calculated the average service rate of each worker over all jobs by:

$$\mu_{avgw} = \frac{1}{|Jb|} \sum_{jb \in Jb} \theta_{jb,wk}, \forall wk \in Wk. \tag{11}$$

Then, we calculated the average of the previous value over all the workers by:

$$\mu_{avgw} = \frac{1}{|Wk|} \sum_{wk \in Wk} \mu_{avgw} w = 1job/2023s. \tag{12}$$

The response time represents the total amount of time a job spends both in the queue and in service and is given by [16]:

$$\frac{C(|Wk|,\lambda/\mu)}{|Wk|\mu-\lambda} + \frac{1}{\mu} \tag{13}$$

Using the previous formula, the average response time of system can be estimated by:

$$AvgR_{ti} = \frac{C(|Wk|,\lambda/\mu)}{|Wk|\mu-\lambda} + \frac{1}{\mu} \tag{14}$$

The probability that an arriving job is forced to join the queue that is, all workers are occupied, is given by:

$$C\left(|Wk|, \frac{\lambda}{\mu}\right) = \frac{1}{1+(1-\rho)\left(\frac{|Wk|!}{(|Wk|\rho|Wk|)}\right)\sum_{k=0}^{|Wk|-1}\frac{(|Wk|\rho)^k}{k!}} \tag{15}$$

which is *Erlang's C formula* [15]. We calculate (14) using *Erlang C formula* and ρ value, $\rho = \frac{1/500}{(6*1/2023)} \approx 67\%$, as follows:

$$C(|Wk|, \lambda/\mu) = \frac{1}{1+(1-\rho)\left(\frac{|Wk|!}{(|Wk|\rho|Wk|)}\right)\sum_{k=0}^{|Wk|-1}\frac{(|Wk|\rho)^k}{k!}}$$

$$\approx \frac{1}{1+\left(\frac{0.33*6!}{4.02^6}\right)\left(\sum_{k=0}^{5}\frac{4.02^k}{k!}\right)} \approx \frac{1}{1+(0.06*43.56)} \approx 0.28.$$

Thus, $AvgR_{t_1} = \frac{0.28}{6*\frac{1job}{2023s}-\frac{1job}{500s}} + \frac{1}{\frac{1job}{2023s}} \approx \frac{0.28}{\frac{6jobs}{2023s}-\frac{1job}{500s}} + 2023s \approx 290s + 2023s = 2313s = 38min33s.$

Using experiment results, the average response time of the system is estimated for 24 distinct jobs as 47min10s

The theoretical average response time of the system $AvgR_{t1}$ is quite shorter than the estimated time using the experiment data $AvgR_{t2}$. This is mainly due to the fact that 2/3 of available jobs use 4 or less than 4 threads during execution and therefore, they are assigned to either worker1 or worker2 or worker3 (1/2 of available workers). Consequently, the bottleneck worker is always one of the previously mentioned workers. This performance drop could be efficiently mitigated by enabling the dynamic scheduling algorithm to implement job migration, that is, preempting and moving already assigned jobs to more powerful or idling workers.

TABLE VI: MAKESPAN RESULTS (H:M:S) FOR DYNAMIC ASSIGNMENT

	24 jobs	48 jobs	72 jobs
FCFS	06:31:49	13:38:40	16:49:14
SUA	06:18:36	11:09:23	15:38:22
Proposal	05:39:48	10:35:38	14:40:01
improvement	00:38:48 (10%)	00:33:45 (5%)	00:58:21 (6%)

TABLE VII: TOTAL CPU TIME RESULTS (H:M:S) FOR DYNAMIC ASSIGNMENT

	24 jobs	48 jobs	72 jobs
FCFS	14:18:04	33:39:58	46:22:17
SUA	15:09:51	30:55:48	45:47:59
Proposal	14:48:23	30:33:12	45:21:11

D. Discussion

The static scheduling algorithm finds better job-worker mappings by iteratively testing several mappings and selecting those that yield the shortest *makespans*. As an optimization algorithm, it requires a set of already available jobs to evaluate the possible job-worker mappings more efficiently. However, since the dynamic scheduling algorithm repeatedly calls the static one whenever a worker is idling, it is very likely that the number of new job arrivals between two iterations of the static algorithm is low. Thus, the static scheduling algorithm only runs on a handful of jobs, most of the time. This is the main reason why the makespan reduction yielded by the dynamic scheduling algorithm is quite low compared to that of the static one.

VI. CONCLUSION

This paper presented the job worker assignment algorithm to minimize the *makespan*, considering the number of job threads and the number of CPU cores. For evaluation of the proposed algorithm, we conducted experiments running 72 jobs on the UPC system with six workers that have various number of threads and CPU cores. The schedules by the algorithm could significantly reduce the *makespan* by up to 20%, compared to other algorithms. Our future work includes evaluation with more jobs and workers.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

AUTHOR CONTRIBUTIONS

Kamoyedji built up the algorithm, generated the data, and wrote the paper; Htet carried out experiments to verify the data; Funabiki supervised the whole study and revised the paper; and Kuribayashi provided advice to improve the paper; all authors had approved the final version of this paper.

REFERENCES

[1] N. Funabiki, K. S. Lwin, Y. Aoyagi, M. Kuribayashi, and W.-C. Kao, "A User-PC computing system as ultralow-cost computation platform for small groups," *Application and Theory of Computer Technology*, vol. 2, no. 3, pp. 10-24, 2017.

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

[3] H. Htet, N. Funabiki, A. Kamoyedji, M. Kuribayashi, F. Akhter, and W.-C. Kao, "An implementation of user-PC computing system using Docker container," *International Journal of Future Computer and Communication (IJFCC)*, vol. 9, no. 4, pp. 66-73, Dec. 2020.

[4] L. Xu, J. Qiao, S. Lin and W. Zhang, "Dynamic task scheduling algorithm with deadline constraint in heterogeneous volunteer Computing Platforms," *Future Internet*, vol. 11, pp. 1-16, 2019.

[5] G. Amalarathinam and A. M. Josphin, "Dual objective dynamic scheduling algorithm (DoDySA) for heterogeneous environments," *Advances in Computational Sciences and Technology*, vol. 10, no. 2, pp. 171-183, 2017.

[6] M. K. Bhatia, "Task scheduling in grid computing: A review," *Advances in Computational Sciences and Technology*, vol. 10, no. 6, pp. 1707-1714, 2017.

[7] C. Ernemann, V. Hamscher, and R. Yahyapour, "Economic scheduling in grid computing," *Lecture Notes in Computer Science*, vol. 2537, Springer, Berlin, Heidelberg, pp. 128-152, 2002.

[8] T. Xie, A. Sung, and X. Qin, "Dynamic task scheduling with security awareness in real-time systems," in *Proc. IEEE International Parallel and Distributed Processing Symposium*, pp. 8-15, 2005.

[9] M. Wang, Z. Du, Z. Liu, and S. Hao, "The dynamic priority-based scheduling algorithm for hard real-time heterogeneous CMP application," *Journal of Algorithms & Computational Technology*, vol. 2, no. 3, pp. 409-427, 2008.

[10] Z. Pooranian, M. Shojafar, J. Abawajy, and M. Singhal, "GLOA: A new job scheduling algorithm for grid computing," *International Journal of Artificial Intelligence and Interactive Multimedia*, vol. 2, no. 1, pp. 59- 64, 2013.

[11] Y. I. Seol and Y. K. Kim, "Applying dynamic priority scheduling scheme to static systems of pinwheel task model in power-aware scheduling," *The Scientific World Journal*, ID 587321, pp. 1-9, 2014.

[12] R. Garg, A. Singh, "Adaptive workflow scheduling in grid computing based on dynamic resource availability," *Engineering Science and Technology, an International Journal*, vol. 18, no. 2, pp. 256-269, 2015.

[13] L. Ozbakir, A. Baykasoglu, and P. Tapkan, "Bees algorithm for generalized assignment problem," *Applied Mathematics and Computation*, vol. 215, no. 11, pp. 3782-3795, 2010.

[14] R. L. Graham, D. E. Knuth, and O. Patashnik, "Concrete mathematics," Addison-Wesley, Reading MA, pp. 244, 1988.

[15] L. Kleinrock, "Queueing systems, volume 1: theory," Wiley-Interscience, 1975.

[16] M. Barbeau and K. Evangelos, *Principles of Ad-Hoc Networking*, John Wiley & Sons, 2007.

A. Kamoyedji received a master's degree in system and information engineering from Ashikaga Institute of Technology, Japan and a bachelor's degree in electrical engineering from UATM GASA Formation, Benin. He is currently a PhD student in the Graduate School of Natural Science and Technology at Okayama University, Japan, and a software engineer. His research interests include optimization algorithm design and distributed computing systems.

N. Funabiki received the B.S. and Ph.D. degrees in mathematical engineering and information physics from the University of Tokyo, Japan, in 1984 and 1993, respectively. He received the M.S. degree from Case Western Reserve University, USA, in 1991. In 2001, he joined the Department of Communication Network Engineering at Okayama University as a professor. His research interests include computer networks, optimization algorithms, and educational technology. He is a member of IEEE, IEICE, and IPSJ.

H. Htet received the B.E. and M.E. degrees in information science and Technology from the University of Technology (Yatanarpon Cyber City), Myanmar, in 2015 and 2018, respectively. He is currently a Ph.D student in the Graduate School of Natural Science and Technology at Okayama University, Japan. His research interests include distributed computing systems, big data analysis, and computer networks.

M. Kuribayashi received the B.E., M.E., and D.E. degrees from Kobe University, Japan, in 1999, 2001, and 2004, respectively. In 2002, he joined the Department of Electrical and Electronic Engineering, Kobe University, as an assistant professor. Since 2015, he has been an associate professor in Graduate School of Natural Science and Technology, Okayama University. His research interests include digital watermarking, information security, cryptography, and coding theory. He is a senior member of IEEE and IEICE.