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 utilization. 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
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:

\[
\sum_{wk \in Wk} \sum_{jb \in Jb} f(jb, wk) \leq |jb| \tag{2}
\]

- 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:

\[
\sum_{wk \in Wk} \sum_{jb \in Jb} f(jb, wk) \geq |Wk|, \forall wk \in Wk \tag{3}
\]

- 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:

\[
\sum_{jb \in Jb} f(jb, wk) \leq |jb|, \forall wk \in Wk \tag{5}
\]

- The resource requirement of any job on any worker do not exceed the usage limit specified by the user:

\[
f(jb, wk)c_{jb, wk} \leq \lim_{r}, \forall (jb, wk) \in (Jb \times Wk) \tag{6}
\]

- \( f \) is a two-variable binary function:

\[
f(jb, wk) \in [0, 1], \forall (jb, wk) \in (Jb \times Wk) \tag{7}
\]

- 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:

\[
N = \sum_{i=1}^{k} \binom{n}{i} l^k \tag{9}
\]

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

\[
\binom{n}{k} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k - i)^n \tag{10}
\]

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 \( \delta_j = \sum_{wk=1}^{\phi} \left( t_{jb, wk} / \text{worker} \right) \) and sort jobs in the class in ascending order of \( \delta \) values. Jobs with the lowest values of \( \delta \) 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.

1. workerClasses[2][n] ← \( \phi \), jobClasses[2][n] ← \( \phi \)
2. for each worker \( wk \) in \( Wk \) do
3.   if current worker \( wk \) has ≤ 4 threads then
4.     add \( wk \) to workerClasses[0]
5. else
6.   add \( wk \) to workerClasses[1]
7. end if
8. end for
9. for each job \( jb \) in \( Jb \) do
10.  if current job \( jb \) uses ≤ 4 threads to execute then
11.    add \( jb \) to jobClasses[0]
12.  else
13.    add \( jb \) to jobClasses[1]
14. end if
15. end for
16. for \( i = 0 \) to 1 do
17.   Sort workers in workerClasses[\( i \)] in ascending order of job processing time (CPU performance).
18. for each job \( jb \) in \( \text{jobClasses}[\( i \)] \) do
19.   \( \delta_j = 0 \)
20. for \( wk = 0 \) to count(workerClasses[\( i \)]) - 1 do
21.   \( \delta_j ← \delta_j + \left( t_{jb, wk} / \text{worker} \right) \)
22. end for
23. end for
24. Sort jobs in jobClasses[\( i \)] in ascending order of \( \delta \) values.
25. for each job \( jb \) in jobClasses[\( i \)], from the lowest values of \( \delta \) to the highest do
26.   Find the available worker in workerClasses[\( i \)] that processes \( jb \) in the shortest amount of time and assign \( jb \) to it.
27. end for
28. end for
29. 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.

1: mapping ← initialMapping
2: mapping ← secondLocalSearchMethod(mapping)
3: mapping ← thirdLocalSearchMethod(mapping)
4: mapping ← fourthLocalSearchMethod(mapping)
5: mapping ← mappingEvaluateMapping(mapping, mapping)
6: bestOverallMapping ← mapping
7: for \( i = 1 \) to 50 do
8:   Randomly select one of the previously mentioned local search methods.
9:   currentIterationInitialMapping ← selectedLocalSearchMethod(initialMapping)
10:  bestTemporaryMapping ← currentIterationInitialMapping
11:  for \( j = 1 \) to 50 do
12:    Randomly select one of the aforementioned 4 local search methods.
13:    newMapping ← selectedLocalSearchMethod(bestTemporaryMapping)
14:    bestTemporaryMapping ← newMapping
15:  end for
16:  end for
17: return the generated job-worker mapping
```

42
In the greedy algorithm, the construction of worker classes takes $O(|W_k|)$ while that of job classes takes $O(|J_b|)$. Then, assuming all sorting operations are being carried out using quick sort algorithm, let $WC$ be the set of worker classes, $JC[i]$ be any worker class and $J_b[i]$ be any job class. Given that the number of worker classes is $\geq 2$, the maximum number of workers in a worker class is $|W_k| - 1$ and the second part of the greedy algorithm (comprising $dj$ calculation for each job in each class, two sorting operations and jobs assignment to workers) takes:

$$O(|WC| \times (|WC| \times \log(|WC|) + |WC|)) = O(|WC| \times |WC| \log|WC|) + O(|WC|) = O(|WC|^2 \log|WC|) + O(|WC|).$$

The greedy algorithm thus takes no more than:

$$O(|WC| \times |WC| \log|WC|) + O(|WC|) + O(|J_b|) = O(|WC| \times |WC| \log|WC|).$$

In the heuristic algorithm, each of the four local search methods takes $O(\alpha(\beta(\beta((\beta|J_b|-1)+|J_b|)+|J_b|)+|J_b|)) = O(\alpha(\beta|J_b|^2) + O(\alpha|J_b|^2)$ since $|J_b| > |W_k|$. The heuristic algorithm thus no more than $O(|J_b|^2)$. The greedy approach, however, takes $O(|J_b|^2)$ since $|J_b| > |W_k|$. In conclusion, our scheduling method takes:

$$O(|WC| |W_k| |J_b|) + O(|J_b|^2) = O(|WC| |J_b|^2)$$

since $|J_b| > |W_k|$. 

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

$$O(|WC| \times (|WC| \times \log(|WC|) + |WC|)) = O(|WC| \times |WC| \log|WC|) + O(|WC|) = O(|WC|^2 \log|WC|) + O(|WC|).$$

The greedy algorithm thus takes no more than:

$$O(|WC| \times |WC| \log|WC|) + O(|WC|) + O(|J_b|) = O(|WC| \times |WC| \log|WC|).$$

In the heuristic algorithm, each of the four local search methods takes $O(\alpha(\beta(\beta((\beta|J_b|-1)+|J_b|)+|J_b|)+|J_b|)) = O(\alpha(\beta|J_b|^2) + O(\alpha|J_b|^2)$ since $|J_b| > |W_k|$. The heuristic algorithm thus no more than $O(|J_b|^2)$. In conclusion, our scheduling method takes:

$$O(|WC| |W_k| |J_b|) + O(|J_b|^2) = O(|WC| |J_b|^2)$$

since $|J_b| > |W_k|$. 

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 $J_b$ with average arrival rate $\lambda$ and a set of workers $W_k$ with average processing rate $\mu$.

Output: None.

function getIdleWorkerSet(workerSet)
  idlingWorkerSet ← Ø
  for each worker $w_k$ ∈ workerSet do
    if current worker $w_k$ is idling then
      Add the current worker to idlingWorkerSet.
    end if
  end for
  return idlingWorkerSet
end function

W_k, J_b, timer.initialize(), startTime ← 0, resultingMapping ← Ø, makespan = 0
1. while true do
2.   if idlingWorkerSet ← getIdleWorkerSet(W_k) then
3.     if idlingWorkerSet = Ø then
4.       resultingMapping ← Static Job Scheduling Method (W_k, J_b)
5.       Assign jobs to idling workers according to resultingMapping.
6.       Update the queue of waiting jobs $J_b$.
7.     end if
8.   end if
9.   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>
<thead>
<tr>
<th>worker #</th>
<th>Core number</th>
<th>type of CPU</th>
<th>Clock rate</th>
<th>Memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td>master</td>
<td>4</td>
<td>icore5</td>
<td>3.20 GHz</td>
<td>8 GB</td>
</tr>
<tr>
<td>worker1</td>
<td>4</td>
<td>icore3</td>
<td>1.70 GHz</td>
<td>2 GB</td>
</tr>
<tr>
<td>worker2</td>
<td>4</td>
<td>icore5</td>
<td>2.60 GHz</td>
<td>2 GB</td>
</tr>
<tr>
<td>worker4</td>
<td>4</td>
<td>icore5</td>
<td>2.60 GHz</td>
<td>2 GB</td>
</tr>
<tr>
<td>worker5</td>
<td>16</td>
<td>icore9</td>
<td>3.40 GHz</td>
<td>4 GB</td>
</tr>
<tr>
<td>worker6</td>
<td>20</td>
<td>icore9</td>
<td>3.70 GHz</td>
<td>8 GB</td>
</tr>
</tbody>
</table>

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, we now 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 powerful the 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>
<thead>
<tr>
<th>job #</th>
<th>job name</th>
<th># of threads</th>
<th>disk usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>program1</td>
<td>Network Simulator (NS)</td>
<td>1</td>
<td>0.392 GB</td>
</tr>
<tr>
<td>program2</td>
<td>Optimization Algorithm (OA)</td>
<td>1</td>
<td>1.5 GB</td>
</tr>
<tr>
<td>program3</td>
<td>DCGAN</td>
<td>17</td>
<td>1.9 GB</td>
</tr>
<tr>
<td>program4</td>
<td>RNN</td>
<td>17</td>
<td>1.9 GB</td>
</tr>
<tr>
<td>program5</td>
<td>CNN</td>
<td>17</td>
<td>1.9 GB</td>
</tr>
<tr>
<td>program6</td>
<td>FFmpeg</td>
<td>18</td>
<td>2.8 GB</td>
</tr>
<tr>
<td>program7</td>
<td>Converter</td>
<td>1</td>
<td>1.1 GB</td>
</tr>
<tr>
<td>program8</td>
<td>Palabos</td>
<td>2</td>
<td>6.7 GB</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>job #</th>
<th>worker1</th>
<th>worker2&amp;3</th>
<th>worker4</th>
<th>worker5</th>
<th>worker6</th>
</tr>
</thead>
<tbody>
<tr>
<td>program9</td>
<td>Flow</td>
<td>4</td>
<td>0.438 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program10</td>
<td>blockchain mining</td>
<td>1</td>
<td>920 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program11</td>
<td>COVID detection</td>
<td>23</td>
<td>2.95 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program12</td>
<td>COVID outbreak prediction</td>
<td>4</td>
<td>1.84 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program13</td>
<td>multimedia content reszing</td>
<td>18</td>
<td>2.8 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program14</td>
<td>multimedia content format changing</td>
<td>18</td>
<td>2.8 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program15</td>
<td>OpenFOAM 5W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program16</td>
<td>OpenFOAM 10W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program17</td>
<td>OpenFOAM 15W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program18</td>
<td>OpenFOAM 20W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program19</td>
<td>OpenFOAM 25W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program20</td>
<td>OpenFOAM 30W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program21</td>
<td>OpenFOAM 35W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program22</td>
<td>OpenFOAM 40W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program23</td>
<td>OpenFOAM 45W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program24</td>
<td>OpenFOAM 50W</td>
<td>1</td>
<td>1.4 GB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Table III: Jobs Standard Processing Time

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 $\lambda$ 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 IV 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:

$$\mu_{avgw} = \frac{1}{|W_k|} \sum_{j \in W_k} \theta_{j, w_k}, \forall W_k \in W_k.$$ (11)

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

$$\mu_{avgw} = \frac{1}{|W_k|} \sum_{W_k \in W_k} \mu_{avgw} = 1 \text{ job}/2023s.$$ (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]:

$$C(W_k, \lambda/\mu) = \frac{1}{|W_k|} \lambda/\mu + \frac{1}{\mu}.$$ (13)

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

$$AvgR_{l} = \frac{C(W_k, \lambda/\mu)}{|W_k|} \mu + \frac{1}{\mu}.$$ (14)

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

$$C(W_k, \frac{1}{\mu}) = \frac{1}{|W_k|} + (1-\rho)(\frac{|W_k|}{|W_k|}) \sum_{k=0}^{|W_k|-1} (\frac{|W_k|}{|W_k|})^k$$ (15)

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

$$C(W_k, \lambda/\mu) = \frac{1}{1 + (1-\rho)(\frac{|W_k|}{|W_k|}) |W_k|-1 (\frac{|W_k|}{|W_k|})^k}$$. (16)

Thus, $AvgR_{l_1} = \frac{0.28}{6/1+2023} \approx 0.28$. $AvgR_{l_2} = \frac{0.28}{360.06+43.56} \approx 0.28$. Thus, $AvgR_{l_1} \approx 2023s \approx 290s + 2023s = 2313s = 38min33s$.

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 $AvgR_{l_1}$ is quite shorter than the estimated time using the experiment data $AvgR_{l_2}$. 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, preemption and moving already assigned jobs to more powerful or idling workers.

<table>
<thead>
<tr>
<th>TABLE VI: MAKESPAN RESULTS (HOURS:MINUTES:SECONDS) FOR DYNAMIC ASSIGNMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS</td>
</tr>
<tr>
<td>Recent</td>
</tr>
<tr>
<td>SUA</td>
</tr>
<tr>
<td>Proposal</td>
</tr>
<tr>
<td>improvement</td>
</tr>
</tbody>
</table>

D. Discussion

The static scheduling algorithm finds better job-worker mappings by iteratively testing several mappings and selecting those that yield the shortest makespan. 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; Hiet 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


Copyright © 2022 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0).
Htet received the B.E. and M.E. degrees in information science and Technology from the University of Technology (Yatanarporn 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.