Pamuretas*: A Novel Algorithm for Parallel Job Scheduling with Multiple Dependency and Resource Constraints

Jiayao Zhang
Department of Computer Science
University of Hong Kong
jyzhang@cs.hku.hk

Abstract

In this paper, we propose Pamuretas (**Pa**ralle **Mu**tiple-Dependency **Re**source-Critical **Ta**sk **S**cheduler). Pamuretas is designed to address one particular instance of the *Customer-Producer* problem with *Resource-Constrained Project Scheduling Problem (RCPSP)* flavour. Very roughly speaking, we adopted a master-slave paradigm by introducing scheduler and dynamic dispatchers. Feedbacks are also included in the working cycle which is aimed at increasing robustness without sacrificing efficiency. We incorporate two deadlock detection and handling system, which also scale very well. We perform thorough tests on Pamuretas by over 5000 combinations of job configurations. From the results, we conclude Pamuretas is indeed *robust*, *flexible* and has good *scalability*. We conclude this writing by identifying the limitation of Pamuretas and suggest several possible directions for future work.

1 Introduction and Problem Formulation

In this writing, we will investigate one concrete problem instance given in the dependency directed acyclic graph in Figure 1. We assume the number of workers, production goals and storage spaces are known *a priori*. Further, we give the job specifications in Table 1. The rest of this writing is structured as follows: Section 2 introduces, in finer details, the proposed method along with the implementation details, schematics are also provided; Running examples are included, which are representative in such a way that those configurations are more likely to trigger deadlock in a less-careful implementation; Section 3 is devoted to a thorough benchmarking of the proposed method by over five thousand test cases. Finally, we conclude this writing by recalling the essence of the proposed method and suggest several directions for future work.

*Code is available at `https://github.com/zjiayao/Pamuretas`. Part of this project has been submitted in partial fulfillment for the course Operating Systems offered by University of Hong Kong, 2016-17. The author wishes to thank the teaching staff for this motivating problem.

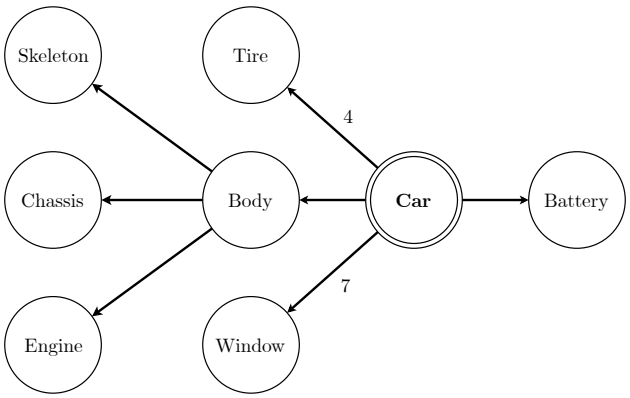


Figure 1: Dependency DAG.

Component	Time/sec	Amount	Storage Space
Skeleton	5	1	1
Engine	4	1	1
Chassis	3	1	1
Body	4	1	1
Window	1	7	1
Tire	2	4	1
Battery	3	1	1
Car	1	–	0

Table 1: Task Specification

2 Proposed Method

2.1 The Framework

The schematics of the proposed algorithm is illustrated in Figure 2. We now give a brief description of the general model and workflow.

The core of the proposed method is SCHEDULER and two job DISPATCHERs. We model the jobs in the dependency graph using two FIFO queues, namely, the *production line queue* and the *assembly line queue*. Those queues are initialized when the job detail is known. Hence the proposed method does not suffer from the waste of production goods; furthermore, based on the detail of job configurations.

As the number of cars, workers and storage spaces are known, the algorithm first of all proceed to initialized the working queues. It first analyze the configuration via `allocate_quota()`, which terminates the process if no valid job assignment scheme is possible. This will only happen when a single worker with no more that 13 spaces is given (why?). It then allocates the quota of the workers that the two DISPATCHERs may use. Heuristically, we assign the WORKERs in the ratio of 3 : 1 to the two lines if possible. We further ensure both lines have at least one WORKER when resources are limited. In the case of single WORKER, we only assign the free WORKER to the production line though; the assembly line is to be assigned and mediated on the run. At this stage, the algorithm also determines the *scheduling policy*, that is, either *normal*

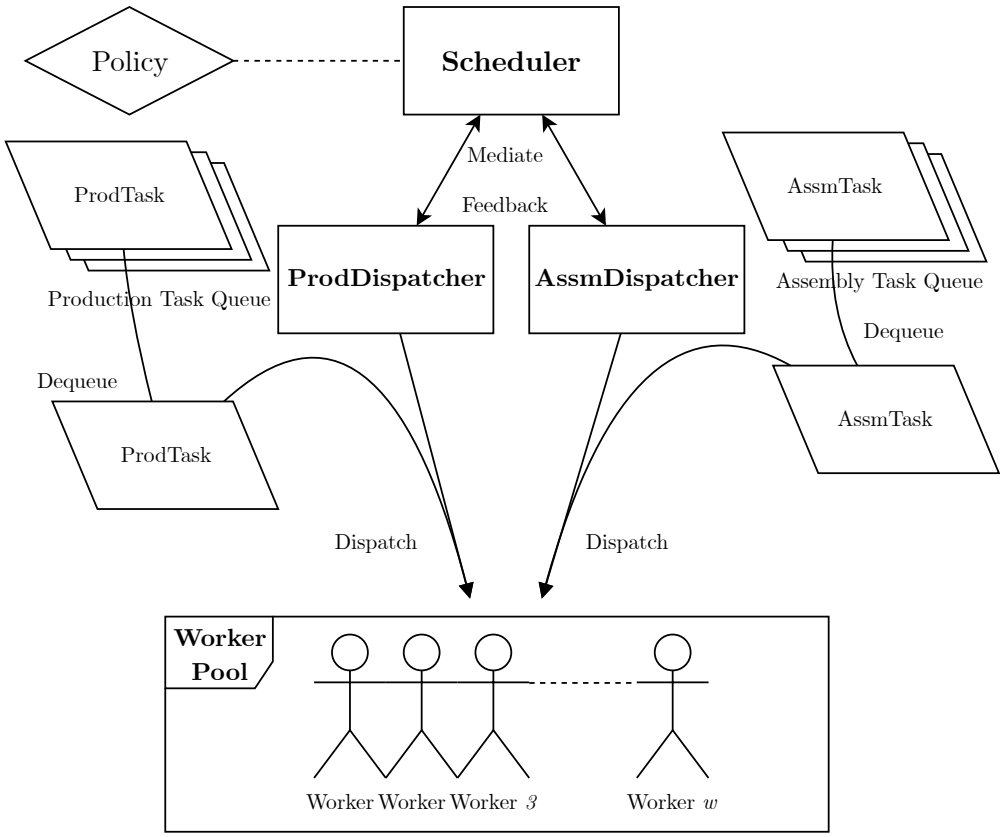


Figure 2: Schematics of the proposed method.

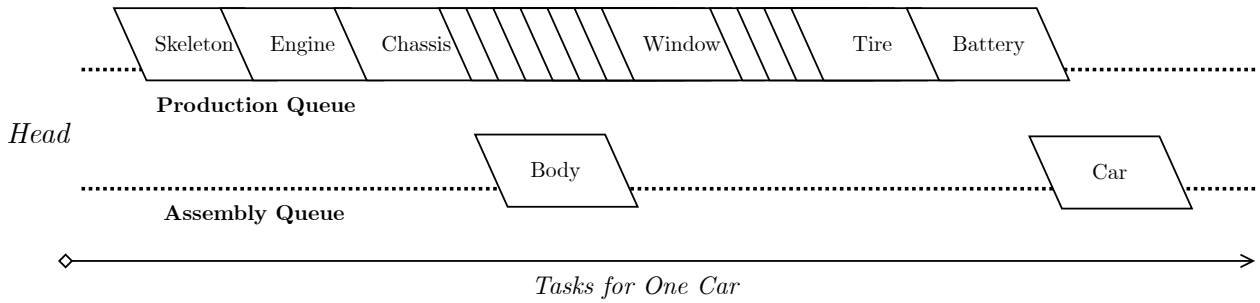


Figure 3: Task queues.

or *critical*. The latter occurs when the number of free spaces are less than 13 or the number of workers are no more than 13. The former is the minimum number required for the successful execution of single WORKER; whilst the latter is heuristically chosen based on empirical results. Under critical policy, more controls will be imposed by the SCHEDULER.

The completion of the quota allocation prophets the construction of the production queues. Among eight individual tasks, five do not have any dependencies (i.e., SKELETON, CHASSIS, ENGINE, WINDOW, TIRE and BATTERY) whereas two have some dependencies (i.e., BODY and CAR). The former are enqueued to the PRODUCTIONQUEUE whilst the latter ASSEMBLYQUEUE, in the order depicted in Figure 3. Generally, we follow the topological order traversing the DAG when enqueuing tasks; the tasks of the same depth are such that the order will be consistent throughout the implementation. Noted in Figure 3, the horizontal span also indicates the (only) correct order of execution for the single WORKER case.

2.2 The Scheduler

After several additional initializations and book-keeping tasks, we proceed to initiate the SCHEDULER and enter the main production loop. The main workflow of the SCHEDULER is outlined in Figure 4. Two components form the integral part of its lifespan, namely, CHECKFAULTS and CHECKSANITY. In the former, we book-keep the number of failure counts of the production and assembly line, which is defined to be the running sum of the counts both dispatchers have to wait until the satisfaction of the pre-conditions:

- Pre-condition for PRODUCTIONDISPATCHER
  There should be sufficient room for storing the components accrued.
- Pre-condition for ASSEMBLYDISPATCHER
  The source materials should be available, i.e., dependencies in Figure 1 are satisfied.

However, a large number of failure counts indicates, for the case of PRODUCTIONDISPATCHER, space is critical, production should be initiated to motivate the assembly line; for the case of ASSEMBLYDISPATCHER, this either implies that materials are insufficient, hence we signal the PRODUCTIONDISPATCHER to supply more materials; or in the case of halting PRODCUTIONLINE: some materials produced needed to be collected. This mechanism may be considered as an infancy form of the negative-feedback loop.

Sanity checking is another important part of the execution that ensures the robustness in case of deadlock. It is scheduled to run very sporadically, and within its invocation, it compares the current state of each components with the previous snapshot. If it observes the number of each component does not change in a row, it would further consider to take measures by invoking DEADLOCKHANDLER. Its conditions are set to be very conservative (e.g., the time

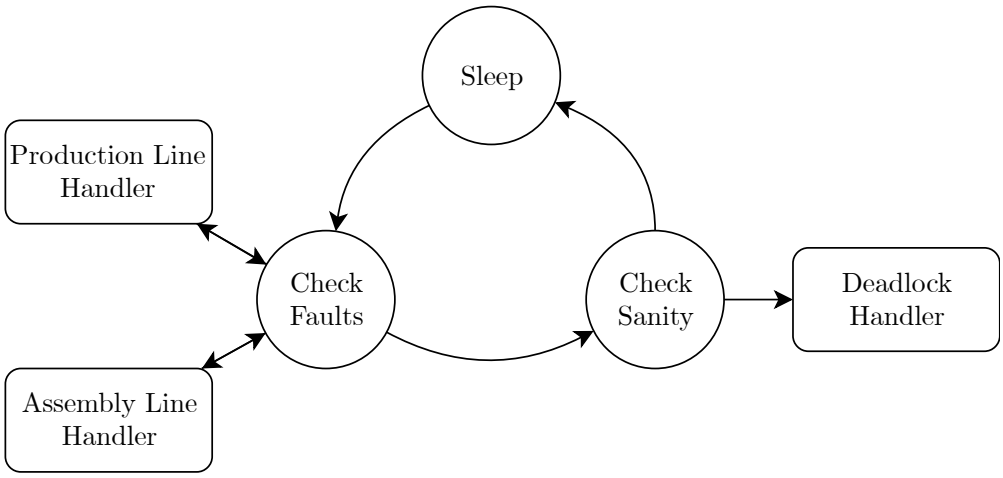


Figure 4: Workflow for SCHEDULER.

current program has executed must at least exceed ten times the expected time of completion, which is based on very conservative estimate of the job configuration). In fact, during our thorough testing detailed next section, we did not observe one single instance of its invocation.

2.3 The Dispatchers

The SCHEDULER does not directly delegate the tasks to the workers though. This is further controlled by two dedicated DISPATCHERs. The main workflow are depicted in Figure 5. In each cycle, the DISPATCHERs would wake up from sleep (either from the satisfaction of pre-conditions or being interrupted), dequeuing the top task to be executed and dispatch to an available WORKER designated to this line. The WORKER thread would continue execution or queuing whilst the DISPATCHER would decide, based on the current job status, whether to signal the other DISPATCHER by feeding back to the SCHEDULER, or dispatch more tasks. On termination, the DISPATCHER would return the WORKER quota to the other DISPATCHER for better resource utilization. This enables smooth execution with limited resource and high resource utilization when the labour force and space are abundant.

Concretely, consider three critical edge cases outlined below:

- Single Worker, Thirteen Spaces.

  This is one very critical scenario. A slight misgiving in the scheduling would cause deadlock. The proposed method would assign the jobs to the only WORKER in accordance with the flow in Figure 3 by delegating between DISPATCHERs. As such, a single WORKER is able to construct one car given thirteen spaces within about 40.02 seconds; and two cars within 80.02 seconds.

- Multiple Workers, One Space.

  This scenario posts two difficulties: how to manage one space; and how to scale up when overwhelming workers are available. To address the former, when space is limited, we tighten the pre-conditions for the ASSEMBLYLINE, in particular, we ask it try to start assembling even when materials are unavailable hence production may continue; To address the latter, we further require that it may not start assembling a CAR when a BODY has not made or no worker is working on it when we have less than two ASSEMBLYLINE workers. This tight control yields a decent result, as reported in Table 2. We wish to point out that,

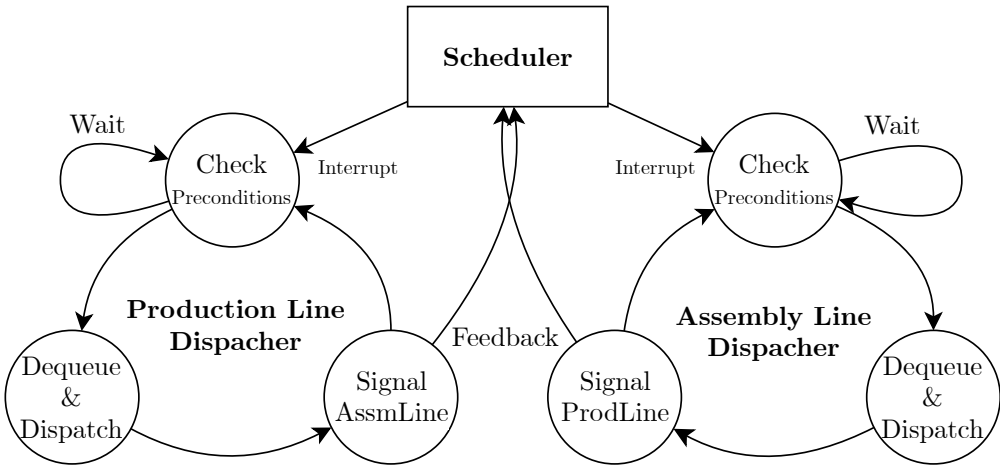


Figure 5: Workflow for DISPATCHER.

given eight workers, our algorithm takes as much time as previous implementations, which was optimized for eight-batch productions; it even outperforms when given 100 workers. However, since only one space is given, we do not expect such boost to be significant.

- Critical Combination.

  Under certain scenarios, our method would fail *were it not* the feedback cycle embedded in the SCHEDULER-DISPATCHER triple. Consider, for example, 15 WORKERs are to build multiple cars with 15 spaces. A greedy method may stuck if all spaces are devoted to the storage of the dependencies of the CAR and thus no space for storing BODY. Removing our scheduling logic *and* the deadlock handler mechanisms yields the deadlock scenario in Figure 6a.

Workers	2	3	4	5	8	One Hundred
Time/sec	74.10	47.09	39.02	37.11	30.08	27.15

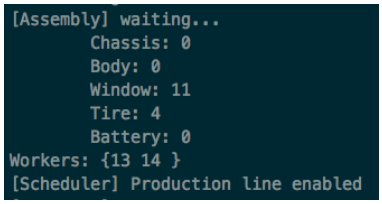
Table 2: Benchmarking on single space case, where the goal is two cars.

2.4 The Worker

WORKERs are the very threads that complete the task. Unlike the previous implementation, each WORKER on the ASSEMBLYLINE may *not* wait for one resource indefinitely. Rather, it would try to grab whatever material available that is needed for the production in a Round-Robin fashion. I.e., if one unavailable, it tries to fetch some other materials first before checking again. This workflow is illustrated in Figure 7. The WORKER would never release resources once acquired though and their work is non-preemptive. The mediation between competition and hunger is handled by the SCHEDULER and DISPATCHERs. This boosts the performance when the working force is huge even spaces are medium, for example, 120 workers can construct 9 cars given 15 storage spaces within about 18.81 seconds (a throughput about 0.5 car(s) per second).

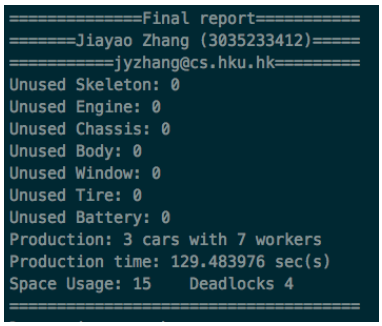
2.5 The Deadlock Handler

In addition to the SCHEDULER logic, we implemented a backup deadlock detection and handling system in case of unexpected system-level failure. In essence, this system is invoked

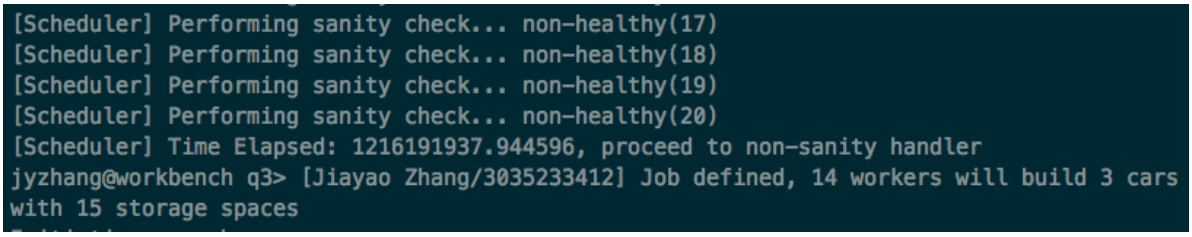


(a) *Deadlock.*

(b) *Scheduler Logic Enabled.*



(c) *Deadlock Handler Enabled.*



(d) *Sanity Checking and Deadlock Handling. The new prompt is spawned by the* DeadlockHandler.

Figure 6: Illustration of deadlock detection and handling: 3 cars with 15 workers and 15 spaces.

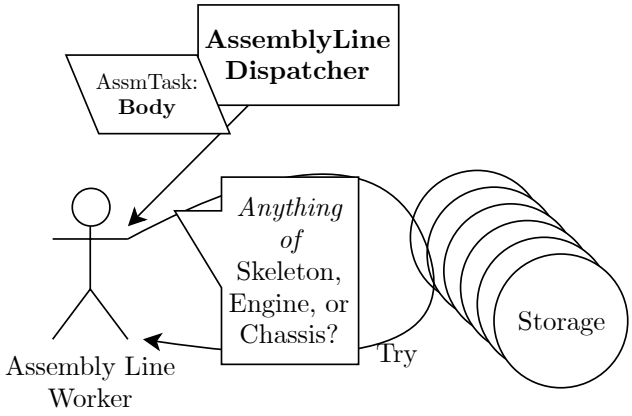


Figure 7: Workflow for Worker.

through the SanityCheck subroutine in the Scheduler. On execution, the DeadlockHandler would check the current on-time, and compared to a conservative estimate from the job configuration. It will give the program a second chance at first notice. After determining the program is trapped into deadlock, the handler will fork another process with possibly modified *worker* configuration. Consequently, *no matter how many times deadlock may occur, the production will eventually terminate with production goal satisfied.* This is true in theory based on the observation that the proposed method is *guaranteed* to complete successfully on single Worker case. Hence the handler would decrement the number of workers on forking, given it is not one; or two, based on the specific configuration. In practice, as commented above, we do not observe any invocation: the proposed method would terminate normally without resorting

to the DeadlockHandler.

As an illustrative example, we consider the 15-space scenario again. After intentionally disabling the Scheduler logic, the handler would fork (Figure 6d) and eventually complete execution at a slightly lower allowance of workers (Figure 6c). Noted throughout the execution, at any instance, at most the prescribed number of threads are running concurrently though they may indeed created by a new process. This poses no violation to the requirements.

2.6 Summary

We conclude this subsection by recalling several key features of the proposed algorithm.

- Robustness

  The proposed method performs very well when the resources (spaces, workers) are highly limited. The built-in deadlock prevention (through pre-computation, and dynamical delegation via signaling), deadlock detection (through sanity check) form the integral parts of the proposed algorithm. The backup deadlock handling mechanism guarantees in theory that the program would eventually jump out of the trap and finish execution correctly.

- Scalability

  The proposed method scales decently well when the resources are abundant. This is due to the dynamical nature of the algorithm and the meditation of the Scheduler and Dispatchers.

3 Experiement and Results

In this section, we report our thorough simulation results with different job configurations. Unless otherwise specified, we compile our program with `O2` flag and `DEBUG` mode off. The simulations are conducted on two shared `Linux x86-64` server, with 125 GB memory and two ten-core Intel Xeon 2.20GHz CPUs.

Concretely, denote by s the number of spaces, w the number of workers, n the number of cars and t time incurred, we enumerate from $w \in [1, 2, \ldots, 20, 30, 50, 64, 72, 86, 100, 120]$, $s \in [1, 2, \ldots, 20, 30, 40, 50, 100]$ and $n \in [1, 2, \ldots, 10, 15, 30]$. Some configurations are not simulated though, for example, one worker constructing one hundred cars is not very interesting to us since we have already known it would work for producing, ten cars. This in total yields more than five thousand test cases.

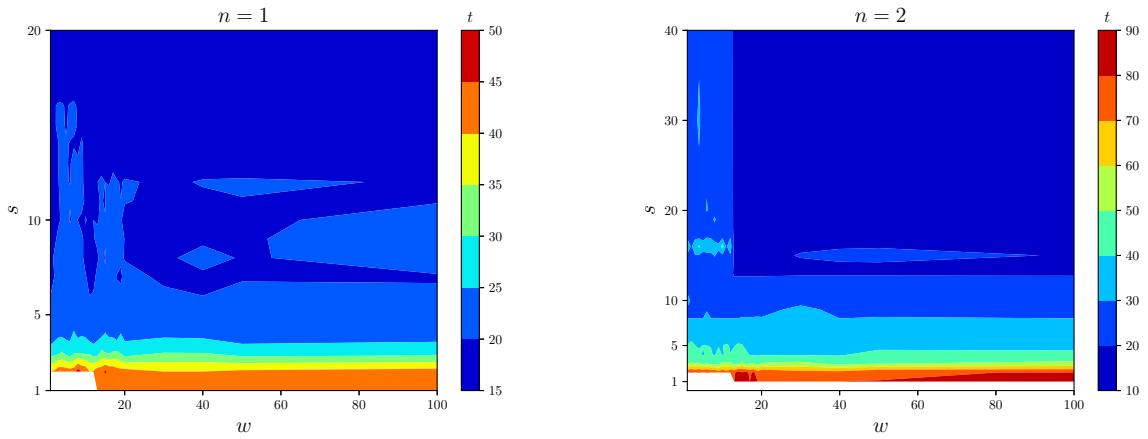
We plot the contours and surfaces under each production task in Figures 8 to 11, where we mainly focus on the trend of the production time between the combinations of the resources. We observe the following traits and characteristics of our method:

- Good Scalability.

  With different production goals, our method is able to boost performance *significantly* when the resources (i.e., workers and spaces) are abundant. This is implied by the decreasing trend of the slant diagonal across all figures. Further implication of scalability is embedded in Figures 12 to 14 and discussions therein.

- Robust Performance Guarantee.

  When given limited resource (be it spaces or workers), our algorithm is able to steadily accomplish various production goals without deadlocks. Nonetheless, we point out that when the production goal increases, the limitation of spaces posts more threats than the constraints on the labour force, as implied by the peaks at low space regions across w axis (Figures 10d and 11c for example).

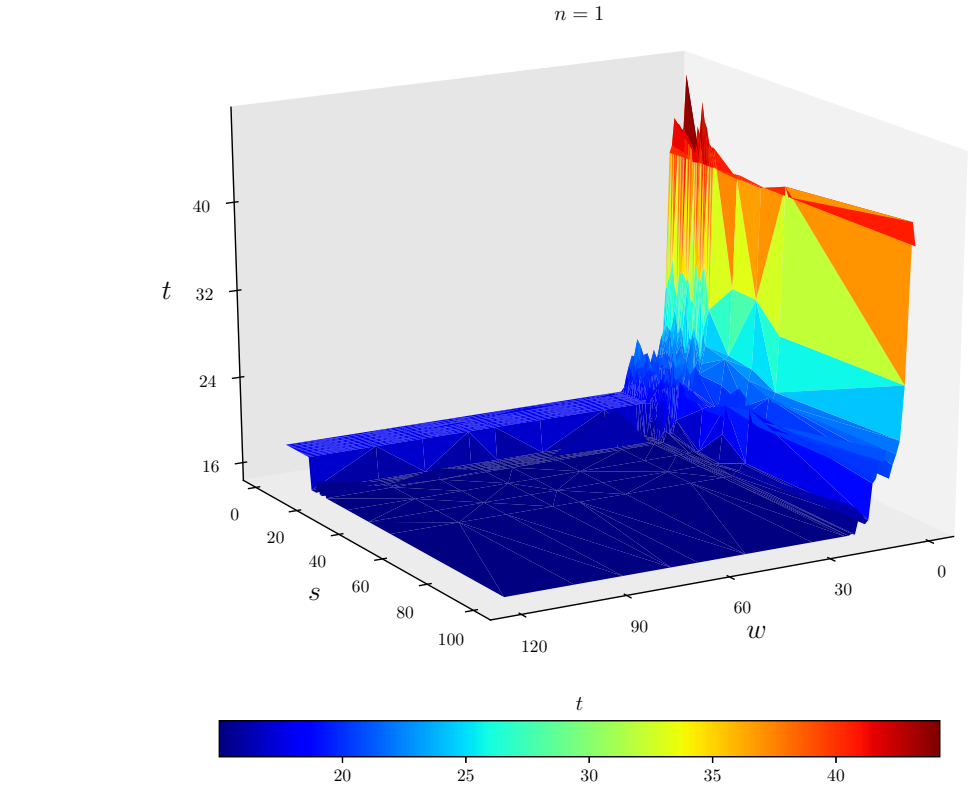


(a) $n = 1$ (b) $n = 2$

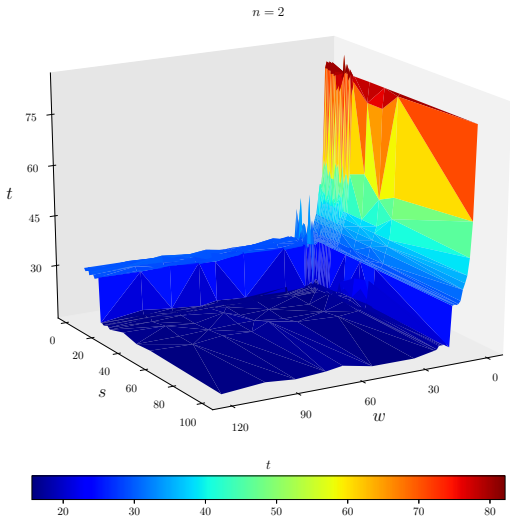
Figure 8: *Best viewed in color.* The contours plots of selected simulation results.

4 Concluding Remarks

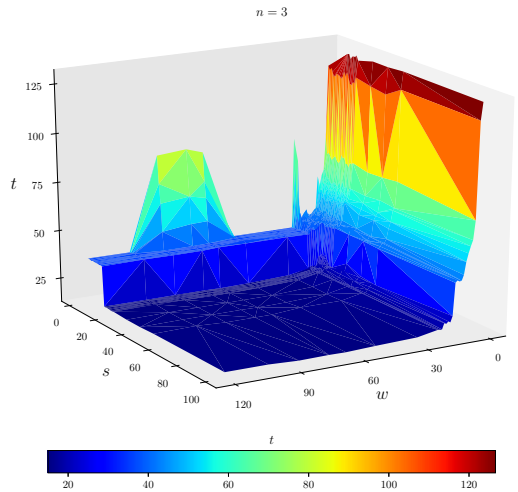
In this writing, we present a robust, scalable algorithm for dynamic job allocation with various dependent resources constraints. Through thorough benchmarking and theoretical analysis, we have shown two key characteristics of the proposed algorithm. We also identify several limitations from the results: for example, given a reasonable amount of resources, e.g., eight workers and twenty spaces, our method may be few seconds slower than previous methods. This is partly due to the overhead incurred by context-switching and the mediation between Scheduler and Dispatchers. Furthermore, our WorkerPool is implemented as a vanilla array, which may be the bottleneck to the proposed method when we have tens or thousands of workers. Yet, this issue can be readily ameliorated by implementing a tree structure for fast indexing. We feel, nonetheless, content about the sacrifice of few seconds in return of the robustness and simplicity, but would suggest these be addressed in future work.



(a) $n = 1$

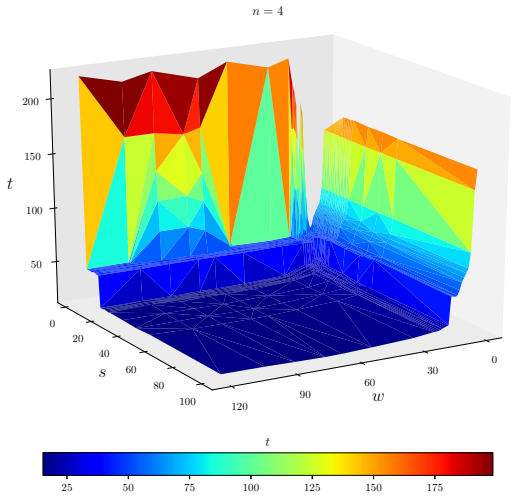


(b) $n = 2$

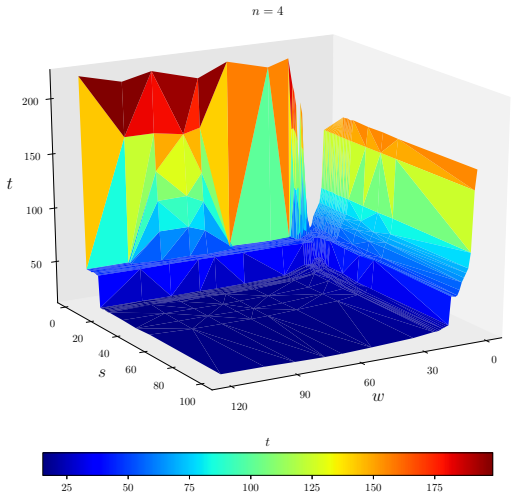


(c) $n = 3$

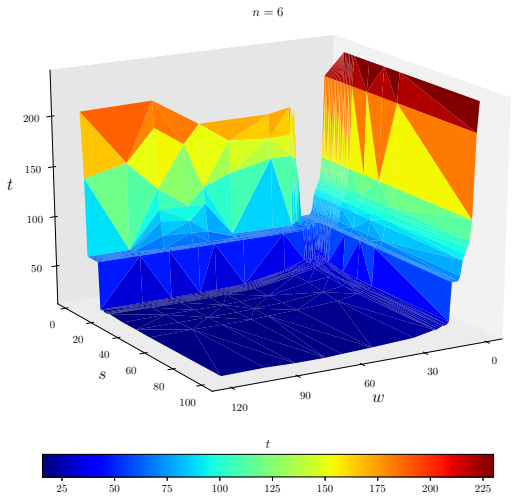
Figure 9: *Best viewed in color.* The surface plots of the simulation results (I).



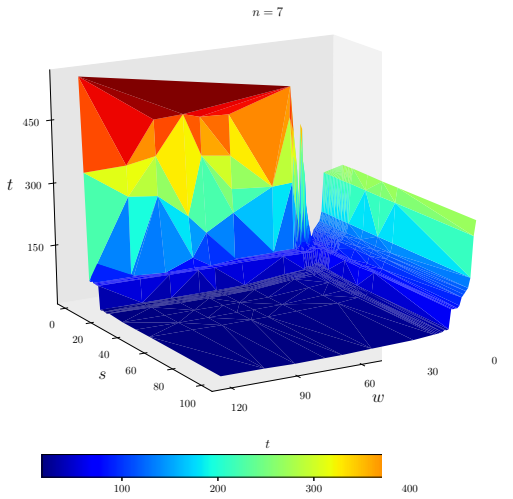
(a) $n = 4$



(b) $n = 5$

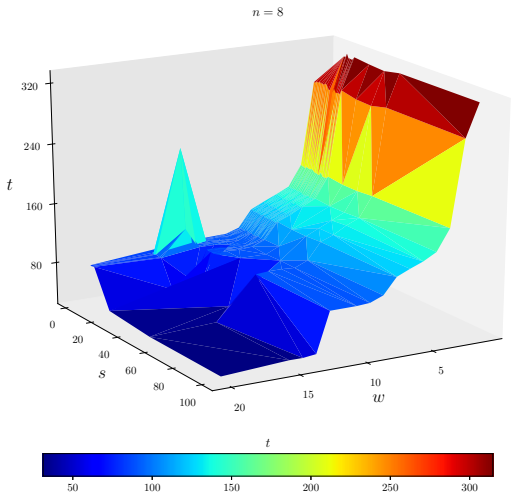


(c) $n = 6$

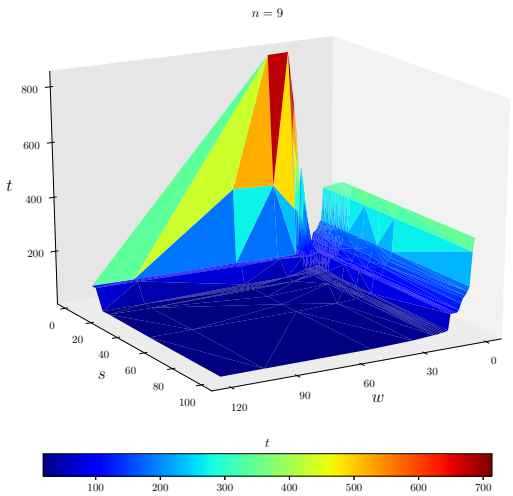


(d) $n = 7$

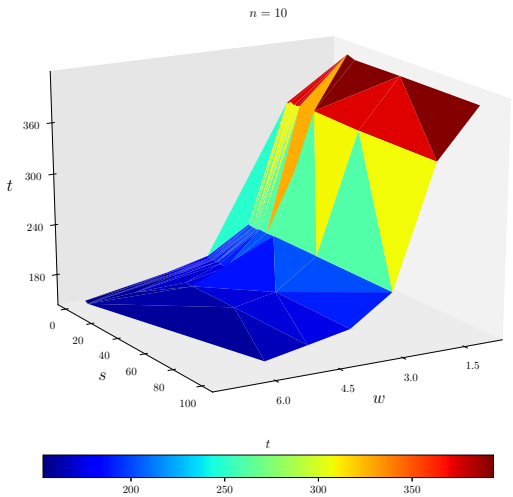
Figure 10: *Best viewed in color.* The surface plots of the simulation results (II).



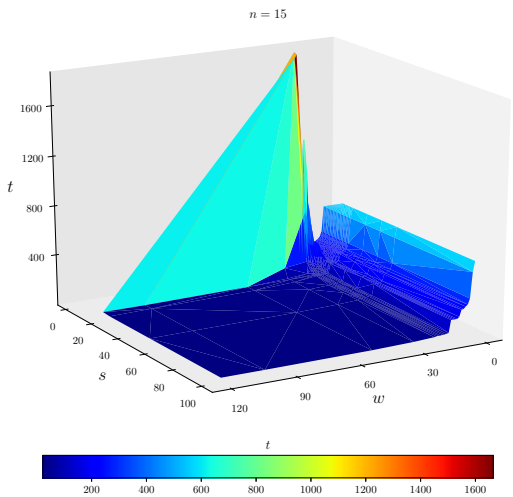
(a) $n = 8$



(b) $n = 9$



(c) $n = 10$



(d) $n = 15$

Figure 11: *Best viewed in color.* The surface plots of the simulation results (III).

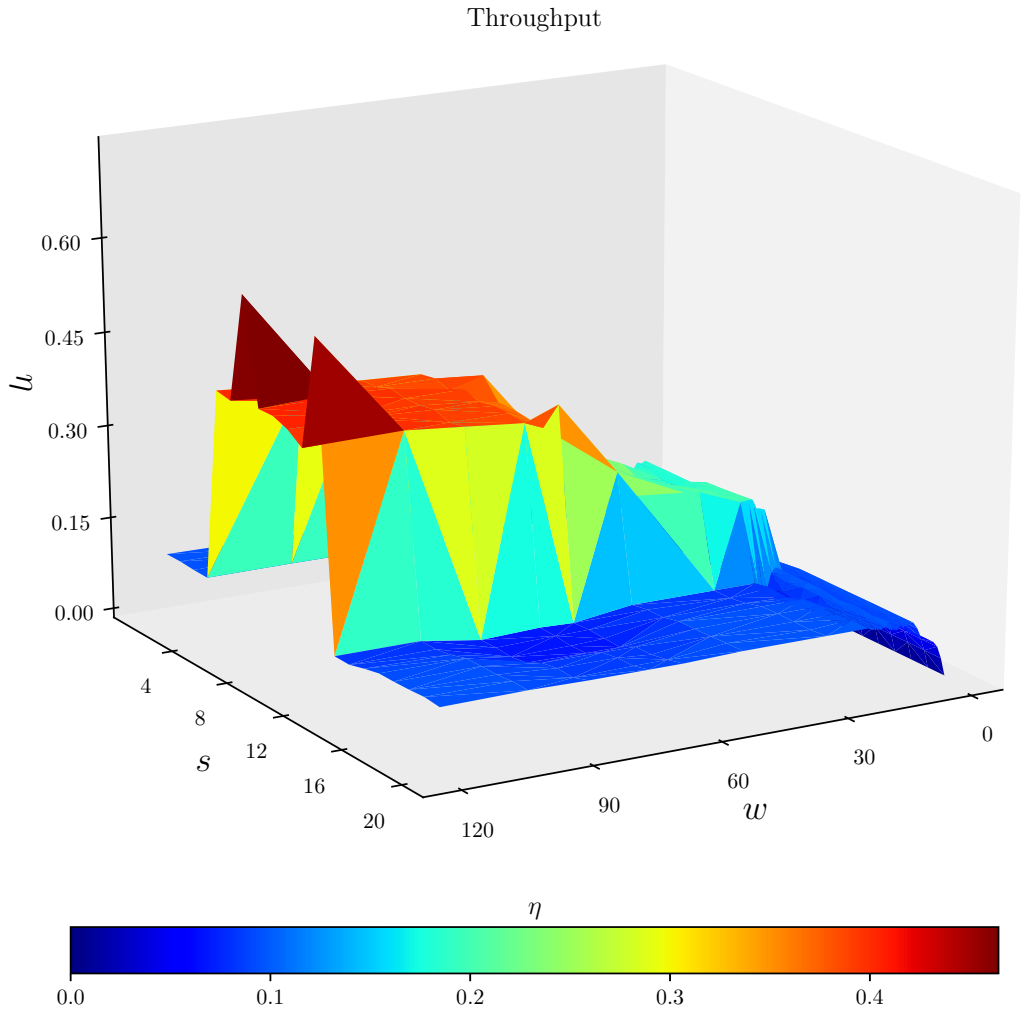


Figure 12: *Best viewed in color.* Maximum throughput of the proposed method. Noted the plotter automatically extrapolates the plateau among the edges of the surface, which should be ignored. Noted the increasing of the throughput when resources are increasing, i.e., along the main diagonal of the WORKER-SPACE plane. Since the throughput increases with the production goal when resources are unlimited, and our simulation only resulted from a medium size of goals, we predict the throughput will increasing as we increase the production goals with ample resources. This is further justified by Figures 13 and 14

.

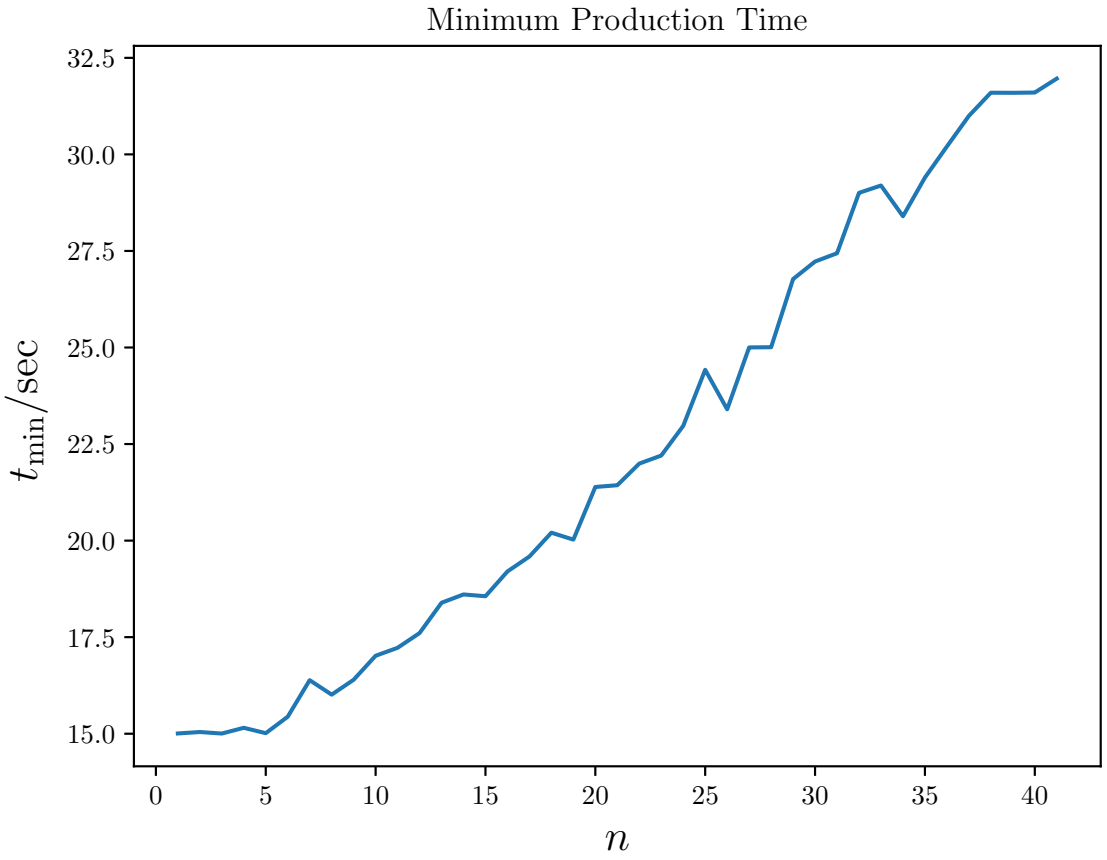


Figure 13: Given a respectable amount of resources (120 workers and 100 spaces), the increasing in production time with respect to production goal is insignificant. This suggests the proposed method scales very well when the resources are not unlimited.

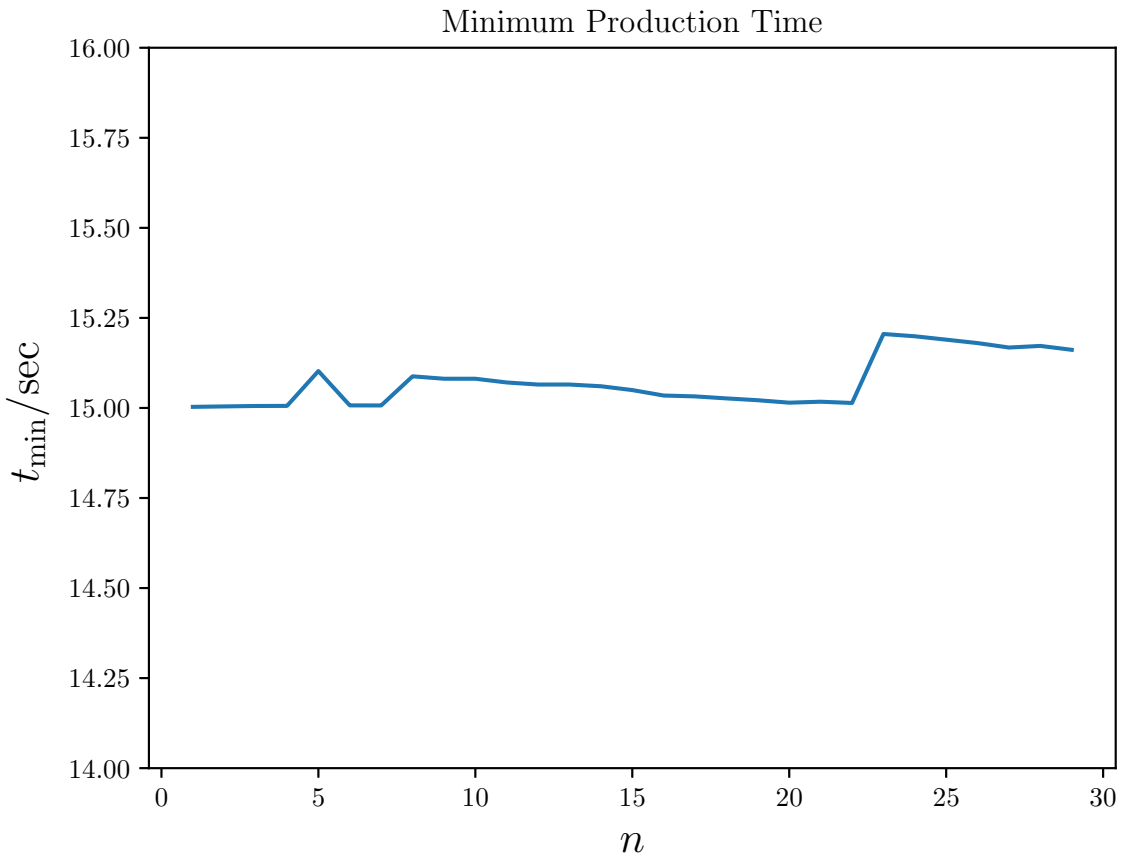


Figure 14: We simulate the unlimited resources case by approximating using about one thousand workers and one thousand spaces. We observe our method maintains an optimal performance of a total production time around fifteen seconds. This suggests a boost in the throughput and good scalability of the proposed method. Noted, however, due to server load and restriction, we failed to simulate production goals above thirty with abundant resources.