# Core Scheduler Task Duplication for Multicore Multiprocessor System

Aya A. Eladgham, Nesreen I. Ziedan, Ibrahim Ziedan
Computer and Systems Engineering Department-Faculty of Engineering, Zagazig University, Egypt

***Abstract*—The increasing complexity of multi-core multiprocessor systems presents significant challenges in task scheduling. The scheduling of tasks across multiple cores remains a significant challenge due to its NP-complete nature, especially with the increasing complexity of multi-core / multi-processors architectures. This paper focuses on Multi-Core Oriented (MCO) scheduling algorithms, which specifically target multi-core multi-processor systems. This paper proposes a novel scheduling algorithm, Core Scheduler Task Duplication (CSD), specifically designed for multi-core multi-processors environment. The CSD algorithm combines static and dynamic task prioritization to enhance processor utilization and performance. The proposed algorithm clusters related tasks to the same cores to improve efficiency and reduce execution time. By leveraging task duplication, the proposed algorithm improves processor utilization and reduces task waiting times. To evaluate the CSD algorithm's performance, the algorithm was implemented and compared against the Modified Critical Path (MCP) scheduling algorithm. A series of experimental tests were conducted on diverse task sets, varying in size and complexity. Simulation results demonstrate that CSD outperforms existing compared approaches in task scheduling and processor utilization, making it a promising solution for multi-core systems.**

***Keywords*—MultiCore; multiprocessor; DAG scheduling; dynamic priority; task duplication; clustering; MCP**

## I. Introduction

With the increasing demand of high-performance and low energy consumption processing, which is a basic requirement in many applications such as image and video processing [1] [2] [3] [4], climate modeling [5] [6] [7], artificial intelligence [8] [9] [10]. Parallel processing is needed to speed up applications performance by splitting its overall job into smaller tasks [11] [12] [13], execute and complete its work across multiple processors. There is an increasing interest in addressing issues related to multi-core chips. The shift to multi-core has emerged because of reaching the physical limits of single core chips, especially clock speed bottleneck [14] [15]. In the last few decades, multi-core processors have evolved from just two cores in a single CPU to multiple cores [11] [12]. It is challenging to find a computer with single-core CPU, as even low-power CPUs are now designed with two or more core per chip [16]. Intel already launches Intel® Xeon® 6 server processor, code-named Sierra Forest with up to 144 cores [17].

In traditional multiple processors, scheduling problem appears to solve the contention between concurrent parts of programs, or arrange programs execution to guarantee enhancement in the overall performance [13]. In multi-core processors the scheduling problem gets worse with the presence of many cores, where a program can be seen as a set of tasks which can run serially or parallelly. The relationship among these tasks may or may not include precedence constrains [11] [12]. Precedence constrains indicates when one task begins or ends in relation to another task [13]. If precedence constrains exist, a Directed Acyclic Graph (DAG) is used to build a task model [12] [13] [18].

The presence and use of multi-core processors is more popular than the traditional multiple processors. In multi-core processors, the scheduling problem is magnified, which is a non-deterministic polynomial (NP-complete) problem [19] [20]. Most of the traditional parallel processing algorithms are designed to handle one-core processors or designed for multi-core processor, but it does not fit multi-core multi-processors [18]. The primary goal of all these algorithms is to try to reduce the program execution time. The DAG based Heuristic algorithms, which can be divided into four categories, which are List based task scheduling algorithms [21] [22] [23] [24] [25] [26], Task Duplication-based scheduling algorithms [27] [28] [29], Cluster based scheduling algorithms [21] [30] [31], and Multi-Core Oriented scheduling algorithms (MCO), which specialize in the types that deal with multi-core multi-processors machines.

Multi-Core Oriented scheduling algorithms are the focus in this paper. Despite the many advantages and the ongoing manufacturing of multi-core multi-processors systems [17]. MCO scheduling algorithms are not independent types, but they apply concepts and methods from previous types targeting multi-core multi-processors systems (MCMP). This paper introduces some attempts.

The utilization of multi-core processor architecture is growing more common in the realm of high-powered computing. This is considered one of the reasons for the emergence of the MCO scheduling algorithms. Some examples of MCO scheduling algorithms are weighted Earliest Finish Time (wEFT) [32], the Priority Queue Task Duplication scheduling algorithm (PQTD) and Genetic-based Scheduling Algorithm on Multicore (GSAM) [33].

The wEFT [32] algorithm is designed for multi-core processor systems, where it assigns the task with minimum earliest completion time to a certain processor core. wEFT performs well when compared with existing task scheduling algorithms, but wEFT is not the best in average waiting time of the tasks. The PQTD is proposed in [18] for multi-core processors. PQTD uses priority queue and task duplication concepts to map the generated task model to processors. As mentioned in [18] The PQTD algorithm has better performance and better processor utilization compared to TDS [34], and CPFD [35].

The GSAM [33] is try to take advantage of multi-core multi-processors and provided solution for the scheduling problem based on genetic approach. The algorithm is repeatedly executed until it reaches a fixed number of iterations. The GSAM algorithm inherits some defects form GA and multi-cores, such as high computation and time consumption. In addition, GSAM suffers from high power need in multi-core architectures. Existing GA based scheduling algorithms have some disadvantages such as high complexity, high power consumption, poor efficiency, poor processors utilization, etc. [36] [37] [15]. The GA and multi core algorithms are intended for some specific applications and are not suitable for other applications.

The proposed scheduling algorithm combines the qualities of multi-core processors algorithms and multi-processors. The proposed algorithm deals with fine grain task graph applications. It overcomes some problems and combines some of the advantages found in others. It increases the processor utilization and tries to cluster the related tasks to run in the same processor cores to improve the performance. The proposed algorithm first uses static priority for level categorization, and then dynamic priority for tasks in each level while assigning tasks. The proposed algorithm Core Scheduler task Duplication, which is CSD.

The remainder of this paper is organized as follows. The proposed scheduling algorithm is presented in Section II. Section III introduces an application example illustrating the proposed scheduling algorithms steps. Section IV provides the simulation results and discussions. Finally, the conclusion is provided in the last section.

## II. The Proposed CSD Scheduling Algorithm Design

### A. Task Mode

The properties associated with any parallel program such as processing time, data dependencies, communication cost, and synchronization requirements must be known before scheduling. The parallel program is represented by node and edge DAG [13].The DAG task model G = {T, E, C, W} where the set of nodes T is a set of n tasks, where n is the total number of tasks. E is a set of edges in the DAG between two vertices (nodes) Ti, and Tj,where $0 \leq$ i,j $\leq$ n. Cij is as set of communication time between two tasks, where $0 \leq$ i,j $\leq$ n. W is a set of processing time of each task. Fig. 1 is an example of DAG with four tasks numbered. from T1 to T4, T1 has processing time equal to 2 unit of time. T1 and T2 are connected with an edge with communication weight equal to 1 unit of time.

### B. Assumptions and Constraints

Some restrictions are necessary to explain the CSD scheduling algorithm, which are as follows:

- CSD targets a machine with a set of n homogenous Processing Elements (PE), where each PE contains a set of m homogenous cores idiomatically called (n) multiprocessor (m) multi-cores system. The core j inside Processing Element i is called (PEi-ci).

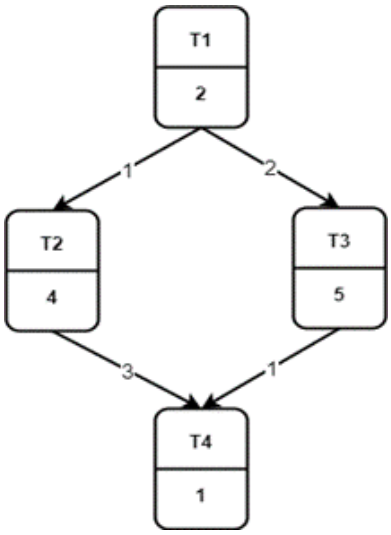


Fig. 1. Example of Directed Acyclic Graph (DAG).

- The processors are fully connected, where link contention and routing strategies used for communication are neglected.
- The communication delay Cij between two cores in the same processor element is negligible (Cij $\approx$ 0), while it takes cij units of time if the two cores belong to different processor elements.
- Any task cannot be started until all its predecessors are completed.
- Assignment procedure will be initiated if either any core is free or the task has been completed, which leads to a free core.
- Transition from any level to another is not allowed until the assignment of all tasks at this level are completed.

### C. CSD Scheduling Algorithm Procedure

CSD scheduling algorithm is formalized in this section.

**Input:**

- A DAG describes tasks workloads, their weights, communication cost between any tasks, and precedence relation.
- The number of PE and how many cores each one contains.

**Output:** Gantt chart illustrates the list of all tasks allocated to each core ordered by their starting, finishing, and execution time. Gantt chart also indicated Idle periods for every core.

**Steps:** Fig. 2 shows the steps of CSD algorithm, while the part concerned with selecting task from unassigned level list is illustrated in Fig. 3. These steps are listed as follows:

1) Arrange the given M cores of each processing elements PE lexicographically in a queue.

2) Build the static task queue, where this queue is divided into static priority levels starting from Level1 (L1) to Levelm (Lm). Tasks belonging to any level Li only depend on the tasks belonging to the previous level Li-1. Tasks arranged at this point have no order. Only the tasks belonging to the same level are stacked together with no priority. Each level is rearranged during the assignment procedure step and accordingly the priorities of the tasks are changed during the successive stages of CSD algorithm within the same level. Because of this change, the task static priorities turn to dynamic priorities.
3) Collect and compute all the following characteristics for each task before the assignment procedure:
   - weight of the task i, $0 \leq i \leq n$ (Wi),
   - list of the predecessors of the task, i.e. parents (predi),
   - successors of the task, i.e. children (succi),
   - the task should be duplicated or not and how many times it should be duplicated (D) according to Eq. (1).

$$\min(\#ofPEs, ceil(\frac{\#ofchildren}{\#ofcores})) \quad (1)$$

   As the number of tasks in the same level increases, the congestion problem emerges and worsens with the duplication according to the (1). This problem was resolved using Eq. (2).

$$\min(\#ofPEs, floor(\frac{\#ofchildren}{\#ofcores})/\#oftasks) \quad (2)$$

   - The task found in critical path (CP) in the DAG, which is the longest path in the DAG or not,
   - degree of the CP, which is number of tasks in the CP, the more tasks the higher the degree (DCP), and located on more than one CP.
   - Select the tasks that can be duplicated if possible and mark it as duplicated task. The selection process satisfies the inequality: # of children $>$ # of cores.
   - Any task chosen for duplication is indeed duplicated in the task queue many times as Eq. (1) or Eq. (2).
4) Assign previously arranged tasks to cores: this step is called assignment procedure, which assigns the tasks to cores and is carried out periodically in every time slot. The procedure is divided into two phases. The first phase is called **core phase** and deals with the cores. Meanwhile the second phase is called **task phase** and deals with the selection of the task to be performed. Each phase is explained as follows.
   **Core phase**: Roll over all free cores to see if they can start any selected task from the next phase. It should be noted that the list of free cores changes every time slot, where the algorithm picks all free cores. The CSD algorithm determines, which core suitable for the priority based ordered tasks.
   **Task phase**: Priorities change at this phase based on a number of criteria to choose the task, which has the highest priority. These priorities change periodically with each start of task selection. CSD considers all the tasks in the same level and rearrange them again, then choose the task that will be executed if:
   • One of its predecessors is completed, and none of its duplicated is running in the same processing element PE. If a tie occurs, then go to the next step.
   • When CSD algorithm reaches this step, there is more than one choice, which are:
   **First**: There is only one task in CP,
   **Second**: There are two or more tasks in CP.
   **Third**: no task in CP.
   For the first case the algorithm chooses this task to be scheduled. For the second and third track, the algorithm calculates the longest path of Ti where the task is on (LPi). The LPi can be calculated using Eq. (3):

$$LPi = longestpathbefore + longestpathafter - Wi \quad (3)$$

   for each task, if two or more tasks have the same LPi length, then the algorithm calculates Forest Cost of task i (FCi). FCi is the number of edges in the following subgraph for this task. If tie occurs again then the algorithm moves to the next action.
   - Choose the task with greater number of children if tie occurs then.
   - Choose the one with lower weight, if tie occurs then.
   - Choose task with small index.

   After selecting the task, the CSD determines if the task is duplicated and how many times. CSD has to decide whether to start all the duplicated tasks all at the same time at available free cores and remove the exceeded copies from task queue. The other choice for CSD is to start the duplicated task any time as there are free cores.
   Repeat step 4 till each free core from core phase gains a task. Consequently, the same steps are repeated and moved from one level to another until the assignment procedure is completed for all tasks.

## III. An Application Example

The following example illustrates how CSD algorithm works.

Assume that the DAG shown in Fig. 4 is given. The objective is to schedule this DAG to 2 PE with 2 cores each. The DAG has 10 nodes, 14 edges, and with CP equal to 13. In the Fig. 4, CP is shown with thick arrows.

1) Arrange the PE, with its cores in processor queue as shown in Fig. 5.
2) Arrange tasks in static levels as shown in Fig. 5.
3) collect all the information about each task in advance as Shown in Table I.
4) Assignment procedure:
   - L1 has only one node (entry node) which is marked as duplicated (2-times) on different cores (PE0-c0), and (PE1-c0).

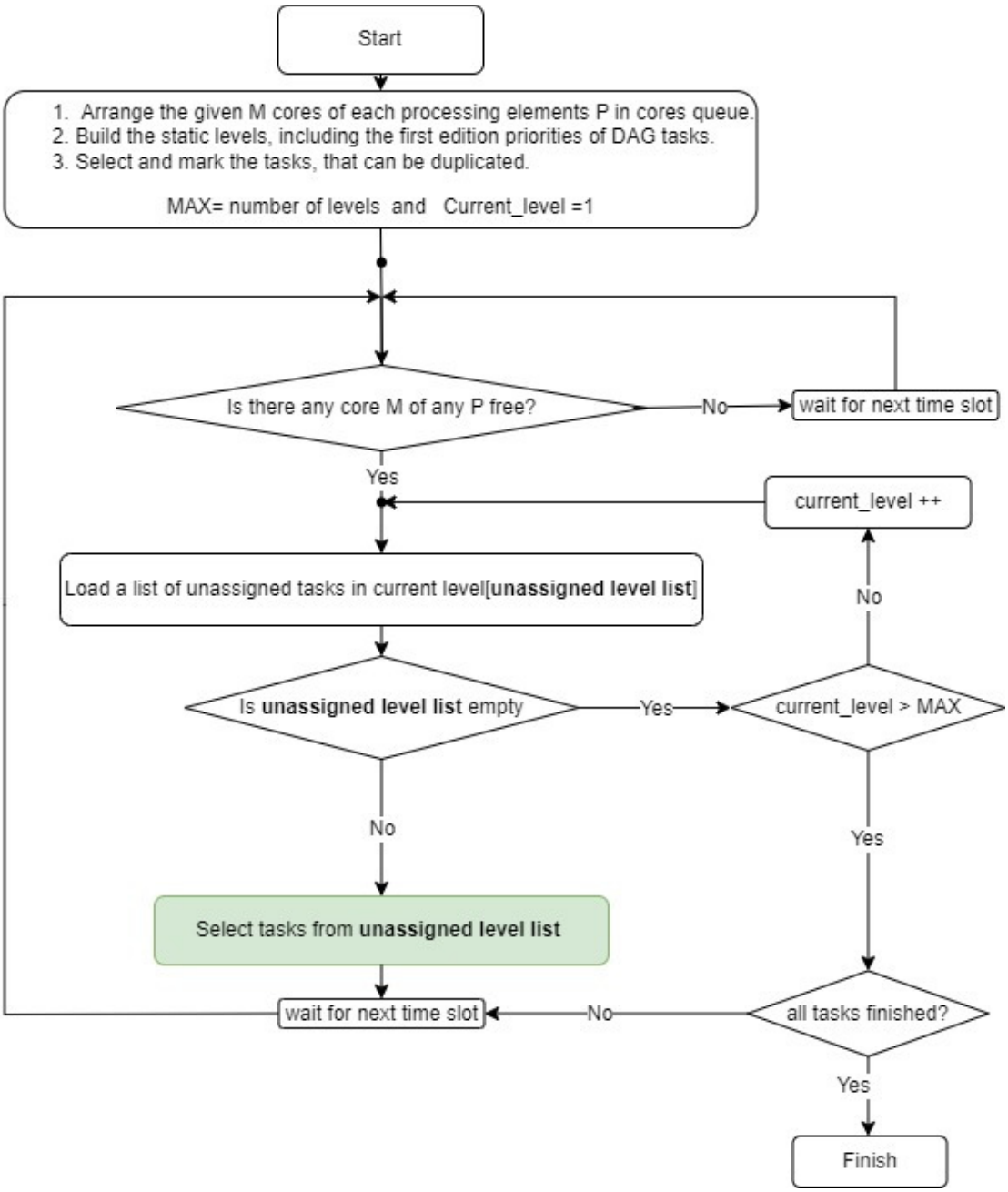


Fig. 2. CSD Scheduling algorithm flowchart.

- The assignment procedure suspended till T1 finished.
- CSD algorithm dynamically rearranges tasks in L2 to get ready_list1, followed by ready_list2: Ready_list1: T2, T3, T4, T5
  Ready_list2: T2, T3, T4, T5
  Final arrangement in this step: T3, T2, T4, T5
- CSD algorithm assigns T3 to (PE0-c0). At the same time slot, it assigns T2 to (PE0-c1), T4 to (PE1-c0), and T5 to (PE1-c1). T4 is marked as duplicated, but there are no free cores available for duplication. At this moment all cores are busy, so CSD waits until one becomes free. The beginning of time slot 4 (PE0-c0), and (PE1-c0) will be free.
- At this moment when L2 is finished CSD algorithm goes to L3, but there are no tasks ready. Therefore, the algorithm waits until the next time slot.
- The algorithm rearranges tasks in L3 until it reaches the final arrangement according to the illustrated criteria in the algorithm procedure. The Final arrangement became T6, T7, T8.
- The algorithm assigns T6 to (PE0-c0).
- T7 is assigned to any of the following (PE0-c1) or (PE1-c0) or (PE1-c1), where (PE0-c1)

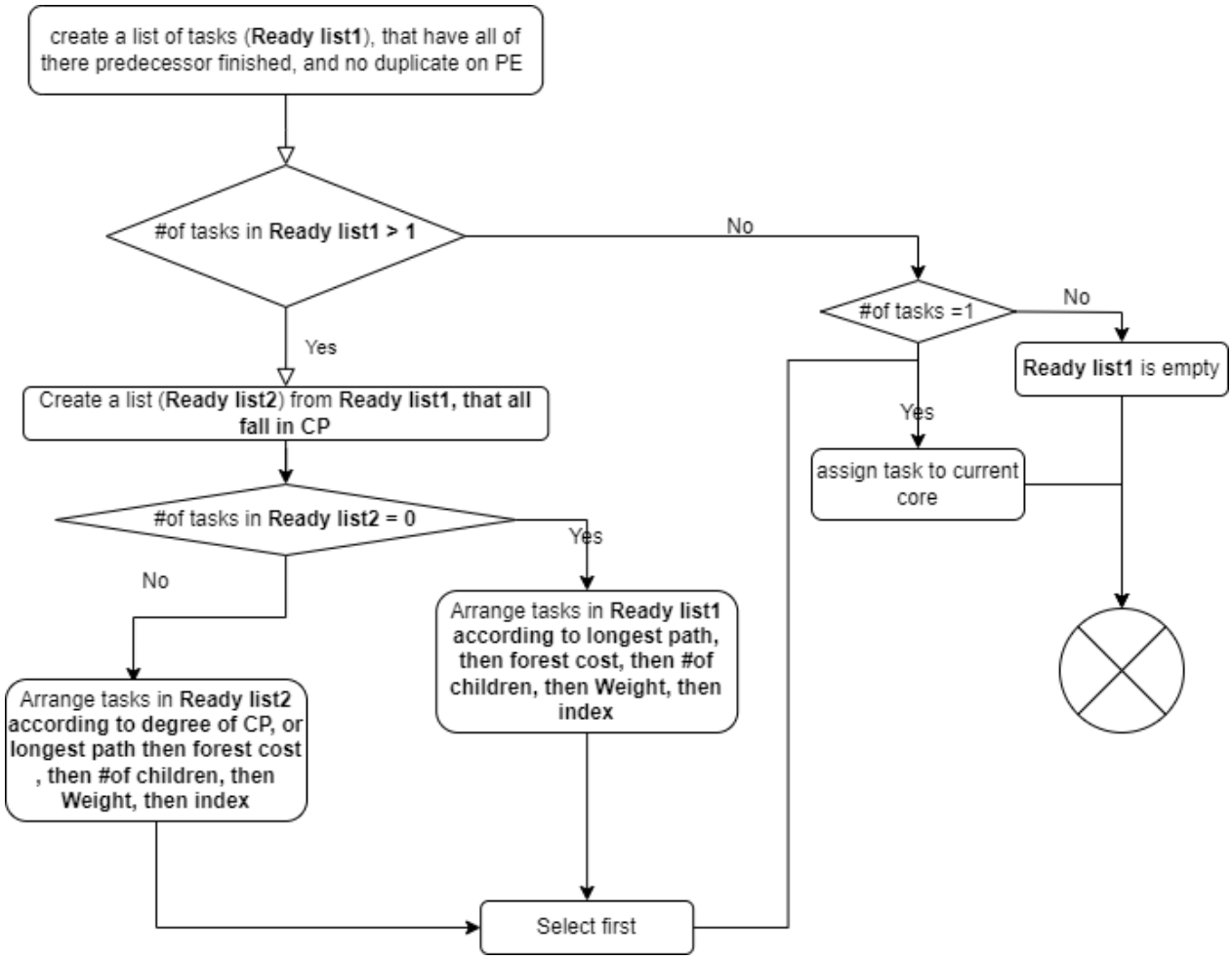


Fig. 3. Task selection flowchart.

TABLE I. TASK CHARACTERISTICS

| | T1 | T6 |
|---|---|---|
| ID | 1 | 6 |
| Wi | 1 | 2 |
| Duplicated | yes | no |
| D | 2 | 2 |
| predi | 2 | 2 |
| succi | – | 2 |
| found in CP or not | yes | yes |
| DCP | 5 | 5 |

starts after 2 time slots from time slot 4, and (PE1-c0) or (PE1-c1) start immediately, so assign T7 to (PE1-c0).

- T8 has similar situations as T7, so the algorithm assigns T8 to (PE1-c1).
- (PE0-c1) is free, and L3 is finished, but T9 cannot start until T6 is finished, so the algorithm waits.
- When T6, T7, and T8 are also finished, then all cores are free. However, the most suitable core is (PE0-c0), so the algorithm assigns T9 to (PE0-c0).
- T10 cannot start until all predecessors are finished (T9, T7, and T8). Once finished, all cores become free. Either PE0 or PE1 starts the execution of T10 at the beginning of the 8th time slot, so all cores have the same chance to execute it, so the algorithm assigns T10 to (PE0-c0).
- The output of the previous steps represented in Gantt chart shown in Fig. 6. The Gantt chart represents the Schedule Length (SL). SL is the total duration or makespan of the schedule, which emphasizes all the tasks is completed.

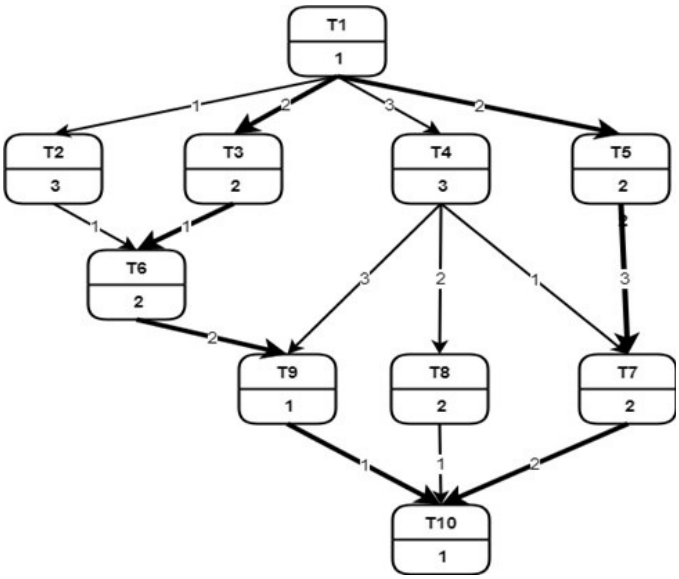


Fig. 4. A sample DAG from [18].

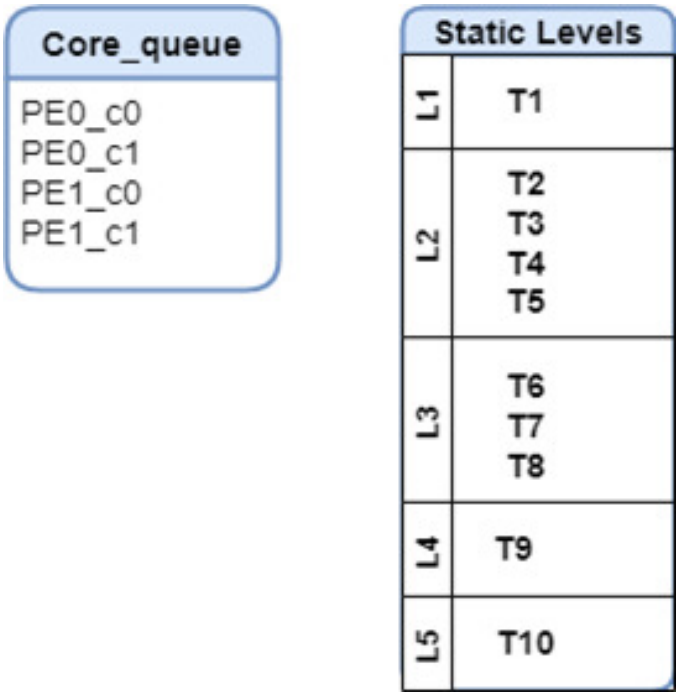


Fig. 5. Arrangement of core queue, and static task levels.

## IV. Simulation and Results

The performance of the proposed scheduling algorithm is evaluated using randomly generated task graphs or task graphs modeled from actual application programs from Standard Task Graph set, which is presented in [38].

1) The first task graph t50_rand1 is composed of 50 tasks and 985 edges with computation to communication ratio (CCR) equal 0.1.
2) t100_rand2 composed of 100 tasks and 3943 edges with CCR equal 0.02.

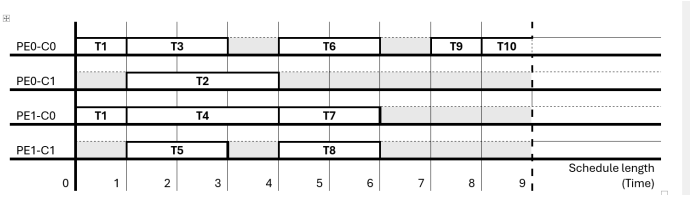


Fig. 6. The schedule generated by CSD algorithm.

3) and robot control application with 88 tasks and 131 edges with CCR equal 0.85 and 5.

The CSD scheduling algorithm simulation is conducted using Python running several times with various processing elements / cores settings. Fig. 7 presents the simulation results of the CSD scheduling algorithm with varying numbers of Processing Elements (PEs), each containing only one core. The x-axis represents the number of PEs, while the y-axis indicates the Schedule Length (SL). The figure also compares the performance of different versions of the CSD scheduling algorithm with the Modified Critical Path (MCP) scheduling algorithm. The difference between the multiple versions of CSD is when to duplicate the chosen task to be duplicated and how many times:

- If duplication occurs during the first copy assignment process at the same level according to (1) Version CSD (once_nolvl) is generated.
- If duplication occurs any time according to (1) Version CSD (any_nolvl) is generated.
- If duplication occurs during the first copy assignment process at the same level according to (2) Version CSD (once_lvl) is generated.
- If duplication occurs any time according to (2) Version CSD (any_lvl) is generated.
- CSD (none) is generated when no duplication occurs.

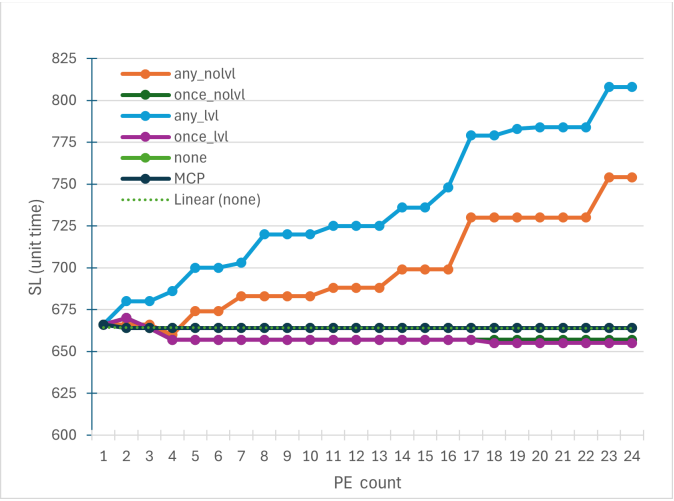


Fig. 7. Scheduling for t50_rand1 DAG.

As seen in Fig. 7 the SL is improved with the CSD (once_lvl and once_nolvl) versions compared to other algo-

rithms. CSD (once_lvl) outperforms CSD (once_nolvl) when the number of tasks in the same level increases. The performance of CSD (none) algorithm coincides with MCP algorithm. Both CSD (none) and MCP algorithms outperform all the reset versions of CSD. While the CSD (any_nolvl) and (any_lvl) deviated too far as it exhausted the free cores in duplication while there are already tasks to begin execution. This scenario is reflected in a significant increase in SL. CSD(Any_nolvl) and CSD(any_lvl) and the results are ignored. Fig. 8 illustrates SL for t100_rand2, which is like t50_rand1. It is noticeable that there is an improvement of CSD (once_lvl) over MCP, as the number of parallel paths increase. Fig. 9 illustrates the average SL output behavior with robot control application generated with CCR equal 0.85 and 5. Robot control DAG in contrast to other DAGs shows a significant improvement performance of MCP over all version of the proposed CSD. This is due to the decrease in parallel paths, which is the primary motivation and cause for duplication.

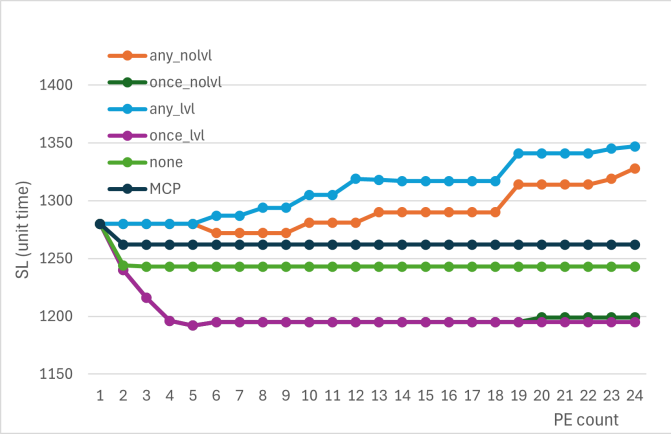


Fig. 8. Scheduling for t100_rand2 DAG.

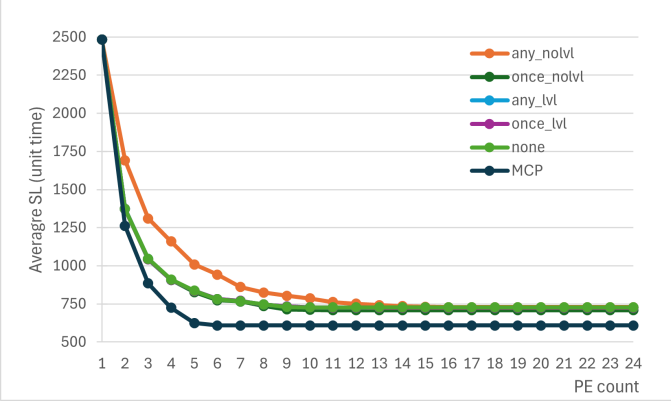


Fig. 9. Average SL for robot control with (0.85,5) CCR.

The CSD scheduling algorithm is simulated again with one PE only that has varying number of cores. Fig. 10 shows the SL with robot control DAG described before. The chosen DAG indicates almost the same response as robot task graph when applying this PE/Cores configuration. The comparisons here are between the versions of CSD and the optimal SL. Optimal SL is the length of CP without communication. The primary goal of the CSD algorithm is to reduce SL by clustering the tasks related to others and give higher priority to CP tasks. CSD algorithm closely approach the optimal demonstrating its effectiveness in finding near-optimal SL.

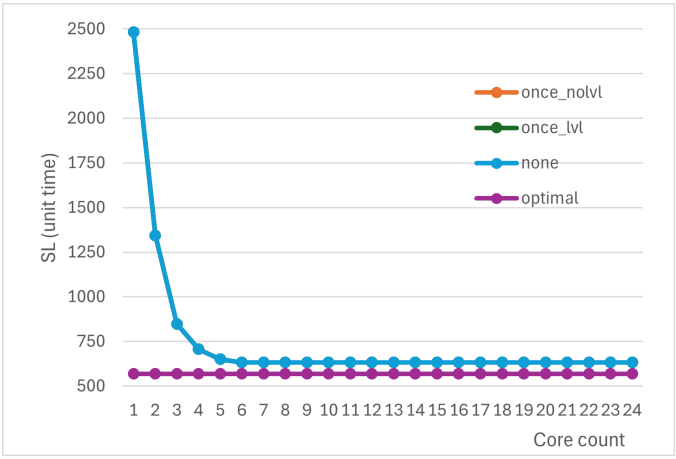


Fig. 10. Scheduling for robot control 88 tasks DAG.

Fig. 11 shows the result when the CSD algorithm is simulated with different number of PEs that have varying number of cores. Given that the total number of processing units (i.e. PE or core) is fixed and equal to 24. xxx shows the average SL for CSD versions on all the task graphs with different CCR. CSD algorithm produces schedule close to optimal as the number of cores increase.

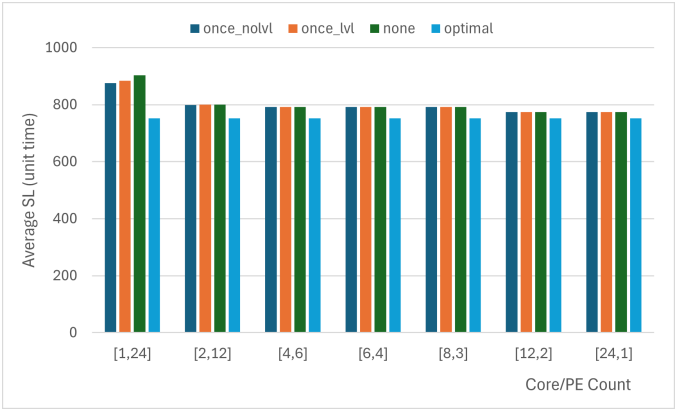


Fig. 11. Average SL for all task graphs.

## V. Conclusion

This paper proposed multiple versions of a scheduling algorithm called CSD. Each version introduces enhancements and modifications that change with the workload behavior. CSD in its primitive version with no duplication matches MCP in the case the used architecture has only one core with different PEs. When comparing the (once_nolvl) version of CSD with MCP, the proposed algorithm outperforms MCP in SL, throughput and processors utilization. The results indicate that no single algorithm outperforms others in all scenarios. Instead, the effectiveness of a scheduling algorithm is dependent on factors

such as the nature of workload and architecture of multi-processor/multi-core system.

## REFERENCES


[1] A. Kika and S. Greca, "Multithreading image processing in single-core and multi-core cpu using java," *International Journal of advanced computer science and applications*, vol. 4, no. 9, 2013.

[2] K. M. Hosny, A. Salah, and A. Magdi, "Parallel image processing applications using raspberry pi," in *Recent Advances in Computer Vision Applications Using Parallel Processing*. Springer, 2023, pp. 107–119.

[3] A. Kamalakannan and G. Rajamanickam, "High performance color image processing in multicore cpu using mfc multithreading," *International Journal of Advanced Computer Science and Applications*, vol. 4, no. 12, pp. 42–47, 2013.

[4] K. Mia, T. Islam, M. Assaduzzaman, T. M. N. U. Akhund, A. Saha, S. P. Shaha, M. A. Razzak, and A. Dhar, "Parallelizing image processing algorithms for face recognition on multicore platforms," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 11, 2022.

[5] T. Radhika, K. Gouda, and S. S. Kumar, "Novel approach for spatiotemporal weather data analysis," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 7, 2022.

[6] L. Su and S. Naffziger, "1.1 innovation for the next decade of compute efficiency," in *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2023, pp. 8–12.

[7] J. Subha and S. Saudia, "Integrating regression models and climatological data for improved precipitation forecast in southern india," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 5, 2023.

[8] M. N. Al-Andoli, K. S. Sim, S. C. Tan, P. Y. Goh, and C. P. Lim, "An ensemble-based parallel deep learning classifier with pso-bp optimization for malware detection," *IEEE Access*, 2023.

[9] R. Pirayeshshirazinezhad, S. G. Biedroń, J. A. D. Cruz, S. S. Güitrón, and M. Martínez-Ramón, "Designing monte carlo simulation and an optimal machine learning to optimize and model space missions," *IEEE Access*, vol. 10, 2022.

[10] R. A. Jain and D. V. Padole, "Scalable and flexible heterogeneous multi-core system," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 12, 2012.

[11] G. S. Almasi and A. Gottlieb, *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., 1994.

[12] T. Rauber and G. Rünger, "Parallel programming: For multicore and cluster systems," *Citado na*, p. 30, 2013.

[13] H. El-Rewini and M. Abd-El-Barr, *Advanced computer architecture and parallel processing*. John Wiley & Sons, 2005.

[14] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 365–376.

[15] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[16] M. Gupta, L. Bhargava, and S. Indu, "Mapping techniques in multicore processors: current and future trends," *The Journal of Supercomputing*, vol. 77, no. 8, pp. 9308–9363, 2021.

[17] I. Website, "Intel." [Online]. Available: https://www.intel.com/content/www/us/en/products/details/processors.html

[18] X. Yao, P. Geng, and X. Du, "A task scheduling algorithm for multi-core processors," in *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 2013, pp. 259–264.

[19] X. Xiao and Z. Li, "Chemical reaction multi-objective optimization for cloud task dag scheduling," *IEEE Access*, vol. 7, pp. 102 598–102 605, 2019.

[20] T. Lively, W. Long, and A. Pagnoni, "Analyzing branch-and-bound algorithms for the multiprocessor scheduling problem," *arXiv preprint arXiv:1901.07070*, 2019.

[21] M.-Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE transactions on parallel and distributed systems*, vol. 1, no. 3, pp. 330–343, 1990.

[22] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*. IEEE, 1999, pp. 3–14.

[23] S. Branch and I. Shoushtar, "List-scheduling techniques in homogeneous multiprocessor environments: a survey," *International Journal of Software Engineering and Its Applications*, vol. 9, no. 4, pp. 123–132, 2015.

[24] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE transactions on Parallel and Distributed systems*, vol. 4, no. 2, pp. 175–187, 1993.

[25] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of parallel and Distributed Computing*, vol. 9, no. 2, pp. 138–153, 1990.

[26] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE transactions on parallel and distributed systems*, vol. 7, no. 5, pp. 506–521, 1996.

[27] I. Ahmad and Y.-K. K. Y.-K. Kwok, "A new approach to scheduling parallel programs using task duplication," in *1994 Internatonal Conference on Parallel Processing Vol. 2*, vol. 2. IEEE, 1994, pp. 47–51.

[28] Y.-C. Chung *et al.*, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors," in *SC Conference*. IEEE Computer Society, 1992, pp. 512–521.

[29] G.-L. Park, B. Shirazi, and J. Marquis, "Dfrn: A new approach for duplication based scheduling for distributed memory multiprocessor systems," in *Proceedings 11th international parallel processing symposium*. IEEE, 1997, pp. 157–166.

[30] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *siam journal on computing*, vol. 18, no. 2, pp. 244–257, 1989.

[31] S. Cao and J. Bian, "Improved dag tasks stretching algorithm based on multi-core processors," in *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 2020, pp. 18–21.

[32] L. Liu and D. Qi, "An independent task scheduling algorithm in heterogeneous multi-core processor environment," in *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. IEEE, 2018, pp. 142–146.

[33] J. Pecero, S. Varrette, and P. Bouvry, "Scheduling dag applications on multi-core processor packages architectures," 2010.

[34] S. Darbha and D. P. Agrawal, "Optimal scheduling algorithm for distributed-memory machines," *IEEE transactions on parallel and distributed systems*, vol. 9, no. 1, pp. 87–95, 1998.

[35] I. Ahmad and Y.-K. Kwok, "On exploiting task duplication in parallel program scheduling," *IEEE Transactions on parallel and distributed systems*, vol. 9, no. 9, pp. 872–892, 1998.

[36] R. Medina, E. Borde, and L. Pautet, "Scheduling multi-periodic mixed-criticality dags on multi-core architectures," in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 254–264.

[37] S. Shah, A. Qahir, M. Safeer, S. Mazahir, and O. Hasan, "Comfast: A comparative framework for analysis of scheduling techniques in multi-core systems," in *2018 Annual IEEE International Systems Conference (SysCon)*. IEEE, 2018, pp. 1–7.

[38] Kasahara; H.; Tobita; T.; Matsuzawa; Sakaida, "S.: Standard task graph set." [Online]. Available: https://www.kasahara.cs.waseda.ac.jp/schedule/