Evaluation of Work Stealing Algorithms

Juan Sebatían Numpaque and Nicolás Cardozo(✉)

Systems and Computing Engineering Department, Universidad de los Andes, Bogotá, Colombia
{js.numpaque10,n.cardozo}@uniandes.edu.co

Abstract. Work stealing is a common model in parallel computing used to schedule the execution of tasks. In this model, tasks are assigned dynamically to different processors as they are generated by a computation. However, to improve execution time, idle processors can steal tasks from other processors. Different underlying techniques have been proposed for work stealing, with FIFO and LIFO among the most used techniques. In this work we propose an alternative to these techniques, based on task priority, as a means to avoid fairness problems in the way that tasks are stolen across processors. To evaluate our technique we use a benchmark of different computation topologies variating the amount of tasks, the dependance between tasks, and the number of processors used. Our results show different performances for the three evaluated techniques. In cases managing smaller computations with fewer processors, both FIFO and LIFO perform better. When we increase the size of the computations and the number of processors used, our proposed priority-based technique performs better. With respect to the fairness of the algorithms, they are all unbalanced and no significant conclusion can be reached.

Keywords: Parallel computing · Work stealing · Dynamic scheduling

1 Introduction

Modern software systems normally incorporate parallel, cloud, machine-learning, or big-data technologies in their development. The development of such systems has risen with the emergence of modern multi-core processors and the advancement in multi-processing algorithms that facilitate development and speedup their computation.

One of the algorithmic techniques to speedup concurrent computations is that of dynamic scheduling [6]. In such algorithms, tasks are dynamically assigned during the execution to each of the available processors. Work stealing algorithms [4,11] are among the most popular approaches for dynamic scheduling. In work stealing, tasks are not strictly assigned to a unique processor as they are spawned, instead, tasks are dynamically allocated by considering the number of available (*i.e.,* idle) processors. The specific algorithm used to choose (*i.e.,* steal) a task from processors' available task queue may vary (most commonly using First-in First-out (FIFO) or Last-in First-out (LIFO) policies), which dictates the scheduler's inner work and performance.

E. Gonzalez et al. (Eds.): CCC 2021, CCIS 1594, pp. 133–150, 2022.
https://doi.org/10.1007/978-3-031-19951-6_9

In this paper we review the classic work stealing algorithms as a means to understand them in depth, with the objective of improving their execution time. To improve the execution time, we propose a new work stealing algorithm based on a priority-based policy, assigning priorities to each task in the computation graph, with respect to their topological order. The idea behind our proposal is that the tasks to execute by available processors are exactly those that come first in the topological order, as many other tasks depend on them to complete their execution.

We focus the evaluation of our proposed priority-based algorithm with respect to the classical algorithms, FIFO and LIFO, from two perspectives, performance and fairness. The performance evaluation focuses on the execution time of different simulated multithreaded computations. Using the same simulation, we evaluate fairness measuring the amount of tasks processed by each processor. Our results show a slight performance improvement for our proposed algorithm, while the behavior of the three algorithms is very similar, and unbalanced, when evaluating fairness.

We start the paper in Sect. 2 providing an outlook and context for work stealing algorithms and discuss the two classical algorithms based on the FIFO and LIFO policies. In Sect. 3 we introduce our proposed priority-based work stealing algorithm. The evaluation of our algorithm, in Sect. 4, presents the execution of our benchmark simulating the execution of different configuration variants for each of the three algorithms. Finally, Sect. 5, presents the conclusion of our work, and offers avenues of future work.

2 Background

This section describes the state-of-the-art and background in work stealing algorithms. We begin by presenting schedulers in multithreaded computations. In second place, we present the model for multithreaded computations used in work stealing schedulers. Finally, we conclude this section presenting the most classic work stealing algorithms.

2.1 Giving Context to Work Stealing

There are many techniques for scheduling aperiodic tasks in multithreading computations [6]. The majority of the work in this area deals with *static scheduling* algorithms, *i.e.,* algorithms that, beforehand, compute which tasks are going to be executed in their corresponding order for each of the processors involved in the computation. One of the major advantages of static scheduling algorithms is their predictability, since, for a given static schedule, it is straightforward to derive information on the application's execution time. However, static scheduling algorithms require precise information on the execution times of the tasks to be scheduled, which are hard to obtain for modern multiprocessor. Moreover, static schedulers heavily depend on the architecture of the machine where the application is going to be executed, which severely affects portability [9].

Unlike static scheduling, *dynamic scheduling* is performed on-the-fly. With the emergence of multiprocessors, dynamic scheduling has gained interest given the fact that many applications create dynamically changing sets of tasks that need to be scheduled among the processors. The main advantages of dynamic scheduling algorithms for multiprocessor systems are the automatic load balancing and improved portability. The downside of using dynamic scheduling is the limited predictability on the execution performance for the overall computation.

Work stealing stands as one of the most widely used approaches for dynamic scheduling. The idea behind work stealing is that an idle processor "steals" work from other randomly chosen processor as a means to accelerate computations. A processor is said to be *idle* if the queue that stores ready tasks (*i.e.,* task ready to execute) is empty. If all processors are busy (their ready task is not empty), then there is no need to migrate tasks between them. Among the benefits of work stealing the following are identified [10]:

- Scalability with respect to the number of processors.
- Idle-initiative task migration minimizes the scheduling overhead.
- Communication overhead is kept low by taking into account data locality.

Work stealing has been employed in many frameworks for parallel programming [3], and has found plenty of applications in simple divide-and-conquer algorithms [5] and complex stream processing applications [2].

2.2 A Model for Multithreaded Computations

Multithreaded computation can be described as a Directed Acyclic Graph (DAG) in which every node represents a unit-size task and each edge models the dependency among these tasks [4]. In this model a task cannot be executed if its parent tasks have not been executed. Moreover, multithreaded computation's DAGs have one root and one sink, representing the first and last tasks in the computation, respectively. Therefore, an execution schedule for a multithreaded computation must obey the constraints imposed by the topology of the DAG representing it.

We quantify and bound the execution time of a multithreaded computation as in Eq. (1). For a given computation, let $T(\mathcal{S})$ denote its execution time, and let T_n be the execution time for a computation with n-processors and schedule $\mathcal{S}$.

$$T_n = \min_{\mathcal{S}} T(\mathcal{S}). \tag{1}$$

T_1 is the time that it takes one processor to execute all tasks in the computation, and T_∞ is the time to execute the computation using an arbitrarily large number of processors. Note that these quantities are proportional to the number of vertices of the DAG modeling the execution and its longest path respectively. Based on the execution time, a scheduler is said to satisfy the *greedy property* if at each execution step in which at least n tasks are ready, then n tasks execute. If fewer than n tasks are ready, then all execute [4]. We then have the following result bounding greedy schedulers' execution time.

Theorem 1. *(Greedy-Scheduling Theorem). For any multithreaded computation and any greedy schedule* $\mathcal{S}$*,*

$$T(\mathcal{S}) \leq \frac{T_1}{n} + T_\infty.$$

As mentioned in Sect. 2.1, in a work stealing scheduler, each processor has a *ready dequeue*, which is a double-ended queue that stores tasks to be executed. Processors successively dequeue tasks from its ready dequeue, executes them, and continues with the next task in their ready dequeue. A task may *spawn* new tasks, represented as the children of a given vertex in the DAG modeling the computation. Spawned tasks are enqueued in the ready dequeue of the processor that executed their parent. Note, however, that this does not imply that all parents of the spawned tasks have been executed. If a processor attempts to execute a child task without completing the execution of its parents, then the processor yields a *stalled* state. For example, in Fig. 1, after executing Task 1, the left-most processor spawned four tasks, enqueued in its ready dequeue. If the ready dequeue of a processor is empty (*i.e.,* the processor is idle), it begins work stealing. When work stealing, a processor steals a task from other processor's ready dequeue. In this example, the idle processors steal tasks 2, 3, and 4 from the left-most processor's ready dequeue (Queue A).

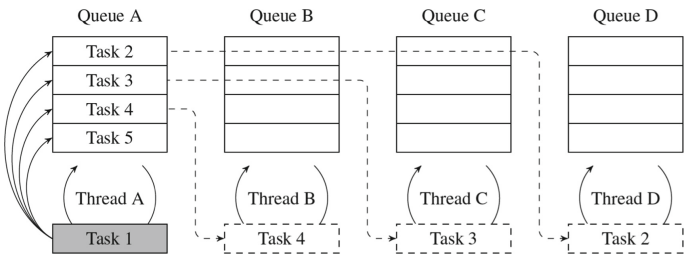

Fig. 1. Work stealing scheduler on a system with four processors [9]

2.3 Work Stealing Policies and Algorithms

In the literature, there are two main algorithms for scheduling multithreaded computations using work stealing which also satisfy the greedy property [4,9]. The main difference between these algorithms is that stealing, local enqueueing and dequeueing is made following either the LIFO or FIFO policies, which we explain now.

In the *LIFO work stealing algorithm* each of the processors executing the computation follows three rules to change their state:

1. ***Spawns.*** If a processor executes a task that spawns a set of tasks A, each task in A is placed at the bottom of the ready dequeue of the processor. In the next step the processor begins to work on the bottom task.
2. ***Stalls.*** If a processor stalls, then it checks its ready dequeue and starts working on the bottom task. If the ready dequeue is empty, the processor begins work stealing.
3. ***Stealing.*** When work stealing, a processor steals the top task from the ready dequeue of a randomly chosen processor and begins working on it. If the victim's ready dequeue is empty, the processor chooses another processor at random.

The main motivation for processors accessing their ready dequeues in LIFO order is that most tasks share data with the task that spawned them. Hence, when a newly created task is executed, it is very likely that the required data is still in the cache of the processor [1]. However, this approach is not *fair* in the sense that a task in the top of the ready dequeue of a processor might never be executed if all worker threads are busy. Thus, there is not guarantee that a run is executed continuously, and there will be no upper bound on the timespan between creation and execution of a task.

In the *FIFO work stealing algorithm*, the proposed solution addresses the fairness problem [9]. If the first enqueued task is the first to execute then there are no tasks perpetually waiting to be executed. The FIFO algorithm is similar to the LIFO algorithm. The main differences between the two algorithms lie in the way of enqueueing, dequeueing, and stealing tasks. The following explains the state change rules for the FIFO-based processors:

1. ***Spawns.*** Spawned tasks are enqueued at the top of the ready dequeue of the processor. As in the LIFO case, the processor always executes the bottom-most task in its ready dequeue.
2. ***Stalls.*** Works in the same way as for the LIFO algorithm.
3. ***Stealing.*** When work stealing, a processor always steals the bottom-most task from other processor's ready dequeue.

3 Priority-Based Work Stealing

The LIFO and FIFO algorithms described before are agnostic to the dependencies between tasks. Our proposal, is to use information on task dependency (*e.g.,* gathered using static analysis) to improve the performance of work stealing schedulers. Our proposal assigns a priority to each task, and steals tasks from other processors based on such priority.

In this section we present the idea behind our priority-based work stealing algorithm, the assignment of priorities based on a DAG structure modeling the computation, and describe the algorithm modeling multithreaded computations.

3.1 Priority-Based Work Stealing

In our analysis of the work stealing algorithms presented in Sect. 2.3, we notice that there is a strong relationship between stalled processors and task dependency. The more tasks depending on the currently enqueued/dequeued tasks, the more processors stall. As tasks that have many children may be potential bottle necks during the execution, they should be given priority when a processor core is dequeuing tasks from its ready dequeue or stealing from other processors.

To solve this problem, we propose a *Priority-based work stealing algorithm*. The motivating idea for our algorithm is to steal, enqueue, and dequeue tasks according to a *priority* assigned to each task in the computation. The priority of tasks depends on how critical is the task for the computation, where the criticality of a task is defined by the amount of subtasks spawned by a task, since these are the tasks more likely to generate bottle necks during the execution. We measure task criticality as the longest path of a vertex describing the task to the sink of the DAG modeling the computation. This metric is reasonable as the longer the path between the vertex and the sink, the more tasks will depend on the task, rising the probability of the task becoming a bottle neck.

A processor running the priority-based work stealing algorithm follows the next rules to change its state:

1. ***Spawns.*** If a processor executed a task that spawns a set of tasks, A, then the tasks in A are inserted, by priority order, in the processor's ready dequeue. The ready dequeue of every processor is ordered in descending priority order. In the next step, the core begins to work on the bottom task (*i.e.,* the task with the highest priority).
2. ***Stalls.*** If a processor stalls, then it checks its ready dequeue and starts working on the bottom task. If the ready dequeue is empty, the processor begins work stealing.
3. ***Stealing.*** When work stealing, a core steals the bottom task from the ready dequeue of a randomly chosen processor and begins working on it. If the victim's ready dequeue is empty, the core tries again picking another core at random.

3.2 Modeling and Implementing Multithreaded Computations

To model a multithreaded computations we use a DAG where each vertex represents a task in the computation, and the out edges of a vertex represent the spawned tasks. In turn, in-edges for a vertex, represent the tasks the vertex depends on. Multithreaded computations are modeled as follows.

Let $G = \langle V, E \rangle$ be a directed graph where V is the set of vertices and E the set of edges between elements in V. Vertices are labeled with the numbers $1, \ldots, N$ and let $A(G)$ be the $V \times V$ adjacency matrix of G such that

$$A(G)_{ij} = \begin{cases} 1 & \text{if there is a directed edge from vertex } i \text{ to vertex } j, \\ 0 & \text{otherwise.} \end{cases}$$

for all $1 \leq i, j \leq N$. Recall that a graph G is a DAG if and only if G admits a topological order. Thus, one can label the vertices of a DAG in such a way that its adjacency matrix is strictly lower triangular.

Conversely, if the adjacency matrix of a graph G is strictly lower triangular, then G is a DAG. We will restrict the application of our algorithm to multithreaded computations that generate DAGs. Note from our definition, that an adjacency matrix with more than one zero column implies that the DAG has more than one root. Similarly, having more than one zero row implies that the DAG has more than one sink. Neither case is applicable to multithreaded computations.

3.3 Priorities Algorithm

As discussed in Sect. 3.1, we define a work stealing strategy based on a priority function given by the longest path from a vertex to the sink vertex of the DAG modeling the computation. Given two vertices i, j in the DAG, the vertex i will have a higher priority than j if the longest path $lp(i)$ from i is greater than $lp(j)$ from j. Algorithm 1.1 shows how the longest path is computed for all vertices in our computation's DAG [8].

```
void lp(int i) { //compute the longest path for vertex i
  G := ⟨V,E⟩
  int[] priority := new PQ(N) //priority queue of size N=|V|

  if(!priority[i]) {
    //d:V → ℕ longest path for vertex i
    int d(int i) {
      if(i != N)
        return  max_{(i,j)∈E}{d(j)+1}
      else
        return 0
    }
    for(int i=1; i<=N; i++)
      priority[i] := d(i)
  }
  //priority is filled with the longest paths from each vertex
       of G to the out vertex of the computation
  return priority[i]
}
```

Algorithm 1.1. Priority computation of all the vertices in a DAG modeling a multithreaded computation.

Note from Algorithm 1.1 that it is sufficient to calculate $d(1)$. Vertex 1 is connected to every vertex i, for $1 < i \leq N$, therefore $d(i)$ is calculated recursively for every task spawned once we calculate $d(1)$.

3.4 Priority Work Stealing Scheduler

Given a DAG for a multithreaded computation (*e.g.,* generated from a static analysis tool to extract the structure from the computation), the first step for our work stealing algorithm is to calculate the priorities for all vertices as in Algorithm 1.1. This is done by a multithreaded computation manager component. Then, a work stealing manager component creates an object to manage all the components to manage the computation. That is, managing the ready dequeue for each processor.

The execution starts by choosing the first task (the root node in the DAG) to execute in a given processor (*e.g.,* processor p_0). As processors visit tasks (vertices in the DAG), the multithreaded computation manager marks them as visited and proceeds to search task's children. If the children tasks are not yet enqueued, they are added to the processor's ready dequeue, following the rules of the work stealing algorithm. Then, the processor checks its ready dequeue and executes the next task, the bottom task (according to the rules of the work stealing algorithm). Recall that if all the parents of the task that the processor is attempting to execute have not been executed yet, the processor stalls meaning that it looks in its ready dequeue for another task to execute. If the core fails to find a task ready to execute, it begins work stealing.

Remember a processor begins work stealing if either its ready dequeue is empty or is stalled and fails to find a task ready to execute in its ready dequeue. In this case, the processor informs the multithreaded computation manager that is looking for a vertex to steal. The multithreaded computation manager adds the processor to a pool of processors looking for vertices to steal and puts it on hold. Meanwhile, the work stealing manager looks, among the procesos that are not stealing, for processors that have tasks available in their ready dequeues to steal from. A processor is chosen as victim if it has more than one task in its ready dequeue. In this case it gives to the multithreaded computation manager the bottom task in its ready dequeue. The multithreaded computation manager gives the stolen task to one of the processors in the pool, which, in the next step, attempts to execute the task. If all the processors attempt to steal at the same time, they generate a deadlock. To address this problem, if the number of processors work stealing at any given step equals the number of processors executing the computation, the pool is reset, causing each of the processors to execute the bottom task in their ready dequeue. Given that the execution time of tasks varies, not all processors will reach a stealing state at the same time, allowing for the computation to progress.

To implement the aforementioned process, each of the components described are represented by a Java class, as described in the following.[1]

1. The multithreaded computation manager component implements Algorithm 1.1 to calculate the priority of each vertex in the DAG.

[1] Available at our GitHub repository: https://github.com/FLAGlab/WorkStealingAlgorithms.

2. A component to update and manage the state of the multithreaded computation. This component knows the number of processors available, which vertices of the DAG have been visited, which vertices are enqueued in the ready dequeue of a processor, and how many of the parents for a vertex have not been executed yet.
3. The work stealing manager component is in charge of synchronizing work stealing among the processors. This is implemented using threads and synchronized procedures.
4. The controller component receives a DAG and the number of available processors, and orchestrates the complete process, managing instances of the other components above.

4 Validation

We evaluate our proposed priority-based work stealing algorithm using a benchmark to compare with the classic FIFO and LIFO based algorithms. We evaluate the performance and processors' load in different computation graph sizes and densities.

4.1 Experimental Design

To measure performance, we define a benchmark with different evaluation scenarios. In each scenario we take into account the average maximum execution time obtained from all processors across five runs, as a means to reduce warm-up or processor clock bias in our results. To measure processors' load we count the number of tasks executed by each processor with respect to the total number of tasks in the computation.

Each validation scenario uses a different computation size (*i.e.,* number of tasks), generating DAGs to represent the computation containing 50, 100, 200, 400, 800, and 1600 nodes in each scenario. Additionally, for each DAG size we evaluate three different graph density values (*i.e.,* the ratio of outgoing edges to nodes) 0.2, 0.5, and 0.8, to observe the impact of the algorithms in different computation settings. Additionally, we vary the number of processors executing the multithreaded computation between 1 (used as the linear baseline) and 96, scaling in powers of 2. For every possible configuration we use the same generated DAG for all runs for each of the three work stealing algorithms, LIFO, FIFO, and priorities.

DAG Generation. All DAGs representing computations used in our evaluation are generated using the following process. The DAG's density takes into account the ratio between the number of edges and the number of vertices in the graph. To manage this, we follow Erdös-Rényi's model for graph construction [7]. In particular, edges are included in the graph with independent probability $0 < p \leq 1$. Algorithm 1.2 describes the DAG generation process. Lines 3–9 follow Erdös-Rényi's algorithm to generate the DAG's adjacency matrix `M`, for a given

random probability p. Lines 10–17 verify that no row in the matrix is zero, as we only allow for a single root and sink nodes in the computation graphs (as stated in Sect. 3.2). In case a zero row is generated, the values are changed following the Erdös-Rényi algorithm.

```
1  let N > 0 && density in 0 < p ≤ 1.
2  int[][] M := new Array(new Array (N))
3  for(int j=0; j<N; j++) {
4     for(int i=0; i<j; i++) {
5        if(probability(p))
6           M[i][j] := 1
7        else if(probability(1-p))
8           M[i][j] := 0
9     }
10    if(isZero(row(j))) {
11       for(int i=0; i<j; i++) {
12          if(probability(p))
13             M[i][j] := 1
14          else if(probability(1-p))
15             M[i][j] := 0
16       }
17    }
18 }
```

Algorithm 1.2. Generate the adjacency matrix for a DAG, modeling a multithreaded computation, with N vertices and density p.

Evaluation Configuration. All our benchmarks ran on a XeonSP G291-281 GPU Server with two RTX2080 CPUs, each with 48 physical cores with a 2.2 GHz frequency and a 128 GB NUMA enabled memory architecture, running the Ubuntu 20.04.2 LTS OS. We use version 1.8 of the JVM for our experiments.

4.2 Results

To evaluate the performance, we take into account the execution time, given in milliseconds, for each of the algorithms, across all DAG sizes for each of the densities. Figures 2 through 13 show the behavior of the three algorithms scaling up the number of processors used.

Our second experiment evaluates the load of each processor when executing a multithreaded computation with a fixed number of tasks and a fixed density for the dependencies between tasks. As mentioned in Sect. 2.3 the original FIFO and LIFO algorithms may be unfair [9] with respect to the way in which processors chose the tasks to execute or steal from other processors. Therefore, the purpose of this experiment is to assess the fairness of our proposed algorithm with respect to the FIFO and LIFO algorithms. Figures 14 through 19, show the number of tasks executed by each processor for a given computation. Here we show the computation of 200 tasks running with 8 and 32 processors respectively, variating

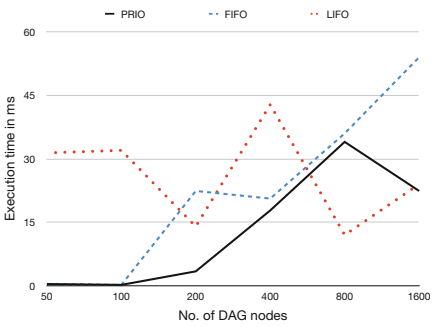

Fig. 2. Algorithms comparison for density 0.2 using 1 processor

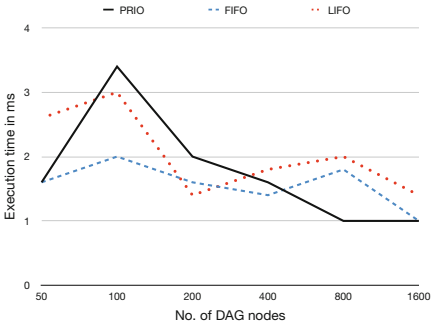

Fig. 3. Algorithms comparison for density 0.2 using 8 processors

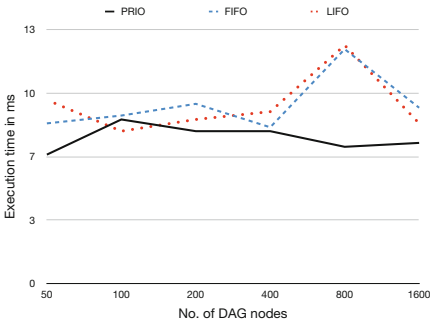

Fig. 4. Algorithms comparison for density 0.2 using 32 processors

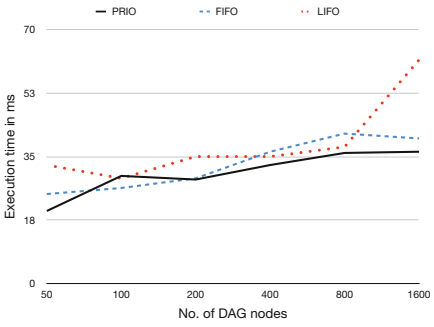

Fig. 5. Algorithms comparison for density 0.2 using 96 processors

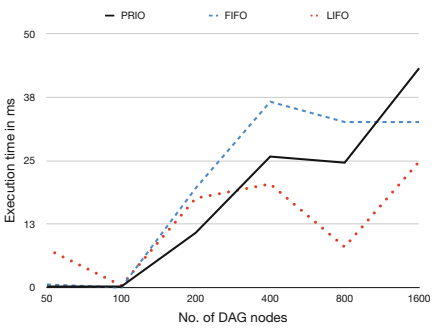

Fig. 6. Algorithms comparison for density 0.5 using 1 processor

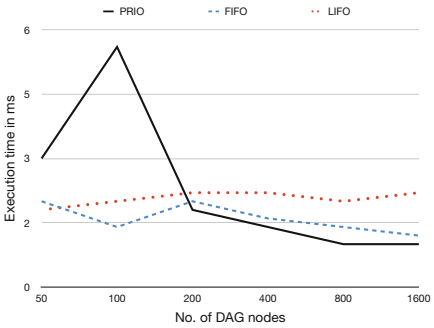

Fig. 7. Algorithms comparison for density 0.5 using 8 processors

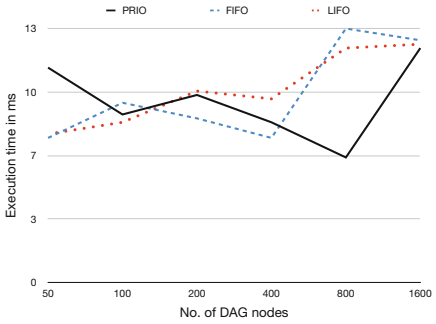

Fig. 8. Algorithms comparison for density 0.5 using 32 processors

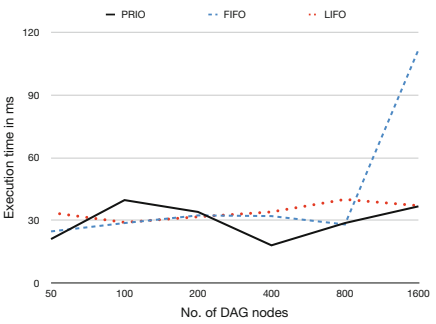

Fig. 9. Algorithms comparison for density 0.5 using 96 processors

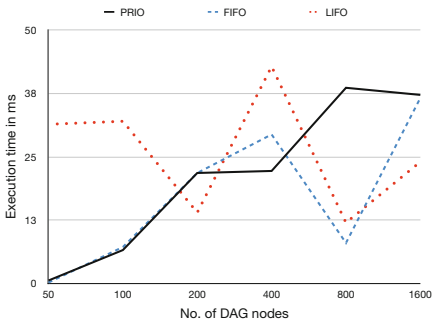

Fig. 10. Algorithms comparison for density 0.8 using 1 processor

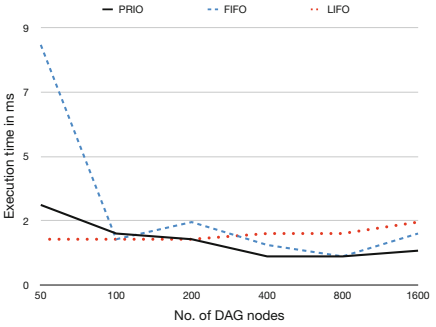

Fig. 11. Algorithms comparison for density 0.8 using 8 processors

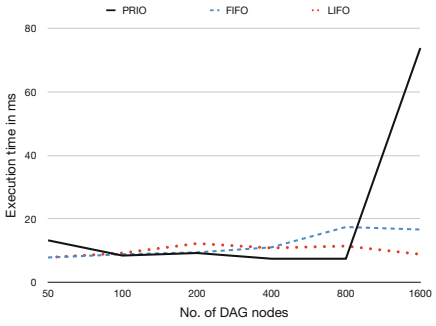

Fig. 12. Algorithms comparison for density 0.8 using 32 processors

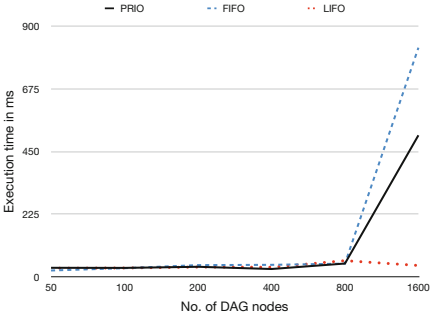

Fig. 13. Algorithms comparison for density 0.8 using 96 processors

the density of dependencies between tasks. Each column shows the number of tasks executed by each of the processors. The behavior of other variations of computation sizes and numbers of processors present a similar behavior, which we present in an online appendix together with all other data and algorithms from our work.[2]

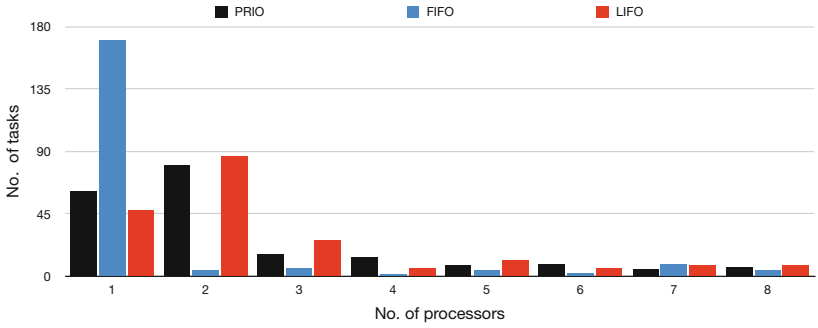

Fig. 14. Load of tasks for 8 processors with 200 task's computations with density of 0.2

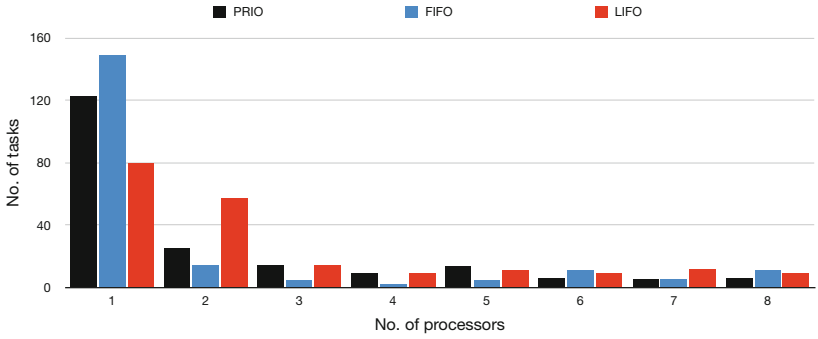

Fig. 15. Load of tasks for 8 processors with 200 task's computations with density of 0.5

4.3 Analysis of the Results

From the performance results we can observe that the behavior of the algorithms is erratic, with individual cases favoring one algorithm over the others. Across all density configurations using fewer processors (*i.e.,* ≤ 8) the best performing algorithm in most cases is FIFO. However, as we increase the number of tasks in the computation (*i.e.,*DAG nodes), and the number of processors, the performance of our proposed algorithm rapidly improves across all configurations. We can

[2] https://flaglab.github.io/WorkStealingAlgorithms/.

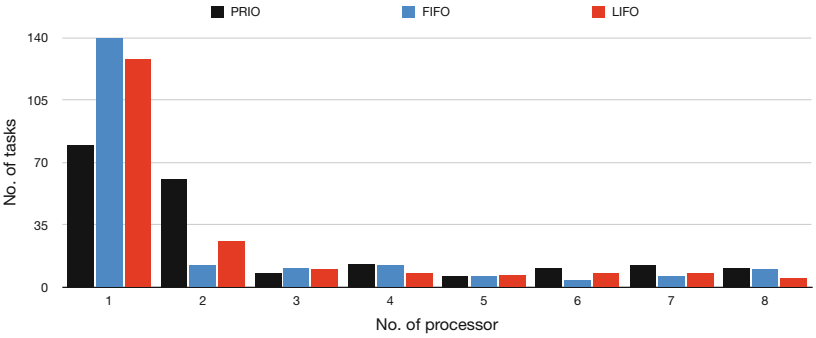

Fig. 16. Load of tasks for 8 processors with 200 task's computations with density of 0.8

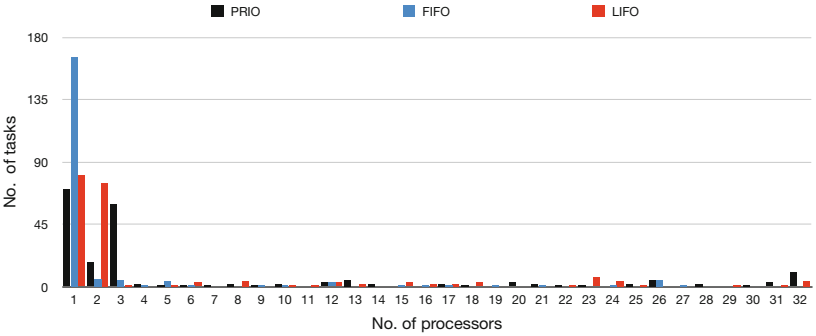

Fig. 17. Load of tasks for 32 processors with 200 task's computations with density of 0.2

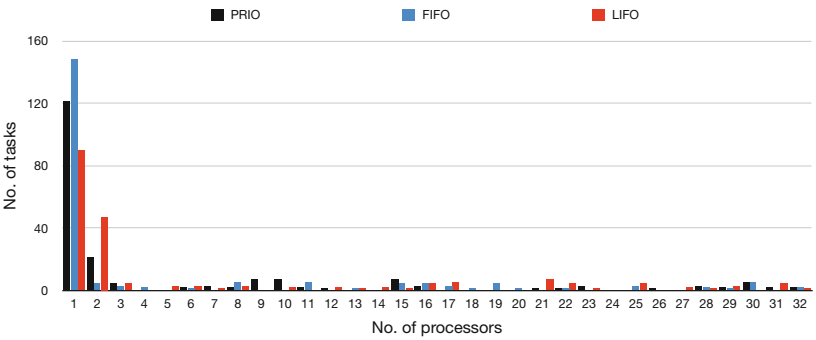

Fig. 18. Load of tasks for 32 processors with 200 task's computations with density of 0.5

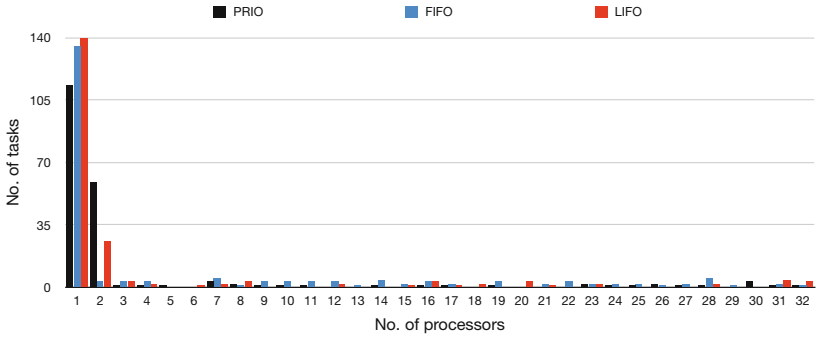

Fig. 19. Load of tasks for 32 processors with 200 task's computations with density of 0.8

observe the FIFO and priority algorithms present a rapid performance decrease around 800 DAG nodes in highly dense graphs with many processors (the last two configurations), while the LIFO algorithm is in average more stable. We also observe that the proposed priority-based algorithm presents a better performance in average than its FIFO and LIFO counterparts in the sparse density scenarios. However, as the density of the graphs and the number of processors used increase, the performance of the three algorithms becomes very similar.

We can observe that the benefit of our proposed priority-based algorithm comes for larger computations as the number of processors used increases (not taking into account outlier performance results for any of the algorithms. Therefore, we argue that for heavy computations including a high level of parallelism, our priority-based algorithm is best suited. When we deal with smaller-size computations, using the base FIFO algorithm can be best suited. However, our evaluation is not conclusive with respect to specific situations in which one of the two classic algorithms triumph the other. The difference in their execution may be due to the specific structure of the underlying DAG for the computation.

In many cases we observe a performance decay as we use 96 processors for the evaluation. This behavior is due to the fact that, as mentioned in Sect. 3.4, our evaluation program for multithreaded computations running with n processors uses $n+2$ threads, where the additional threads are used for managing the overall computation; one controlling the processors, and one controlling the stealing controller. Given the configuration of the machine used for our evaluation has a maximum of 96 physical threads, we overload the machine's capacity, which may cause the observed performance decrement.

With respect to the fairness evaluation (Figs. 14, 15, 16, 17, 18 and 19) we observe that most of the tasks are managed by the first couple of processors across all cases. However, we note that as the computation graphs are denser, the load among the processors is more unbalanced, specially when more processors are available. In particular, most of the tasks execute on processor 1. One possible reason for this is that, for instance, when setting a 0.8 density, after executing the first task, processor 1 enqueues in its ready dequeue 80% of the vertices of the

DAG modeling the multithreaded computation. Thus on the whole computation the remaining processors will be stealing tasks from processor 1. Moreover, after a processor steals a task from processor 1, it is highly probable that it will need to steal again since the children of the stolen task are likely to already be in processor's 1 ready dequeue.

4.4 Threats to Validity

We identify different situations that may add noise and bias the internal validity of our evaluation. These are related to factors that could affect the variables and the relations being investigated.

The generation of the computations DAGs, following Algorithm 1.2, used to simulated our evaluation scenarios may diverge from DAGs modeling a real multithreaded computation. As a matter of fact, to keep the evaluation scenarios simple, we omit additional conditions imposed in the topology of a DAG modeling a multithreaded computation [4] that could have and impact in the overall performance of the algorithms scheduling a real multithreaded computations.

5 Conclusion and Future Work

Multithreaded computations are becoming the norm in modern software systems. Therefore, being able to exploit the best possible performance from the underlying parallel infrastructure, *e.g.,* schedulers, is key for the success of many software systems.

The main goal of our work is to present a review and evaluate existing and new algorithms to schedule multithreaded computations by means of work stealing. As a result, we proposed a new work stealing algorithm based on a priority assigned to each vertex in the DAG modeling a multithreaded computation. The definition of the priority is calculated algorithmically taking into account the importance of tasks to complete the computation. Tasks with required to finish in order to complete other tasks receive a higher priority and therefore should be scheduled more promptly. In order to evaluate and compare the performance of the work stealing algorithms we use a benchmark to simulate and execute multithreaded computations by means of the generation of a DAG representing the computation. We measure the total execution time and the work load of each processor involved in the multithreaded computation for different configurations variating the size of the computation, the density of interactions between tasks in the computation, and the number of processors used to execute the computation.

Our results show that the proposed algorithm is effective in executing multithreaded computations based on it performance in comparison to the execution of the FIFO and LIFO-based algorithms. Our algorithm shows most useful for larger computations using more processors, while both FIFO and LIFO perform better for smaller-size computations using fewer processors. Our evaluation also shows that the three algorithms are unbalanced, executing most tasks on the

first couple of processors and leaving all other processors to manage only a hand full of tasks. In some cases our priority-based presents a better balance than the FIFO and LIFO algorithms, nonetheless, the difference is not significant.

There is no perfect or definitive work stealing algorithm that is best suited for any multithreaded computation. As the results, shown in Sect. 4.2, indicate there are run configurations in which the FIFO presents a better performance, the same way LIFO and priority based do for other configurations. This suggests that there are conditions in the topology of the DAG and the number of processors used to run the multithreaded computation that may favor one algorithm over another. These conditions should be explored as future work.

Our Priority-based work stealing algorithm is not optimally efficient yet. There are improvements that could be made in the priority function in which this algorithm is based to get a better performance. For instance, instead of calculating the priorities of the vertices with the longest path, we can think of calculating them with the number of vertices in their spanning tree. Alternatively, if the DAG we are considering is weighted, the edge's weights are yet another variable to be considered when defining the priority function. A more in-depth evaluation to explore this is needed.

Finally, we could improve or explore alternatives to our priority-based work stealing algorithm by using a graph-theoretical approach to improve and refine the priority function, in which this algorithm is based.

References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures, pp. 1–12. SPAA 2000, ACM, New York (2000). https://doi.org/10.1145/341800.341801
2. Anselmi, J., Gaujal, B.: Performance evaluation of work stealing for streaming applications. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) Principles of Distributed Systems, pp. 18–32. Springer (2009)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 207–216. PPOPP 1595, ACM, New York (1995). https://doi.org/10.1145/209936.209958
4. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999). https://doi.org/10.1145/324133.324234
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
6. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 1–44 (2011). https://doi.org/10.1145/1978802.1978814
7. Erdös, P., Rényi, A.: On random graphs I. Publicationes Math. **6**, 290–297 (1959)
8. Khan, M.: Lecture notes for the course CSE-221 Graduate Operating Systems (2011)

9. Mattheis, S., Schuele, T., Raabe, A., Henties, T., Gleim, U.: Work stealing strategies for parallel stream processing in soft real-time systems. In: Herkersdorf, A., Römer, K., Brinkschulte, U. (eds.) ARCS 2012. LNCS, vol. 7179, pp. 172–183. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28293-5_15
10. Neill, D., Wierman, A.: On the benefits of work stealing in shared-memory multiprocessors. Carnegie Mellon University, Tech. rep. (2010)
11. Yang, J., He, Q.: Scheduling parallel computations by work stealing: a survey. Int. J. Parallel Program. **46**(2), 173–197 (2017). https://doi.org/10.1007/s10766-016-0484-8