Optimizing Work Stealing Algorithms

Juan Sebastián Numpaque Roa

A thesis submitted in partial fulfillment for the degree of
Systems and Computing Engineer

Advisor:
Nicolás Cardozo Álvarez Ph.D.

Universidad de los Andes
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas y Computación
Bogotá, Colombia
2020

Tu bien sabes, Merceditas, desde donde quiera que estés, que a ti consagro toda labor y trabajo que hago.

Acknowledgements

I would like to thank my parents for all their support throughout this journey, for never stop believing in me and for its infinite love and comprehension. It is heart-warming to know that both of you, Alba and Pedro, are always supporting me.

Verónica, my partner in crime, my best friend. Without your support and love this work would not have been possible. I love you madly.

Julián, the one and only friend that I made in these years studying engineering. Working with you have been a pleasure, you are an incredible person and engineer. If I had never met you I probably would not be writing these words.

Last but not least, I would like to thank to my advisor, Nicolás, for proposing to me the interesting topic exposed in this work. Also, for its insightful comments and suggestions along the process of completing this work.

Contents

Introduction **6**

1 What is Work stealing? **7**
1.1 Giving context to Work stealing 7
1.2 A model for multithreaded computations 8
1.3 Work stealing algorithms 9

2 Priority-based Work stealing & Modelling of the problem **11**
2.1 Priority-based Work stealing 11
2.2 Modeling and implementation 12
2.2.1 DAG generation 12
2.2.2 Priorities algorithm 13
2.2.3 Work stealing scheduler program 14

3 Tests & Results **17**
3.1 Design of experiments 17
3.2 Results & Graphs 17
3.3 Analysis of the results 24
3.4 Threats to validity 25

4 Conclusions & Future work **26**
4.1 Conclusions 26
4.2 Future work 27

List of Figures

1.1 Work stealing scheduler on a system with four processor cores. Taken from: [MSR+12] . . . 9

3.1 50 vertices - 0.2 density. Time vs. Number of processors graph . . . 18
3.2 50 vertices - 0.5 density. Time vs. Number of processors graph . . . 18
3.3 50 vertices - 0.8 density. Time vs. Number of processors graph . . . 18
3.4 100 vertices - 0.2 density. Time vs. Number of processors graph . . . 18
3.5 100 vertices - 0.5 density. Time vs. Number of processors graph . . . 18
3.6 100 vertices - 0.8 density. Time vs. Number of processors graph . . . 18
3.7 200 vertices - 0.2 density. Time vs. Number of processors graph . . . 19
3.8 200 vertices - 0.5 density. Time vs. Number of processors graph . . . 19
3.9 200 vertices - 0.8 density. Time vs. Number of processors graph . . . 19
3.10 400 vertices - 0.2 density. Time vs. Number of processors graph . . . 19
3.11 400 vertices - 0.5 density. Time vs. Number of processors graph . . . 19
3.12 400 vertices - 0.8 density. Time vs. Number of processors graph . . . 19
3.13 800 vertices - 0.2 density. Time vs. Number of processors graph . . . 20
3.14 800 vertices - 0.5 density. Time vs. Number of processors graph . . . 20
3.15 800 vertices - 0.8 density. Time vs. Number of processors graph . . . 20
3.16 1600 vertices - 0.2 density. Time vs. Number of processors graph . . . 20
3.17 1600 vertices - 0.5 density. Time vs. Number of processors graph . . . 20
3.18 1600 vertices - 0.8 density. Time vs. Number of processors graph . . . 20
3.19 50 vertices - 0.2 density. Load per processor core graph . . . 21
3.20 50 vertices - 0.5 density. Load per processor core graph . . . 21
3.21 50 vertices - 0.8 density. Load per processor core graph . . . 21
3.22 100 vertices - 0.2 density. Load per processor core graph . . . 21
3.23 100 vertices - 0.5 density. Load per processor core graph . . . 21
3.24 100 vertices - 0.8 density. Load per processor core graph . . . 21
3.25 200 vertices - 0.2 density. Load per processor core graph . . . 22
3.26 200 vertices - 0.5 density. Load per processor core graph . . . 22
3.27 200 vertices - 0.8 density. Load per processor core graph . . . 22
3.28 400 vertices - 0.2 density. Load per processor core graph . . . 22
3.29 400 vertices - 0.5 density. Load per processor core graph . . . 22
3.30 400 vertices - 0.8 density. Load per processor core graph . . . 22
3.31 800 vertices - 0.2 density. Load per processor core graph . . . 23
3.32 800 vertices - 0.5 density. Load per processor core graph . . . 23

3.33 800 vertices - 0.8 density. Load per processor core graph 23
3.34 1600 vertices - 0.2 density. Load per processor core graph 23
3.35 1600 vertices - 0.5 density. Load per processor core graph 23
3.36 1600 vertices - 0.8 density. Load per processor core graph 23

Introduction

With the emergence of modern multi-core processors dynamic scheduling algorithms have gained interest. In these algorithms tasks are dynamically assigned, during the execution, to each one of the processor cores involved in it. Work stealing algorithms, which are this work's subject of study, are among the most popular approaches for dynamic scheduling. The main goal of this work is then to understand, review and look for improvement points in the classical Work stealing algorithms. As a result of this study, a new Work stealing algorithm, based on a priority assigned to each task in the computation, is proposed.

In the first chapter, we give context to Work stealing algorithms and discuse the two classical Work stealing algorithms which are the FIFO and LIFO based ones. In the second chapter we introduce the priority-based Work stealing algorithm that we are proposing in this work. Also, we outline the execution of a program that allowed us to simulate, evaluate and ran experiments on the Work stealing algorithms studied in this thesis. In the third chapter we explain the tests made and present the results obtained. Finally, in the fourth chapter, we conclude and open the window for future works.

Chapter 1

What is Work stealing?

The purpose of this chapter is to give meaning and to understand the "world" in which Work stealing lives. To do so we will first talk about schedulers in multithreaded computations. In second place, we will give ourselves a model for multithreaded computations to work with and we will understand how a Work stealing scheduler operates. Finally, we conclude this chapter studying the most classic Work stealing algorithms.

1.1 Giving context to Work stealing

Many techniques for scheduling aperiodic tasks in multithreading computations have been developed [DB11]. Plenty of the work in this area deals with *static scheduling* algorithms, i.e., algorithms which compute beforehand what tasks are going to be executed in each of the cores of the processor involved in the computation and in what moments. One of the major advantages of these algorithms is their predictability since, for a given static schedule, it is straightforward to derive information on the application's timing. However, static scheduling algorithms require precise information on the execution times of the tasks to be scheduled, which are hard to obtain for modern multicore processors. Moreover, static schedulers heavily depend on the architecture of the machine where the application is going to be executed, which severily affects portability [MSR+12].

Unlike static scheduling, *dynamic scheduling* is performed on-the-fly. With the emergence of multicore processors, dynamic scheduling has gained interest given the fact that many applications create dynamically changing sets of tasks that need to be scheduled among the processor cores. The main advantages of dynamic scheduling algorithms for multicore systems are automatic load balancing and improved portability. The downside, on the other hand, is a limited predictability since there is usually little information about the timing.

Work stealing stands as one of the most widely approaches for dynamic scheduling. The idea behind work stealing is that an idle processor core "steals" work from

other randomly chosen core. A core is said to be *idle* if the queue that stores ready tasks is empty, and thus there is no need to migrate tasks between the cores if all of them are busy. Among the benefits of work stealing are [NW10]:

- Scalability with respect to the number of processor cores.
- 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 [BJK+95] and has found plenty of applications from simple divide-and-conquer algorithms [CLRS09] to more complex stream processing applications [AG09].

1.2 A model for multithreaded computations

We can think of a multithreaded computation as a connected Directed Acyclic Graph (DAG) in which every node represents a unit-size task and each edge models the dependency among these tasks [BL99], i.e., a task cannot be executed if its parent tasks have not been executed yet. Moreover, multithreaded computation DAGs strictly have only one root and one sink representing the first and last tasks in the computation. Therefore, a execution schedule for a multithreadeding computation must obey the constraints imposed by the topology of the DAG that is modeling it.

We can quantify and bound the execution time of a multithreaded computation on a multicore processor. For a given computation, let $T(\mathcal{S})$ denote the time to execute the computation using an n-core processor execution schedule $\mathcal{S}$ and let

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

Thus, T_1 would be the amount of time that takes a processor with one core to execute the whole computation and T_∞ the time that would take a processor with an arbitrarily large number of cores to execute it. Note that these quantities are proportional to the number of vertices of the DAG modelling the execution and its longest path respectively. An n-core processor scheduler satisfies the *Greedy property* if at each step of the execution, in which at least n tasks are ready, then n tasks execute. And, if fewer than n tasks are ready, then all execute [BL99]. We then have the following bound for greedy schedulers:

Theorem 1.2.1. (*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 Section 1.1, in a work stealing scheduler, each processor core has a *ready dequeue*, which is a double-ended queue that stores tasks to be executed. Each

core successively dequeues task from its ready dequeue, executes it and continues with the next task in its ready dequeue. A task, of course, may *spawn* new tasks which represent the children of a given vertex in the DAG modelling the execution. These spawned tasks are enqueued in the ready dequeue of the processor that executed its parent. Notice, however, that this do not imply that all the parents of the spawned tasks have been executed. If a core attempts to execute one of these tasks, the core is said to be *stalled*. For example, in Figure 1.1, after executing Task 1, the left-most processor spawned four tasks, which are then enqueued in its ready dequeue. If the ready dequeue of a core happens to be empty, that is, if the core is idle, it begins work stealing. When work stealing, a core steals a task from other core's ready dequeue. In the example of Figure 1.1, the idle processor cores steal tasks 2,3 and 4 from the left-most core ready dequeue.

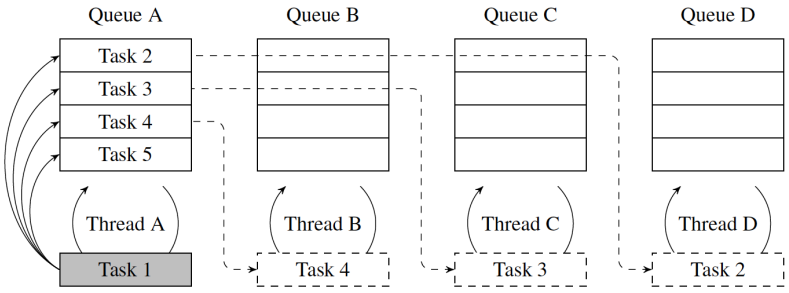

Figure 1.1: Work stealing scheduler on a system with four processor cores.
Taken from: [MSR+12]

Until now we have not discussed the policies that rule the stealing, and local enqueuing and dequeuing of tasks. This is discussed in the following section.

1.3 Work stealing algorithms

In the literature, there are two major algorithms for scheduling multithreaded computations using work stealing which also satisfy the greedy property [BL99, MSR+12]. The main difference between these algorithms is that stealing, local enqueueing and dequeueing is made following First In First Out (FIFO) and Last In First Out (LIFO) policies which we explain now.

In the *LIFO work stealing algorithm* each one of the cores of the processor executing the computation follows three rules:

1. ***Spawns.*** If a core executed a task that spawns a set of tasks A, then the elements of A are placed at the bottom of the ready dequeue of the core, and in the next step the core begins to work on the bottom-most task.

2. ***Stalls.*** If a core stalls, then it checks its ready dequeue and starts working on the bottom-most task. If the ready dequeue is empty, the processor begins work stealing.

3. ***Stealing.*** When work stealing, a core steals the top-most 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.

The main motivation for cores accesing their ready dequeues in LIFO order is that most tasks share data with their parent task, that is, 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 caches of the processor [ABB00]. However, this approach is not *fair* in the sense that a task in the top of the ready dequeue of a core 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 time span between creation and execution of a task.

A *FIFO work stealing algorithm* is proposed as a solution that addresses the fairness problem [MSR+12]. If the first enqueued task is the first one to be executed then there will not be tasks perpetually waiting to be executed. This algorithm works similarly as the previous one. As mentioned, there are differences which lie in the way of enqueueing, dequeueing and stealing tasks. The following rules explain this process:

1. ***Spawns.*** Spawned tasks are enqueued at the top of the ready dequeue of the core. As in the LIFO case, the core always executes the bottom-most task in its ready dequeue.

2. ***Stalls.*** Is done the same as for the LIFO algorithm.

3. ***Stealing.*** When work stealing, a core always steals the bottom-most task from other core's ready dequeue.

Chapter 2

Priority-based Work stealing & Modelling of the problem

None of the algorithms reviewed in the last chapter considers the dependence among the tasks. This is why we propose, in the first part of this chapter, a new Work stealing algorithm based on a priority assigned to each task in the computation.

In order to benchmark and compare the efficiency of the three Work stealing algorithms studied in this thesis, we first need to simulate a multithreaded computation. Therefore, in the second part of this chapter, we review the main components of a program that we wrote which models multithreaded computations and we outline, as well, its operation.

2.1 Priority-based Work stealing

Based on the review of the work stealing algorithms presented in Section 1.3, we notice that none of these take into account the amount of tasks depending on the ones that are being enqueued or dequeued. Tasks that have many children may be potential bottle necks during the execution and should be given priority when a processor core is dequeuing tasks from its ready dequeue.

This observation serves as motivation for the *Priority-based work stealing algorithm* proposed in this thesis. The basic idea is to steal, enqueue and dequeue tasks according to a *priority* assigned to each task in the computation. The priority of a task will depend, by definition, on how critical is the task for the computation in the sense of how likely is it to generate bottle necks during the execution. A good way to measure this is through the longest path of a vertex since the longer the longest path of a vertex, the more tasks will depend on it and the higher will be the probability that the task will be a bottle neck. Thus, we define the priority of a task or vertex as its longest path to the sink vertex of the DAG modeling the computation.

A processor core running the priority-based work stealing algorithm must follow the rules:

1. ***Spawns.*** If a core executed a task that spawns a set of tasks, A, then the elements of A are inserted by priority order in the ready dequeue of the processor core. The ready dequeue of every core is ordered in descending priority order. In the next step, the core begins to work on the bottom-most task, i.e., the task with the highest priority.

2. ***Stalls.*** If a core stalls, then it checks its ready dequeue and starts working on the bottom-most task. If the ready dequeue is empty, the processor begins work stealing.

3. ***Stealing.*** When work stealing, a core steals the bottom-most 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.

2.2 Modeling and implementation

We decided to use Java as the implementation language for a program that helps to benchmark and compare the performance of the three schedulling algorithms. The first problem to address is how to model a multithreaded computation. As we have already seen, DAGs serve us on this purpose. The second problem is the representacion of a core processor and threads are helpful with this. Thus, the program written is essentially a bunch of threads, which represent the cores of the processor executing the multithreaded computation, that are visiting synchronically and subject to the policies of the work stealing algorithms each one of the vertices in the DAG modeling the computation. More details on the implementation of the model will be explained now.

2.2.1 DAG generation

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 adjacency matrix of G. Remember that $A(G)$ is the $V \times V$ matrix whose entries are given by

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

for all $1 \leq i, j \leq N$. Also, 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. In the context of this project, we are interested in generating strictly lower triangular matrices in which the only zero rows and columns are, respectively, the first and the last one. Note that having more than one zero column implies that the DAG has more than one root and, similarly, having more than one zero row would imply that the DAG has more than one sink which is not the case we are interested in.

When generating DAGs, we also take into account its density, i.e., the ratio between the number of edges and the number of vertices. For this, we follow Erdős-Rényi model which, broadly speaking, tells us that a graph is constructed by connecting vertices randomly. In particular, each edge is included in the graph with probability $0 < p \leq 1$ independent from every other edge. A pseudocode for a DAG generator is now presented in Algorithm 1.

Algorithm 1: Generate the adjacency matrix for a DAG, modeling a multithreaded computation, with N vertices and density p.

1. Set a number of vertices $N > 0$ and a density $0 < p \leq 1$.
2. Create a $N \times N$ matrix, M.
3. For $0 \leq j \leq N$ and $0 \leq i < j$, assign 1 to M_{ij} with probability p and 0 with probability $1 - p$.
4. If one row or column happens to be entirely of zeroes, iterate over its entries and assign 1 with probability p and 0 with probability $1 - p$.

Result: The $N \times N$ matrix M which is the adjacency matrix of the DAG modeling a multithreaded computation.

2.2.2 Priorities algorithm

As discussed in Section 2.1, 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 execution. Therefore, given two vertices i, j in the DAG, the vertex i will have more priority than j if the longest path from i has greater length than the longest path from j. The pseudocode for the algorithm, inspired in lecture notes in [Kha11], that computes the longest path from each vertex in a DAG is shown in Algorithm 2.

Algorithm 2: Compute the longest path or priority of all the vertices in a DAG modeling a multithreaded computation.

1. Set a DAG, $G = \langle V, E \rangle$, that models a multithreaded computation.
2. Create an array, *priority*, of size $N = |V|$, that will store the longest path from each vertex. More precisely, $priority[i] :=$ Longest path from vertex $i \in V$.
3. Define $d : V \to \mathbb{N}$ as $d(i) :=$ Longest path from vertex i. Note that

$$d(i) = \begin{cases} \max_{(i,j)\in E}\{d(j) + 1\} & \text{if } i \neq N, \\ 0 & \text{if } i = N. \end{cases}$$

4. Fill out the array *priority* setting $priority[i] = d(i)$.

Result: The array *priority* filled with the corresponding longest paths from each one of the vertices of G.

Remark 2.2.1. It is sufficient, in Algorithm 2, to calculate $d(1)$. The vertex 1 is connected to every vertex $1 < i \leq N$ so $d(i)$ will be calculated recursively at some point in the execution of the algorithm.

2.2.3 Work stealing scheduler program

There are six Java classes in the program modeling our work stealing scheduler:

1. A DAG generator that implements the algorithm described in Subsection 2.2.1.
2. One that implements the algorithm described in Subsection 2.2.2, which calculates the priority of each vertex in the graph generated by the DAG generator.
3. The class modeling a processor core which is implemented using threads.
4. A class for the work stealing manager, which synchronizes work stealing among the processors. This is also implemented with the help of threads.
5. A class that updates and manages the state of the multithreaded computation. This class knows how many processors are executing the computation, which vertices of the DAG have been visited, are enqueued in the ready dequeue of a processor core and how many of its parents have not been executed yet.
6. A controller class that receives a DAG and a number of processors, creates the corresponding instances of the classes described above and starts the execution.

Now we outline the operation of the program. Users enter a number of vertices and a density for the DAG that is going to model the multithreaded computation, the number of processors that are going to visit the DAG or execute the multithreaded computation and chooses one of the three Work stealing algorithms that will rule the local enqueuing, dequeuing and stealing.

Then, the DAG generator creates a graph with the number of vertices and density given by the user. This DAG and the number of processors are passed to the controller which creates the processors, the stealing manager and the multithreaded computation manager. If the Work stealing algorithm chosen is the priority-based algorithm, the multithreaded computation manager calls the class that calculates the priority of each one of the vertices. Once all the objects involved in the computation have been created, the controller starts the processors and the work stealing manager. Each one of the threads representing the cores and the work stealing manager will be alive while there are unvisited vertices in the DAG modeling the multithreaded computation.

Just after execution starts, core processor with id 0 executes the first task in the DAG. When a core visits a vertex in the DAG, it marks the task as visited in the multithreaded computation manager, then looks for the children of this task that have not been enqueued by another core and adds them to its ready dequeue following the rules of the Work stealing algorithm governing the execution. Then, the core checks its ready dequeue and executes the next task according to the rules of the Work stealing algorithm. Recall that if all the parents of the task that the processor core is attempting to execute have not been executed yet, the core stalls meaning that it looks in its ready dequeue for another task to execute. The search can be made from the top-most task to the bottom-most or the other way around according to the policies of the Work stealing algorithm. If the core fails to find a task ready to execute, it begins work stealing.

Remember, a processor core 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 core informs the multithreaded computation manager that is looking for a vertex to steal. Multithreaded computation manager adds the core to the pool of processor cores looking for vertices to steal and puts it on hold. Meanwhile, the stealing controller is looking for "victims" among the cores that are not stealing, i.e., is looking for cores that have tasks available in their ready dequeues to be stolen. A core is chosen as victim if it has more than one task in its ready dequeue. In this case it gives to to the multithreaded computation manager the top-most or bottom-most task in its ready dequeue according to the Work stealing algorithm. The multithreaded computation manager gives the stolen task to one of the cores in the pool and in the next step this attempts to execute the stolen task. It may happen that all the cores are stealing at the same time which produces a dead-lock. To address this problem, if the number of cores work stealing at any given step equals the number of cores executing

the computation, the pool of cores in hold is emptied and, in the next step, each core attempts to execute the top-most or bottom-most task of its ready dequeue according to the rules of the Work stealing algorithm.

Chapter 3

Tests & Results

3.1 Design of experiments

As mentioned in Section 2.2, we intend to benchmark and quantify the performance of the three Work stealing algorithms under evaluation which are FIFO, LIFO, and the priority-based one. For our benchmarks, we measured both the total execution time of the program, and the load of each processor core. For the first, we measure the execution time of each core involved in the execution and then we take the maximum among the obtained values. For the latter, we use a counter variable in each class modelling a core. Such variable counts the number of tasks or vertices that each core visits.

The tests were performed on DAGs of 50, 100, 200, 400, 800, and 1600 vertices with density values of 0.2, 0.5, and 0.8, so that we have diversity on the topologies of the graphs for the evaluation. Additionally, we vary the number of processor cores, executing the multithreaded computation, between 1 and 4. For every possible configuration we run 20 iterations, for each one of the three Work stealing algorithms reviewed in this thesis.

All benchmarks were conducted on a four-core Intel Core i5 with 2.4GHz running macOS Big Sur. The version of the JVM (Java Virtual Machine), were the tests runned, was 1.8.

3.2 Results & Graphs

In the following graphs we show the average times and core loads obtained for all tested configurations. In the first set of graphs we compare the execution time, which is given in nanoseconds, of each Work stealing algorithm, varying the number of cores for fixed number of vertices and density DAGs. The second set compares the load of each core when executing a multithreaded computation with a fixed number of vertices and density. We refer to the processor cores as P0, P1, P2 and P3.

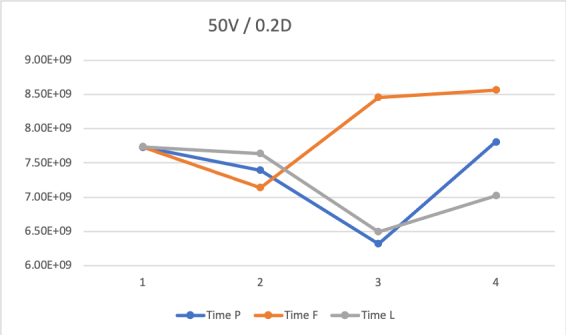

Figure 3.1: 50 vertices - 0.2 density. Time vs. Number of processors graph

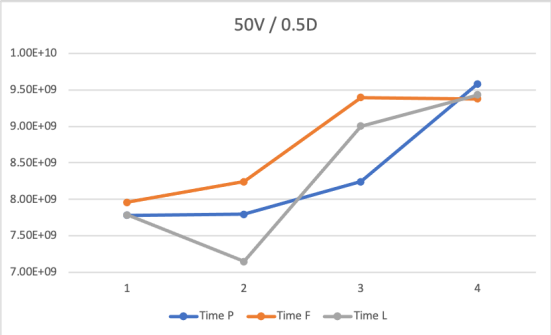

Figure 3.2: 50 vertices - 0.5 density. Time vs. Number of processors graph

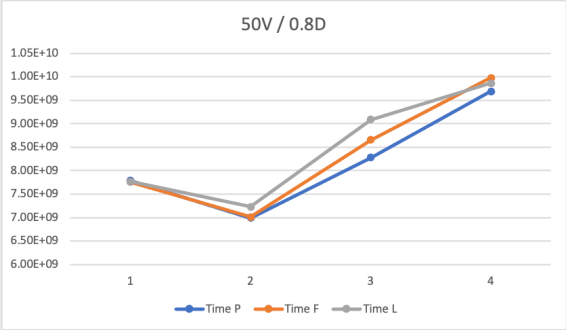

Figure 3.3: 50 vertices - 0.8 density. Time vs. Number of processors graph

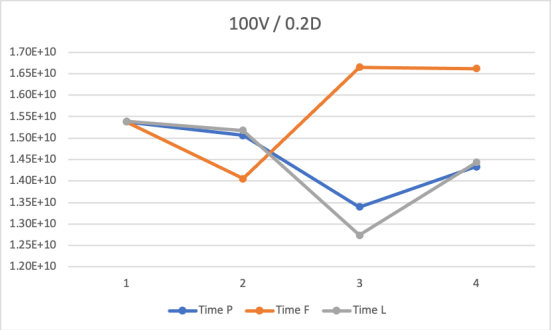

Figure 3.4: 100 vertices - 0.2 density. Time vs. Number of processors graph

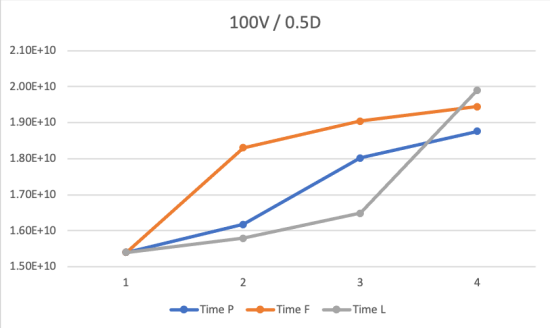

Figure 3.5: 100 vertices - 0.5 density. Time vs. Number of processors graph

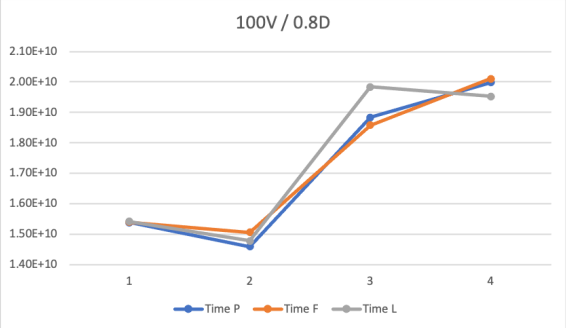

Figure 3.6: 100 vertices - 0.8 density. Time vs. Number of processors graph

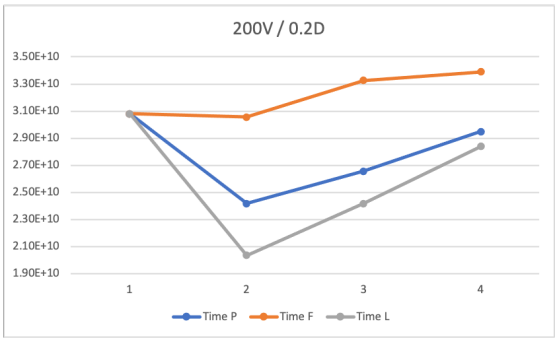

Figure 3.7: 200 vertices - 0.2 density. Time vs. Number of processors graph

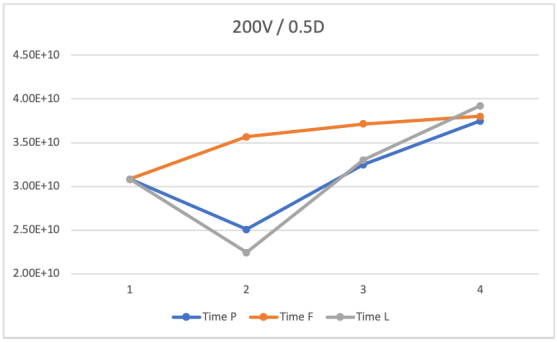

Figure 3.8: 200 vertices - 0.5 density. Time vs. Number of processors graph

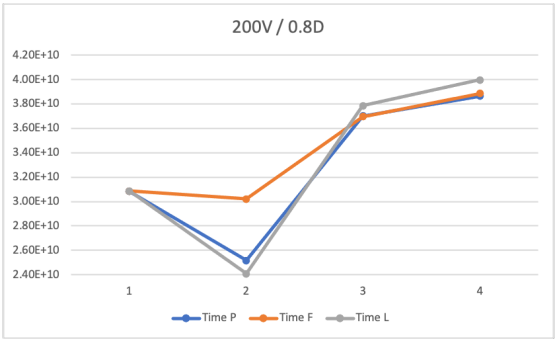

Figure 3.9: 200 vertices - 0.8 density. Time vs. Number of processors graph

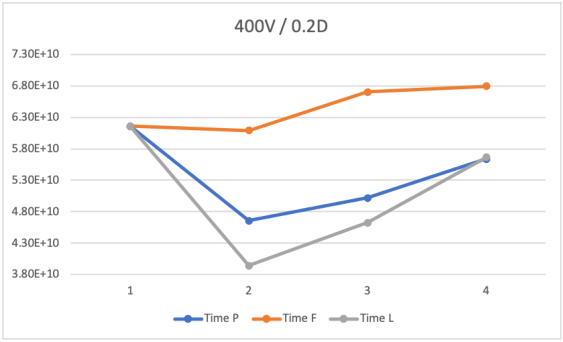

Figure 3.10: 400 vertices - 0.2 density. Time vs. Number of processors graph

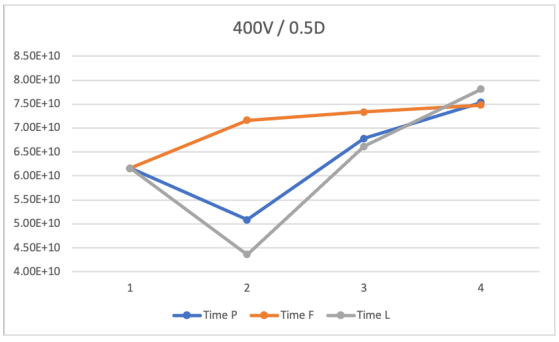

Figure 3.11: 400 vertices - 0.5 density. Time vs. Number of processors graph

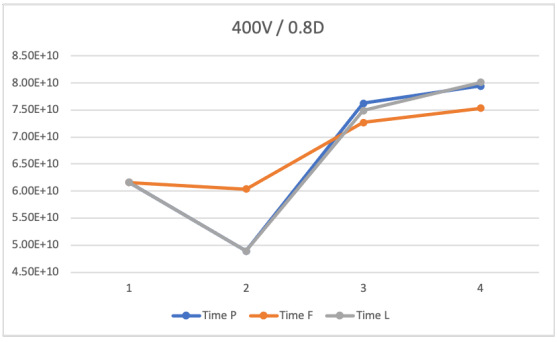

Figure 3.12: 400 vertices - 0.8 density. Time vs. Number of processors graph

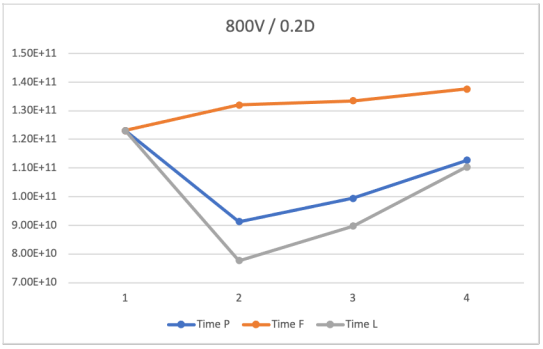

Figure 3.13: 800 vertices - 0.2 density. Time vs. Number of processors graph

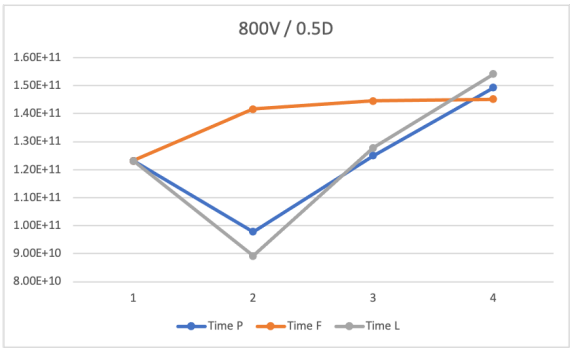

Figure 3.14: 800 vertices - 0.5 density. Time vs. Number of processors graph

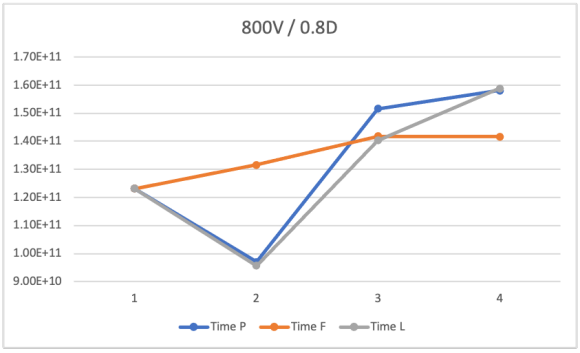

Figure 3.15: 800 vertices - 0.8 density. Time vs. Number of processors graph

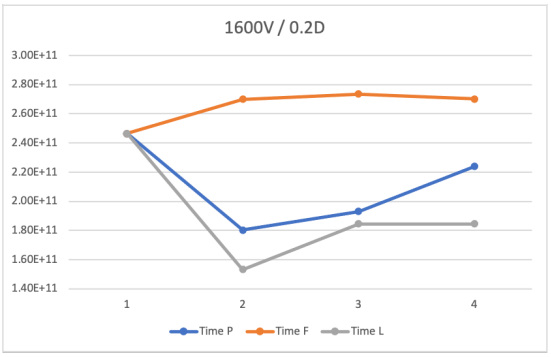

Figure 3.16: 1600 vertices - 0.2 density. Time vs. Number of processors graph

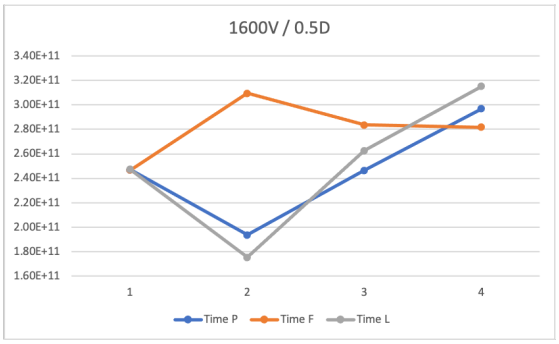

Figure 3.17: 1600 vertices - 0.5 density. Time vs. Number of processors graph

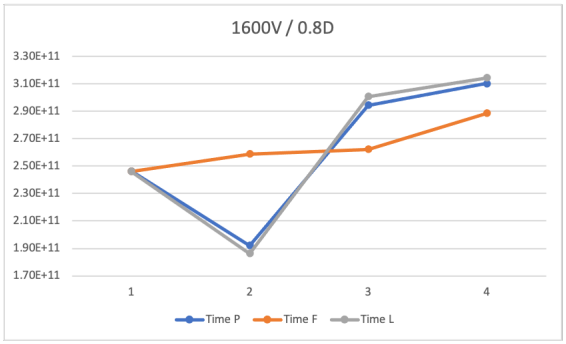

Figure 3.18: 1600 vertices - 0.8 density. Time vs. Number of processors graph

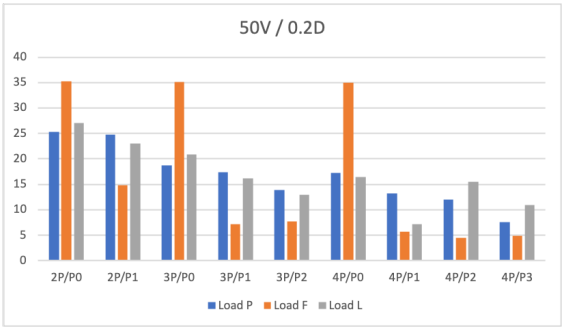

Figure 3.19: 50 vertices - 0.2 density. Load per processor core graph

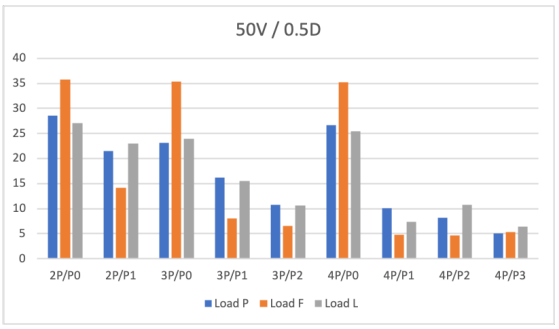

Figure 3.20: 50 vertices - 0.5 density. Load per processor core graph

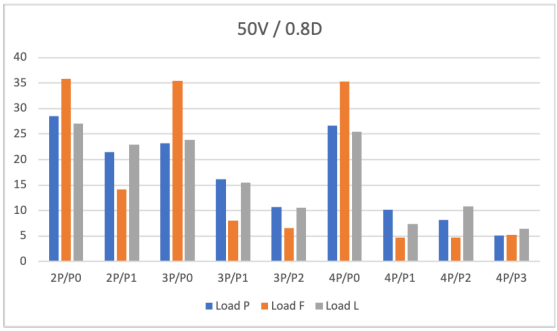

Figure 3.21: 50 vertices - 0.8 density. Load per processor core graph

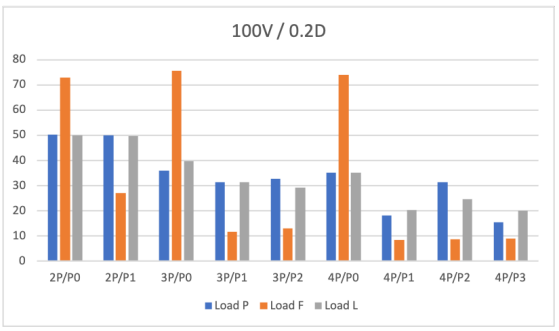

Figure 3.22: 100 vertices - 0.2 density. Load per processor core graph

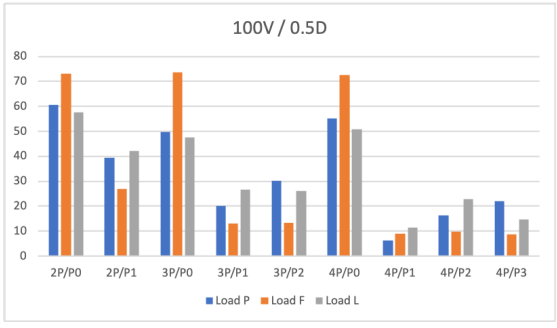

Figure 3.23: 100 vertices - 0.5 density. Load per processor core graph

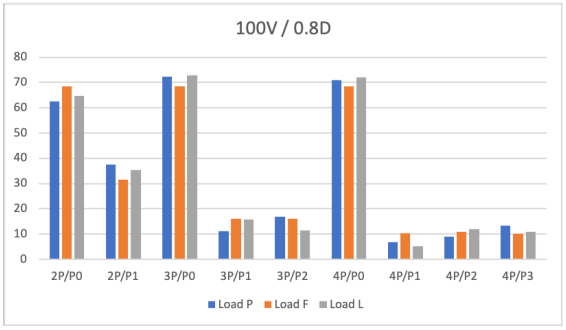

Figure 3.24: 100 vertices - 0.8 density. Load per processor core graph

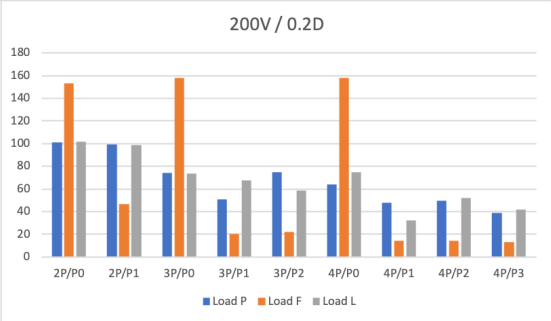

Figure 3.25: 200 vertices - 0.2 density. Load per processor core graph

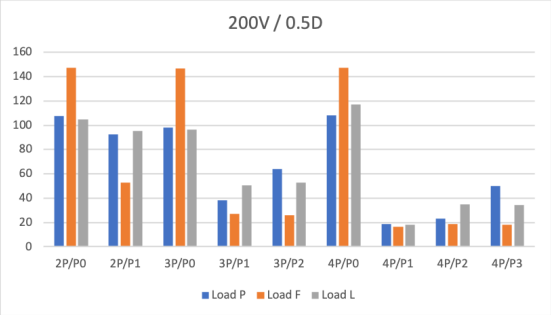

Figure 3.26: 200 vertices - 0.5 density. Load per processor core graph

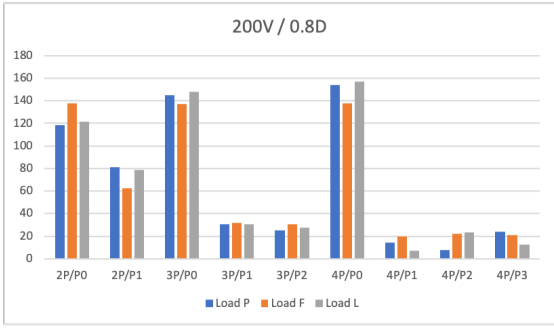

Figure 3.27: 200 vertices - 0.8 density. Load per processor core graph

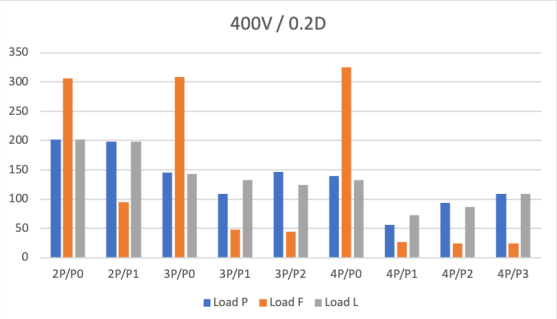

Figure 3.28: 400 vertices - 0.2 density. Load per processor core graph

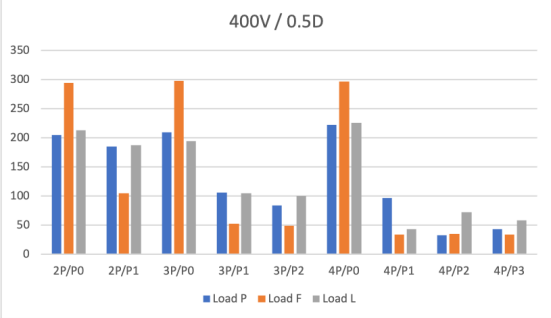

Figure 3.29: 400 vertices - 0.5 density. Load per processor core graph

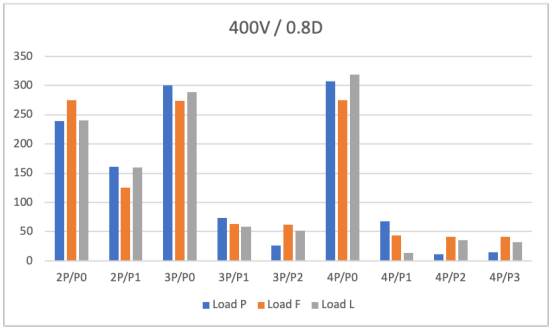

Figure 3.30: 400 vertices - 0.8 density. Load per processor core graph

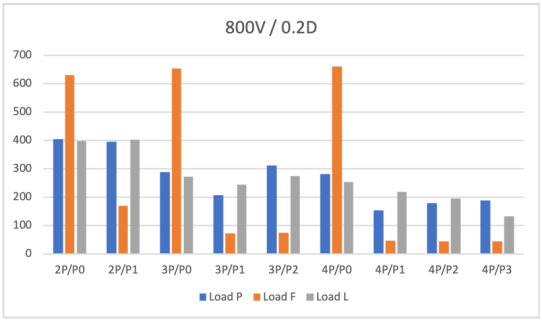

Figure 3.31: 800 vertices - 0.2 density. Load per processor core graph

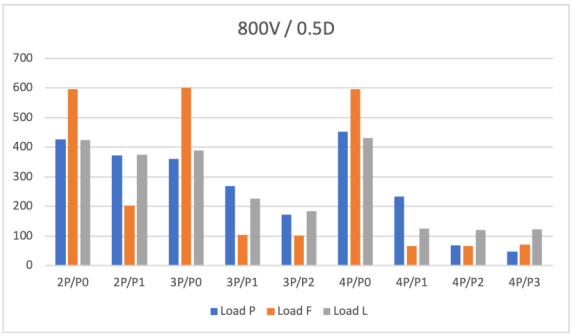

Figure 3.32: 800 vertices - 0.5 density. Load per processor core graph

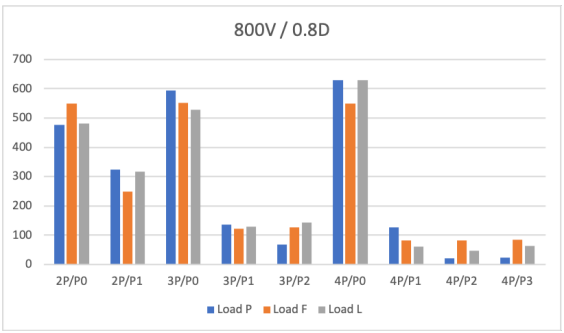

Figure 3.33: 800 vertices - 0.8 density. Load per processor core graph

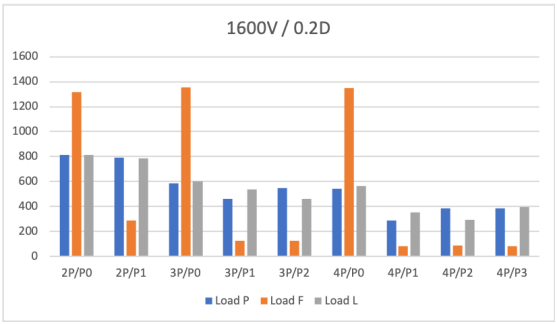

Figure 3.34: 1600 vertices - 0.2 density. Load per processor core graph

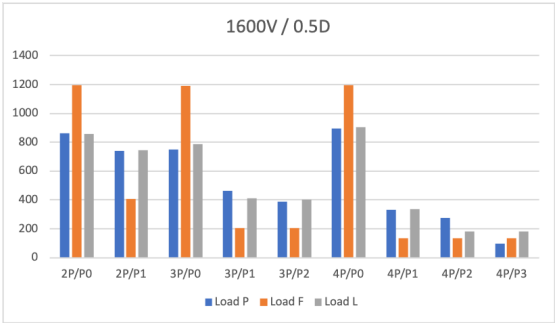

Figure 3.35: 1600 vertices - 0.5 density. Load per processor core graph

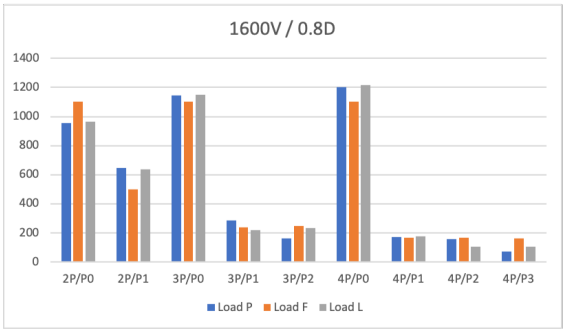

Figure 3.36: 1600 vertices - 0.8 density. Load per processor core graph

3.3 Analysis of the results

Overall, the worst performing algorithm is the FIFO Work stealing algorithm. We notice that the total execution time of the computation is usually higher when the FIFO algorithm is running compared to the other algorithms. Nonetheless, there are configurations in which this algorithm performs better, e.g., 800 vertices and 0.8 density (Figure 3.15) and 1600 vertices and 0.8 density (Figure 3.18) computations running with 4 processor cores. Moreover, the core load when the FIFO Work stealing algorithm is running is usually higher on core P0 regardless of the amount of cores involved in the execution.

When comparing the priority-based algorithm with the LIFO algorithm, the latter performs slightly faster in the majority of the cases. However, as with the FIFO Work stealing algorithm, there are run configurations in which the priority-based algorithm performs better than the LIFO Work stealing algorithm. See, for instance, Figure 3.2, Figure 3.5, Figure 3.14 and Figure 3.17 in which the configurations using 3 and 4 processor cores are faster running the priority-based algorithm, than the LIFO algorithm. Regarding to the core load balances, both the Priority-based Work Stealing algorithm and the LIFO Work stealing algorithm distribute equally the task load among the cores executing the multithreaded computation (cf. Figures 3.19 through 3.36).

When running multithreaded computations with more than two processors, execution times for the LIFO and priority-based algorithms tend to be higher than the time that takes one processor core to run them. Recall, from Subsection 2.2.3, that the program of a multithreaded computation running with n cores works with $n + 2$ threads which stand for the main thread of the program, the processor cores and the stealing controller. Therefore, given the configuration of the machine where tests were executed, one may expect delays on the $n = 3, 4$ cases. As we can observe from the results, the best execution times are obtained when $n = 2$.

We noticed that when the graph is denser, no matter the algorithm we are running, the load among the cores is more unbalanced. In particular, most of the tasks execute on P0. One possible reason for this is that, for instance, when setting 0.8 density, after executing the first task, P0 enqueues in its ready dequeue 80% of the vertices of the DAG modelling the multithreaded computation. Thus on the whole computation the remaining processor cores will be stealing tasks from P0. Moreover, after a core steals a task from P0, it is highly probable that it will need to steal again since the children of the stolen task that this has just executed are likely to already be in P0's ready dequeue.

3.4 Threats to validity

We identify different situations that may add "noise" and bias the internal validity of the results reported in this thesis. These are related to factors that could affect the variables and the relations being investigated.

First of all, tests were conducted on a machine whose processor cores are not dedicated. That is, besides executing the program that benchmarks the work stealing algorithms, the processor cores of the machine, where the tests are held, are also running and managing its operating system.

In second place, the DAGs generated, by Algorithm 1, may diverge from the DAGs modelling a real multithreaded computation. As a matter of fact, there are extra conditions imposed in the topology of a DAG modelling a multithreaded computation [BL99] that we do not take into account in this thesis for the sake of simplicity.

Finally, we want to address the fact that the Work stealing algorithms discussed in this work run, in practice, at core level. However, the simulations that we performed and presented here were written in Java which is a high-level programming language.

Chapter 4

Conclusions & Future work

4.1 Conclusions

The main goal of this work was to understand, review and look for improvement points in the classical Work stealing algorithms. As a result, we proposed a new Work stealing algorithm based on a priority assigned to each vertex in the DAG modelling a multithreaded computation. In order to evaluate and compare the performance of the Work stealing algorithms studied we wrote a program that let us ran experiments and simulate the execution of a multithreaded computation. In these experiments we measured the total execution time and the work load of each processor core involved in the multithreaded computation.

Our results evidence that, on one hand, in most of the runnning configurations, the FIFO Work stealing algorithm falls short in terms of execution time and load balancing among the processor cores. On the other hand, the LIFO Work stealing algorithm is the most efficient in terms of execution time. Both the LIFO and priority-based Work Stealing algorithms distribute equally the work load among the processor cores.

There is not a "perfect" Work stealing algorithm that is best suited for any multithreaded computation. As the results, shown in Section 3.2, indicate there are run configurations in which the FIFO and the priority-based Work stealing algorithms outperform the LIFO algorithm. This suggests that there are conditions in the topology of the DAGs and number of cores in the processor running the multithreaded computation that may favor one algorithm over another.

The Priority-based Work stealing algorithm was not as efficient as we expected. Nonetheless there are improvements that could be made in the priority function in which this algorithm is based. 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. Or, if the DAG we are considering is weighted, the edge's weigths are yet another variable to be considered when defining the priority function.

4.2 Future work

The work presented in this thesis can be furthered in three different research avenues.

First, it would be interesting to determine what running configurations (number of vertices and density of the DAG modelling a multithreaded computation and number of processor cores executing the computation) favor a particular Work Stealing algorithm over the others. This will allow us to determine beforehand which Work stealing algorithm is more suitable for a given run configuration.

The Priority-based Work stealing algorithm presented in this thesis is just a first attempt to improve and optimize the existing Work stealing algorithms. This means that we can keep improving and refining the priority function, in which this algorithm is based, through a graph-theoretical approach.

Finally, to test these Work stealing algorithms in dedicated machines and even in servers will enhance our undertanding on how this algorithms behave and perform. Recall that the tests presented in this work ran on a machine with limited capabilities which could have affected the algorithms' performance.

Bibliography

[ABB00] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, page 1–12, New York, NY, USA, 2000. Association for Computing Machinery.

[AG09] Jonatha Anselmi and Bruno Gaujal. Performance evaluation of work stealing for streaming applications. In Tarek Abdelzaher, Michel Raynal, and Nicola Santoro, editors, *Principles of Distributed Systems*, pages 18–32, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 207–216, New York, NY, USA, 1995. Association for Computing Machinery.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), October 2011.

[Kha11] Mumit Khan. Lecture notes for the course CSE-221 Graduate Operating Systems, April 2011.

[MSR+12] Sebastian Mattheis, Tobias Schuele, Andreas Raabe, Thomas Henties, and Urs Gleim. Work stealing strategies for parallel stream processing in soft real-time systems. In Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2012*, pages 172–183, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[NW10] Daniel Neill and Adam Wierman. On the benefits of work stealing in shared-memory multiprocessors. 2010.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.