Analysis of Work Stealing with latency

Nicolas Gast, Mohammed Khatiri, Denis Trystram, Frédéric Wagner

Analysis of Work Stealing with latency

Nicolas Gast[a], Mohammed Khatiri[a,b], Denis Trystram[a] and Frédéric Wagner[a]

[a] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
[b] University Mohammed First, Faculty of Sciences, LaRI, 60000, Oujda, Morocco
Email: firstname.lastname@inria.fr

ARTICLE INFO

Keywords:
Work Stealing, Latency, Makespan Analysis

ABSTRACT

We study the impact of communication latency on the classical *Work Stealing* load balancing algorithm. Our paper extends the reference model in which we introduce a latency parameter. By using a theoretical analysis and simulation, we study the overall impact of this latency on the Makespan (maximum completion time). We derive a new expression of the expected running time of a *bag of independent tasks* scheduled by Work Stealing. This expression enables us to predict under which conditions a given run will yield acceptable performance. For instance, we can easily calibrate the maximal number of processors to use for a given work/platform combination. All our results are validated through simulation on a wide range of parameters.

1. Introduction

Work stealing (WS) is a distributed on-line scheduling algorithm proposed for shared-memory multi-cores [5]. With work stealing, each processor maintains a local list of tasks to be executed. When a processor has executed all its local work, it probes others by sending work requests to randomly chosen processors. We say that this processor attempts to "*steal*" work from others. Despite its apparent simplicity, work stealing has been proven to be very effective for scheduling tasks in shared-memory homogeneous architectures [8]. Today, the research on WS is driven by the question on how to extend the analysis for the characteristics of new computing platforms (distributed memory, large scale, heterogeneity). Notice that beside its theoretical interest, WS has been implemented successfully in several languages and parallel libraries including Cilk [13, 20], TBB (Threading Building Blocks) [27], the PGAS language [11, 23] and the KAAPI run-time system [15].

In this paper, we study the performance of WS in a distributed-memory context. Such an architecture consists of independent processing elements with private local memories linked by an interconnection network. Communication latency is crucial and highly influences the performance of the applications [18]. The impact of scheduling is important since the whole execution can be highly affected by a large communication latency of interconnection networks [17].

Contributions In this work, we study how communication latency impacts work stealing. Our work has four main contributions. First, we create a new realistic scheduling model for distributed-memory clusters of p identical processors including latency (denoted by λ). Second, we provide an upper bound of the expected makespan to execute a bag of $\mathcal{W}$ independent unitary tasks by p processors. This bound is the sum of two terms. The first is the usual lower bound on the best possible load-balancing $\frac{\mathcal{W}}{p}$; The second term is proportional to $\lambda \log(\frac{\mathcal{W}}{\lambda})$. Third, we develop a lightweight Python simulator for running adequate experiments. Our simulator is developed to be flexible enough to simulate different topologies and applications with different variants of Work Stealing algorithms. The code is available on GitHub (`https://wssimulator.github.io`) along logs of presented experiments. The simulator is generic enough to be used in different contexts of online scheduling and interfaces with standard trace analysis tools Using this simulator, we provide simulation results that challenges this bound. These experiments show that the second additional term has indeed the form $c\lambda \log_2(\frac{\mathcal{W}}{\lambda})$, with $c \approx 3.8$. Note that our theoretical upper bound shows that $c \leq 16.12$. Finally, we show that our methodology can be easily extended to the case of tasks with precedence. We provide an upper bound of the expected makespan in this case. This bound is also composed of two terms. The first is the usual lower bound $\frac{\mathcal{W}}{p}$ and the second additional term is proportional to the critical path ($6\lambda D$)

Our analyses are based on the use of adequate potential functions. While the overall approach is similar to existing work, and in particular [30], there are two technical difficulties that had to be lifted. First, a natural extension of the

ORCID(s):

potential functions used in [30] does not work as one needs to account for the work in transit. Second, we do not assume that task execution and steal requests are synchronized. This leads to consider time steps of duration equal to the communication latency.

The rest of the paper is organized as follows. We review the related works in Section 2. We present the independent task model in Section 3. We derive our main technical result in Section 4 in which we obtain a bound on the makespan. We introduce our work stealing simulator in Section 5 that we use to study the limits of our analysis. We show how to extend our analysis to tasks with precedence in Section 6 and conclude in Section 7. Last, Appendix A contains some technical proofs.

2. Related Work

Let us first introduce the problem of scheduling: It is the process which aims at determining where and when to execute the tasks of a target parallel application. The applications are represented as directed acyclic graphs where the vertices are the basic operations and the arcs are the dependencies between the tasks [10]. Scheduling is a crucial problem which has been extensively studied under many variants for the successive generations of parallel and distributed systems. The most commonly studied objective is to minimize the makespan (denoted by $C_{\max}$) and the underlying context is usually to consider centralized algorithms. This assumption is not always realistic, especially if we consider distributed memory allocations and an on-line setting.

Work Stealing (WS) is a distributed version of scheduling that received sustained attention over the last twenty years. It has been studied originally by Blumofe and Leiserson in [8]. They showed that the expected Makespan of a series-parallel precedence graph with $\mathcal{W}$ unit tasks on p processors is bounded by $E(C_{\max}) \leq \frac{\mathcal{W}}{p} + \mathcal{O}(D)$ where D is the length of the critical path of the graph (its depth). This analysis has been improved in Arora *et al.* [5] using potential functions. The case of varying processor speeds has been studied by Bender and Rabin in [6] where the authors introduced a new policy called *high utilization scheduler* that extends the homogeneous case. The specific case of tree-shaped computations with a more accurate model has been studied in [29]. However, in all these previous analyses, the precedence graph is constrained to have only one source and an out-degree of at most 2 which does not easily model the basic case of independent tasks. Simulating independent tasks with a binary precedences tree gives a bound of $\frac{\mathcal{W}}{p} + \mathcal{O}(\log_2(\mathcal{W}))$ since a complete binary tree of $\mathcal{W}$ vertices has a depth $D \leq \log_2(\mathcal{W})$. However, with this approach, the structure of the binary tree dictates which tasks are stolen. More recently, in [30], Tchiboukjian *et al.* provided the best bound known at this time: $\frac{\mathcal{W}}{p} + c.(\log_2 \mathcal{W}) + \Theta(1)$ where $c \approx 3.24$.

Acar *et al.* [1, 2] studied data locality of WS on shared-memory and focus on cache misses. The underlying model assumes that the execution time takes m time units if the instruction incurs a cache miss and 1 unit otherwise. A steal attempt takes at least s and at most ks steps to complete ($k \geq 1$ multiple steal attempts before succeeding). When a steal succeeds, the thief starts working on the stolen task at the next step. This model is similar to the classical WS model without communication since the thief does not wait several time steps to receive the stolen task.

In complement, [14, 24] provided a theoretical analysis based on a Markovian model using mean field theory. They targeted the expectation of the average response time and showed that the system converges to a deterministic Ordinary Differential Equation. Note that there exist other results that study the steady state performance of WS when the work generation is random including Berenbrink *et al.* [7], Lueling and Monien [22] and Rudolph *et al.* [28].

In all these previous theoretical results, communications are not directly addressed (or at least they are taken implicitly into account by the underlying model). Most studies on WS largely focused on shared memory systems and its performance on modern platforms (distributed-memory systems, hierarchical platform, clusters with explicit communication cost) is not really well understood. The difficulty lies in the problems of communication which become more crucial in modern platforms [4]. Let us briefly review the most relevant works dealing with WS with communications.

Dinan *et al.* [11] implemented WS on large-scale clusters, and proposed to split each tasks queues into a part accessed asynchronously by local processes and a shared portion synchronized by a lock which can be access by any remote processes in order to reduce contention. The authors propose an efficient task management mechanism which is to divide the application into a large number of fine grained tasks which generate a large amount of small communications between the CPU and the GPGPU accelerators. However, these data transmissions are very slow. They consider that the transmission time of small data and large data is the same.

Besides the large literature on theoretical works, there exist more practical studies implementing WS libraries where some attempts were provided for taking into account communications: SLAW is a task-based library introduced

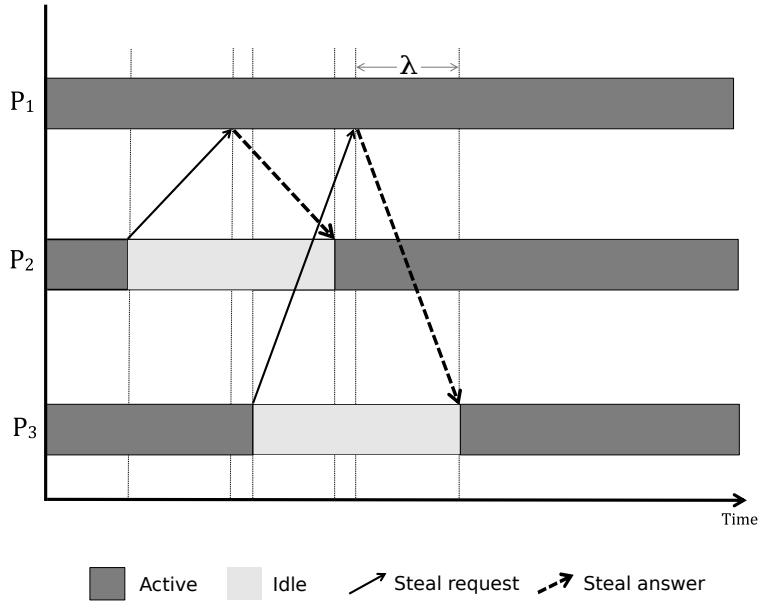


Figure 1: Example of a work stealing execution with 3 processors

in [16], combining work-first and help-first scheduling policies focused on locality awareness in PGAS (Partitioned Global Address Space) languages like UPC (Unified Parallel C). It has been extended in HotSLAW, which provides a high level API that abstracts concurrent task management [23]. [21] proposes an asynchronous WS (AsynchWS) strategy which exploits opportunities to overlap communication with local tasks allowing to hide high communication overheads in distributed memory systems. The principle is based on a hierarchical victim selection, also based on PGAS. Perarnau and Sato presented in [26] an experimental evaluation of WS on the scale of ten thousands compute nodes where the communication depends on the distance between the nodes. They investigated in detail the impact of the communication on the performance. In particular, the physical distance between remote nodes is taken into account. Mullet *et al.* studied in [25] Latency-Hiding, a new WS algorithm that hides the overhead caused by some operations, such as waiting for a request from a client or waiting for a response from a remote machine. The authors refer to this delay as *latency* which is slightly different that the more general concept we consider in our paper. Agrawal *et al.* proposed in [3] an analysis showing the optimality for task graphs with bounded degrees and they developed a library in Cilk++ called *Nabbit* for executing tasks with arbitrary dependencies, with reasonable block sizes.

3. Work-Stealing

In this section, we introduce some formal notations and we present the WS algorithm that we study.

3.1. Notation and definition

We consider a discrete time model. The target parallel platform is composed of p identical processors. We denote by $\mathcal{W}_i(t) \in \{0, 1, \dots\}$ the amount of work on processor P_i at time t (for $i \in \{1 \dots p\}$). A unit of work corresponds to one unit of execution time. We denote the total amount of work on all processors by $\mathcal{W}(t) = \sum_{i=1}^{p} \mathcal{W}_i(t)$. we assume that at $t = 0$ the whole work is located on one processor, say P_1. The total amount of work at time 0 is denoted by $\mathcal{W} = \mathcal{W}_1(0)$.

3.2. Work stealing algorithm

Work Stealing is a decentralized list scheduling algorithm where each processor maintains its own local queue or deque of tasks to execute. When a processor i has nothing to execute, it selects another processor j uniformly at random and sends a work request to it. When a processor j receives this request, it answers by either sending some of its work or by a fail response.

We analyze a model of WS algorithm that has the following features:

- **Latency:** All communication takes a time $\lambda \in \mathbb{N}^+$ that we call the latency. Figure 1 presents an example of Work Stealing execution with latency λ, A work request that is sent at time $t - \lambda$ by a thief will be received at time t by the victim. The thief will then receive an answer at time $t + \lambda$. As we consider a discrete-time model, we say that a work request arrives at time t if it arrives between $t - 1$ (not-included) and t. This means that at time t, this work request is treated. We do not assume that the work requests are synchronized. The number of incoming work requests at time t is denoted by $R(t) \in \{0, 1, \dots, p - 1\}$. It is equal to the number of processors sending a work request at time $t - \lambda$. When a processor i receives a work request from a thief j, it sends a part of its work to j. This communication takes again λ units of time. The processor j receives the work at time $t + \lambda$. We denote by $s_i(t)$ the amount of work in transit from P_i at time t. At end of the communication s_i becomes 0 until another work request arrives.

- **Single work transfer:** We assume that a processor can send some work to at most one processor at a time. While the processor sends work to a thief, it replies by a fail response to any other work request. Hence, the work request may fail in the following cases: (1) when the victim does not have enough work, (2) when it is already sending some work to another thief, or (3) when the victim receives more than one work request at the same time. In the latter, the processor picks random thief and send a negative response to the remaining thieves.

- **Steal Threshold:** The main goal of WS is to share work between processors in order to balance the load and the speed-up execution. In some cases however it might be beneficial to keep work locally and answer negatively to some work requests. We assume that if the victim has less than λ units of work to execute, the work request fails (answering such a work request would increase the makespan as the time to answer a request is λ units of time).

- **Work division:** Since the tasks are independent, we suppose that the victim sends to the thief half of its work:

$$w_i(t) = \frac{w_i(t-1) - 1}{2} \qquad \text{and} \qquad s_i(t) = \frac{w_i(t-1) - 1}{2}.$$

4. Analysis of the Completion Time

This section contains the main result of the paper which is a bound on the expected makespan. Before presenting the detailed analysis, we first describe its main steps before jumping into the technical analysis.

4.1. General principle

We denote by $C_{\max}$ the makespan (*i.e.*, total execution time). In a WS algorithm, each processor either executes work or tries to steal work. As the round-trip-time of a communication is 2λ and the total amount of work is equal to $\mathcal{W}$ and the number of processors is p, we have $pC_{\max} \leq \mathcal{W} + 2\lambda\#\{\text{Work Requests}\}$ where p is the number of processors. This leads to a straightforward bound of the Makespan:

$$C_{max} \leq \frac{\mathcal{W}}{p} + 2\lambda \frac{\#\{\text{Work Requests}\}}{p} \tag{1}$$

Note that the above inequality is not an equality because the execution might end while some processors are still waiting for work.

The key element of our analysis is to obtain a bound on the number of work requests. For that, we use what we call a potential function that represents how the tasks are (un)balanced. We bound the number of work requests by showing that each event involving a steal operation contributes to the decrease of the potential. Our approach is similar to the one of [30] but with one additional key difficulty: communications take λ time units. At first, it seems that longer communications should translate linearly into the time taken by work requests but this would neglect the fact that longer communications also reduce the number of work requests.

In order to analyze the impact of λ, we reconsider the time division as periods of duration λ. We analyze the system at each time step $k\lambda$ for $k \in \mathbb{N}$. By abuse of notation, we denote by $w_i(k)$ and $s_i(k)$ the quantities $w_i(k\lambda)$ and $s_i(k\lambda)$. We denote the total number of incoming work requests in the interval $(\lambda(k-1), \lambda k]$ by $r(k) = \sum_{j=1}^{\lambda} R((k-1)\lambda + j)$ and we denote by $q(r(k))$ the probability that a processor receives one or more requests in the interval $(\lambda(k-1), \lambda k]$ (this function will be computed in the next section). Note that since a steal requests takes at least λ units of time to be answered, we have $0 \leq r(k) \leq p$.

The key steps of the analysis are as follows:

1. First, we will define a potential function $\phi(k)$ that is such that we can bound the expected decrease of the potential as a function of $r(k)$, the number of work requests in the time interval $(\lambda(k-1), \lambda k]$:

   $$\mathbb{E}\left[\phi(k+1) \mid r(k)\right] \leq h(r(k))\phi(k).$$

   This will be done in Lemma 1.

2. Second, we will show that this bound implies that the number of work requests is upper bounded by $\gamma \log_2 \phi(0)$, where $\gamma = \max r/(-p \log_2 h(r))$. This will be done in Lemma 2.

3. We will then obtain a bound on the Makespan by using Equation (1).

4.2. Expected Decrease of the Potential

Our results are based on the analysis of the decrease of the potential. The potential at time-step $k\lambda$ is denoted by $\phi(k)$ and is defined as :

$$\phi(k) = 1 + \frac{1}{\lambda^2} \sum_{i=1}^{p} \left(w_i(k)^2 + 2s_i(k)^2 \right). \tag{2}$$

We also denote by $\phi_i(k) = w_i(k)^2 + 2s_i(k)^2$ the contribution of processor i to the potential (for $i \in \{1 \dots, p\}$).

The rational behind the definition of Equation (2) is as follows. The potential is maximal when all the work is contained in one processor which is the potential function at time 0 and is equal to $\phi(0) = 1 + \mathcal{W}^2/\lambda^2$. The schedule completes when the potential reaches 1. Up to the multiplicative factor $1/\lambda^2$, the rest of Equation (2) is composed in two terms: $\sum_{i=1}^{p} w_i(k)^2$ and $2\sum_{i=1}^{p} s_i(k)^2$. These terms serve to measure how unbalanced is the work. The first term is only here to ensure that the potential is never smaller than 1 (this is a technical condition that ensures that $\log \phi(k) \geq 0$ and that is used in the proof of Lemma 2).

The following lemma shows that $\phi(k)$ decreases in expectation with k. In this lemma, we denote by $\mathcal{F}_k$ all the events up to the interval $((k-1)\lambda, k\lambda]$. The notation $\mathbb{E}[\phi(k+1) \mid \mathcal{F}_k]$ indicated the conditional expectation of $\phi(k+1)$ given all information available at time k. We show that it can be upper bounded by a simple function of the potential at time step k and of the number of work requests $r(k)$.

Lemma 1. *The expected ratio between $\phi(k+1)$ and $\phi(k)$ knowing $\mathcal{F}_k$ is bounded by:*

$$\mathbb{E}[\phi(k+1) \mid \mathcal{F}_k] \leq h(r(k))\phi(k),$$

where the potential is defined as in Equation (2) and

$$h(r) = \frac{3}{4} + \frac{1}{4}\left(\frac{p-2}{p-1}\right)^r. \tag{3}$$

Proof (Sketch of proof). Each event related to a steal request decreases the potential:

- When a steal arrives at a processor with w_i jobs, approximately $w_i/2$ jobs remain on this processor while $w_i/2$ jobs go into a term s_i. In Equation (2), this transforms a w_i^2 into $(w_i/2)^2 + 2(w_i/2)^2 = 3w_i^2/4$, leading to a decrease of potential by a factor $3/4$ for each successful steal.
- When some work arrive at a processor, a term s_i of Equation (2) is transformed into a term w_i. Because of the factor 2, in the potential, this transforms $2s_i^2$ into s_i^2.

The proof of Lemma 1 can be obtained by computing the probability that a given processor receives a work request. This probability depends of the number of work active processors. The full proof is detailed in Appendix A.1.

4.3. Analysis of the makespan

We are now ready to prove the bound on the total completion time $C_{\max}$. To show that, we first obtain a bound on the Number of Work Requests by using Lemma 1. Let us define the constant γ as follows:

$$\gamma \stackrel{\text{def}}{=} \max_{1\leq r\leq p} \frac{r}{-p\log_2(h(r))}, \tag{4}$$

where h is defined in Equation (3).

Lemma 2. *Let $\phi(0)$ denote the potential at time 0 and let τ be the first time step at which the potential reaches 1. Then, the number of incoming work requests until τ, $R = \sum_{k=0}^{\tau-1} r(k)$, satisfies:*

$$(i) \qquad \mathbb{E}\left[R\right] \leq p\gamma \log_2 \phi(0);$$
$$(ii) \qquad \mathbb{P}\left[R \geq p\gamma(\log_2 \phi(0) + x)\right] \leq 2^{-x}.$$

The constant γ is such that $\gamma < 4.03$.

Proof (Sketch of Proof). We detail here an informal proof that assumes that the evolution of the potential satisfies $\phi(k+1) \leq h(r(k))\phi(k)$ (without the conditional expectation of Lemma 1). This allows us to make a simpler exposition of the main idea of the proof. Because of the stochastic evolution of $\phi(k)$, the real proof is more technical and requires the use of Martingale arguments and of Jensen's inequality. It is detailed in Appendix A.2.

Assume that when r steal requests arrive during a time-slot, the expected potential is multiplied by $h(r) < 1$. If τ is such that $\Phi(\tau) = 1$, this shows that if the number of steal requests that arrive per time slots 0 to $\tau - 1$ are $r(0)$, $r(1)$,$\ldots r(\tau-1)$, then $1 = \phi(\tau) \leq h(r(1)) \ldots h(r(k))\phi(0)$, which implies that $-\log_2 h(r(1)) - \cdots - \log_2 h(r(k)) \leq \phi(0)$. The definition of γ in Equation (4) is a worst case analysis of what would happen if an adversary could choose the $r(k)$. Equation (4) implies that $r(k) \leq -\gamma p \log h(r(k))$. This shows that $R \leq \gamma p \log_2 \phi(0)$.

Lemma 2 provides a bound on the number of work requests before the end of the schedule. By using Equation (1), this translates almost immediately into a bound on the makespan, as summarized by the following theorem.

Theorem 3. *Let C_{max} be the Makespan of $\mathcal{W}$ unit independent tasks scheduled by WS with latency λ. Then,*

$$(i) \qquad \mathbb{E}\left[C_{max}\right] \leq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2 \frac{\mathcal{W}}{\lambda} + 2\lambda\gamma;$$
$$(ii) \qquad \mathbb{P}\left[C_{max} \geq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2 \frac{\mathcal{W}}{\lambda} + x\right] \leq 2^{-x/(2\lambda\gamma)}.$$

The constant γ is the same as in Lemma 2. In particular $\gamma < 4.03$.

Proof. By Lemma 2, the number of incoming work requests until τ is bounded by $p\gamma \log_2 \phi(0)$, with $\phi(0) = 1 + \mathcal{W}^2/\lambda^2$. Moreover by definition, the schedule is finished at time τ. Thus by Equation (1) we have

$$\mathbb{E}\left[C_{max}\right] \leq \frac{\mathcal{W}}{p} + 2\lambda\gamma \log_2\left(1 + \frac{\mathcal{W}^2}{\lambda^2}\right) \leq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2\left(\frac{\mathcal{W}}{\lambda}\right) + 2\lambda\gamma,$$

where we used that $\log_2(1+x) \leq 1 + \log_2(x)$ for $x \geq 1$.

By the same way, we use Lemma 2 (ii) and Equation 1 we obtain:

$$\begin{aligned}\mathbb{P}\left[C_{max} \geq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2\left(\frac{\mathcal{W}}{\lambda}\right) + 4\lambda\gamma + x\right] &\leq \mathbb{P}\left[2\lambda\frac{R}{p} \geq 2\lambda\gamma \log_2 \phi(0) + x\right] \\ &= \mathbb{P}\left[R \geq p\gamma \log_2\left(\frac{\mathcal{W}}{\lambda}\right) + p\gamma\frac{x}{2\lambda\gamma}\right] \leq 2^{-x/(2\lambda\gamma)}.\end{aligned}$$

5. Experiments Results for Independent tasks

In the previous section, we proved a new upper bound of the Makespan of WS with an explicit latency. The objective of this section is to study WS experimentally in order to confirm the theoretical results and to refine the constant γ. To this end, we developed an *ad-hoc* simulator that follows strictly the WS model. We focus on the case of independent tasks. We start by describing this simulator and the considered test configurations. Using the experimental results, we show that the previous theoretical bound is close to the experimental results. We conclude with a discussion on where would the analysis be made more precise.

5.1. Simulator

Simulation analysis is a very popular and powerful method for understanding complex problems when mathematical analysis is unreachable.

For this purpose, there exist many simulators on parallel and distributed computing. Most of them are developed for specific research projects by researchers and are not well-documented, and/or no longer maintained. However, there exist several general purpose and high quality simulators like SimGrid [9] that includes many features: It allows to consider complex situations like congestion, cache effects for particular architectures, etc. However, such simulators are usually very computationally expensive, and it requires a high entry cost to develop in.

Our purpose here is less ambitious since we target simple processing units. We are interested in observing a simple aspect of the execution process of WS algorithms on platforms with different topologies. For these reasons, we developed a lightweight simulator, Its flexibility allows us to experiment several variants of WS algorithms, with different topologies, and different types of application. The simulator generates also extensive logs for a detailed analysis for each tested scenario.

WS-simulator can run different models of execution under the WS paradigm of an application on a platform. An application consists of a list of tasks with or without dependencies, and the platform consists of multiple processors linked by a specific network/topology. The simulator allows to execute a scenario with a specific task on a specific platform.

The overall architecture of WS-Simulator is composed of six main units (called engines), that are depicted in Figure 2. The event engine is the core of the simulator, it manages the processors' events during the time to run the simulation of a scenario. The events are executed through the processor engine, which provides different functionalities to perform the tested WS algorithm. The processor engine uses the task engine to manage the execution of the tasks and it uses the topology engine to manage the interactions between the processors. During the execution of a simulation, the log engine keeps track of different information and generates different logs. More details are available at the website `https://wssimulator.github.io` which describes in detail our simulator and how to use it.

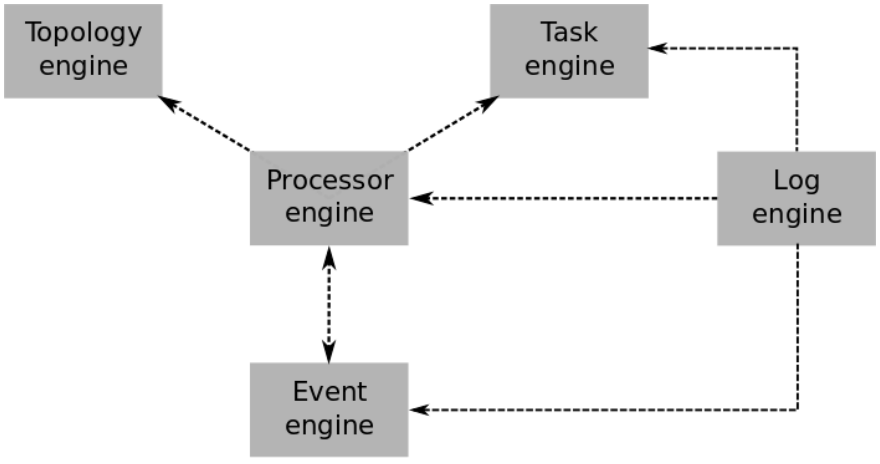


Figure 2: The overall architecture of WS-simulator

5.2. Configurations

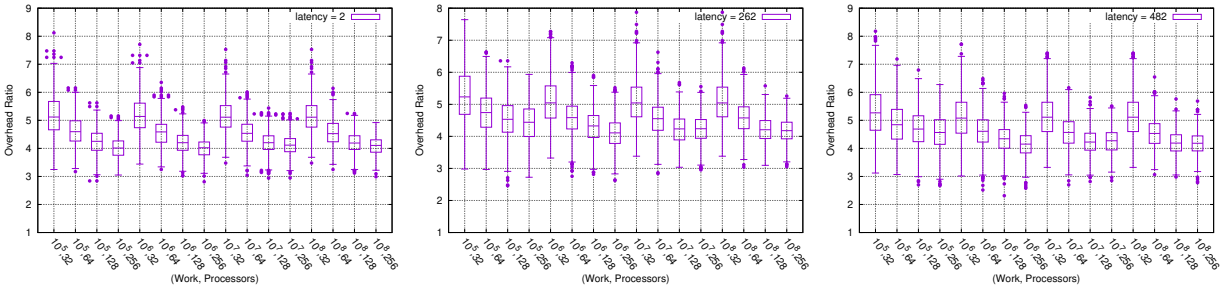
For our tests, we configure our WS-simulator in order to follow the model of independent tasks described in Section 3 to schedule $\mathcal{W}$ unitary independent tasks on a distributed platform composed of p identical processors. The communication latency between the processors is equal to λ. To ensure reproducibility, this configuration of the simulator is detailed in WS-Simulator website[1].

5.3. Validation of the bound and definition of the "overhead ratio"

As seen before, the bound of the expected Makespan is the sum of two terms: The first term is the ratio $\mathcal{W}/p$ which does not depend on the configuration nor on the algorithm; The second term represents the overhead related to work requests. Our analysis bounds the second term to derive our bound on the Makespan. To analyze the validity of our bound, we define what we call the *overhead ratio* as the ratio between the second term of our theoretical bound ($4\gamma\lambda\log_2(\mathcal{W}/\lambda)$) and the simulated execution time minus the ratio $\mathcal{W}/p$: for a given simulation, we define

$$\text{Overhead\_ratio} = \frac{4\gamma\lambda\log_2(\mathcal{W}/\lambda)}{\text{Simulated\_execution\_time} - \frac{\mathcal{W}}{p}}. \tag{5}$$

To study this overhead ratio, we vary the number of unit tasks $\mathcal{W}$ between 10^5 and 10^8, the number of processors p between 32 and 256 and the latency λ between 2 and 500 which is a realistic range of actual systems. Each experimental setting has been reproduced 1000 times in order to compute median or interquartile ranges.



(a) Latency $\lambda = 2$ (c) Latency $\lambda = 262$ (d) Latency $\lambda = 282$

Figure 3: Overhead ratio (defined in Equation (5)) as a function of $(\mathcal{W}, p)$ for different values of latency λ.

Figure 3 plots the overhead ratio according to each couple $(\mathcal{W}, p)$, for different latency values $\lambda = \{2, 262, 482\}$ units of time. The x-axis is $(\mathcal{W}, p)$ for all values of $\mathcal{W}$ and p intervals and the y-axis shows the overhead ratio. We use here a BoxPlot graphical method to present the results. BoxPlots provides a good overview and a numerical summary of a data set. The "interquartile range" in the middle part of the plot represents the middle quartiles where 50% of the results are presented. The line inside the box presents the median. The whiskers on either side of the IQR represent the lowest and highest quartiles of the data.

We observe that our upper bound is about 4 to 5.5 times greater than to the one computed by simulation (depending on the range of parameters). The ratio between the two bounds decreases with the number of processors but seems fairly independent of $\mathcal{W}$.

5.4. Discussion : where does the overhead ratio come from?

The challenge of this paper is to analyze WS algorithm with an explicit latency. We presented a new analysis which derives a bound on the expected Makespan for a given $\mathcal{W}$, p and λ. It shows that the expected Makespan is bounded by $\mathcal{W}/p$ plus an additional term bounded by $4\gamma\lambda\log_2(\mathcal{W}/\lambda)$ with $4\gamma \approx 16$. As observed in Figure 3, the constant 4γ is about four to five times larger than the one observed by simulation. A more precise fitting based on simulation results leads to the expression $\mathcal{W}/p + 4\lambda\log_2(\mathcal{W}/\lambda)$ (the value 4 is an average over all our experiments). We explain below where does the discrepancy between the theoretical bound of 16 and the experimental result of 4 come from by looking at the different steps of the proof. Our analysis makes essentially three approximations: -1) The function $h(r)$ is an upper bound on the potential diminution. -2) We consider a worst case scenario for the number of steal requests

[1] https://wssimulator.github.io/pages/wssimulator-one-cluster.html

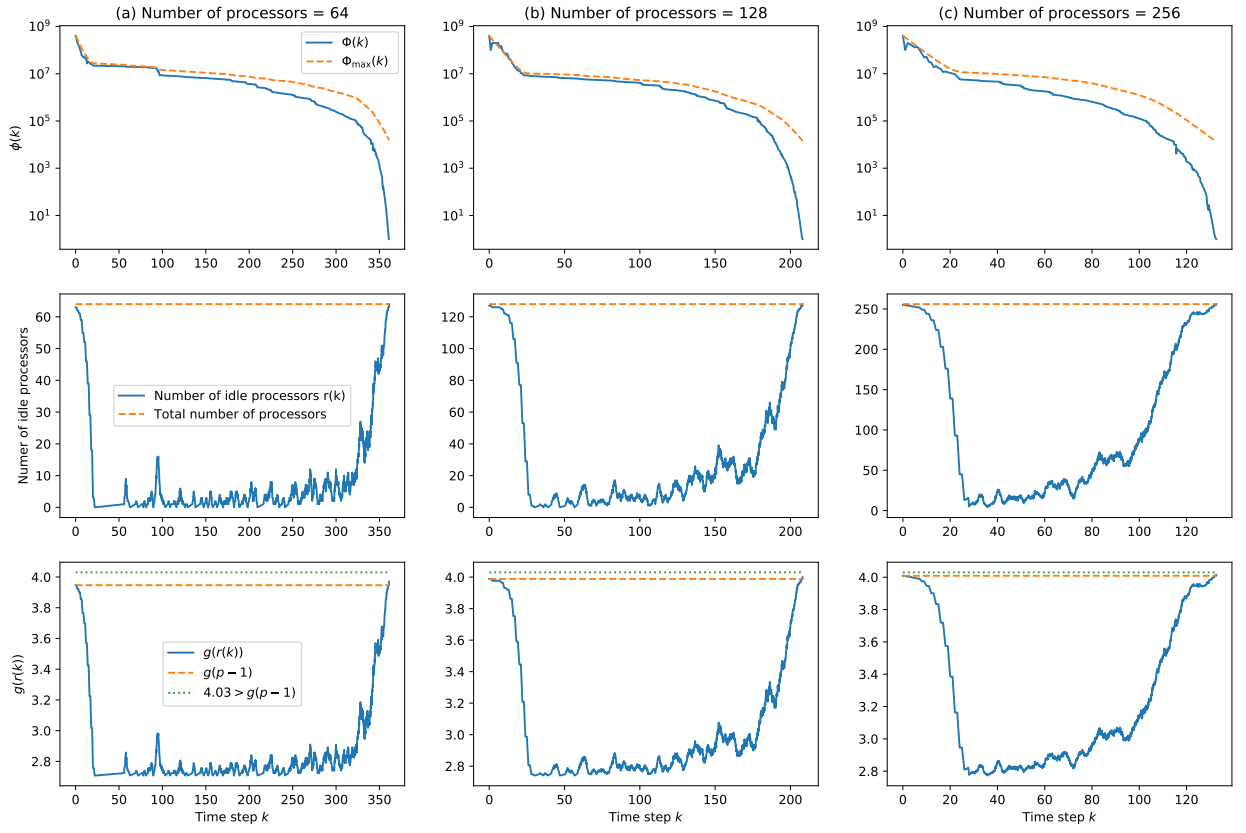


Figure 4: Evolution of the state of the systems for different number of processors ($\lambda = 500$, $W = 10^7$). Each column corresponds to a number of processors (64, 128 or 256). The first line corresponds to $\phi_{\max}$ (defined in Equation (7)). The second line displays the number of idle processors $r(k)$. The third line displays $g(r(k))$ (defined in Equation (12)). In all figures, the x-axis corresponds to the number of time steps k.

when we define $\gamma = \max_r g(r) = g(p-1)$; and (3) We bound γ by 4.03. We review below the contribution of each approximation to the overhead ratio.

To illustrate our explanations, we run three simulations by using our simulator and report the results in Figure 4. These three executions have the same number of tasks $W = 10^7$ and latency $\lambda = 500$ and we consider three values of processors: $p \in \{64,\ 128,\ 256\}$. This figure illustrates the evolution with time of the potential $\phi(k)$, the number of idle processors and the value of γ defined in section 4.3. Each column represents a value of p. The lines represent the results for each metric.

5.4.1. Impact of the bound $h(r)$

The first step of our analysis is to prove a bound on the decrease potential: We show in Lemma 1 that

$$\mathbb{E}\left[\Phi(k+1) \mid \mathcal{F}_t\right] \leq h(r(k))\Phi(k). \tag{6}$$

This bound is obtained by computing the diminution of the potential in the various cases of the proof of Lemma 1. These various cases make different approximations. First, our bound $h(r)$ is the maximum between the ratio $2/3$ of Case 1 and the ratio $h(r)$ of Case 2a. Second, we assumed that we do not know when a work requests arrive in the interval. We therefore always took the worst case (arrivals at the end of the intervals). Third, we neglect the diminution of the potential due to working processors (Case 2a).

To see measure the impact of this approximation, we define a theoretical function $\phi_{\max}(k)$ that would corresponds

to the potential of the system if the inequality of Equation (6) was an equality:

$$\phi_{\max}(k) = \begin{cases} \phi(0) & \text{if } k = 0, \\ h(r(k))\phi_{\max}(k-1) & \text{otherwise.} \end{cases} \tag{7}$$

In Figure 4-(Line 1), we plot this theoretical function and the real potential function as a function of time step k. This figure indicates that the distance between the real potential and the theoretical bound is relatively small at the beginning of the execution, which makes sense since the diminution of the potential is dominated by the diminution related to Case 2a. The two function starts diverging slowly in the middle of the execution and this divergence is accentuated at the end: when the execution is close to finishing, the actual potential decreases much faster that its bound. We believe that this divergence is mostly due to neglecting the diminution of potential due to working processors: At the beginning of the execution when all processors have many tasks to execute, neglecting working processors is negligible; At the end of the execution when the remaining work is small, the potential diminution is greatly affected by the working processors. Our analysis could be tightened by taking a more complex potential function that will take more carefully the impact of working processors.

5.4.2. Evolution of the number of work requests

In our analysis, we study how the potential decreases a function of the number of steal requests $r(k)$. To obtain a bound, we then use a worst-case analysis and define γ as the maximal of a function $g(r) = r/(-p\log_2 h(r))$: $\gamma = \max_r g(r)$. As shown in Figure 5, $g(r)$ is between 2.8 where r is small to 4 when r is large. On the second Line 2 of Figure 4, we depict how the number of idle processors evolve over time. We observe that an execution has essentially three phases: At the beginning, there is a high number of idle processors since the work has to be divided among processors; In the middle of the execution, the number of idle processors is small as everybody is working. In the final execution phase, it increases as finding work becomes harder.

In Figure 4-(Line 3), we plot $g(r(k))$ and γ as a function of the time step k. The figure shows that $g(r(k))$ is often about 2.8 (because the period where the number of idle processors is low is long). This suggests that our bound $\gamma = \max_r g(r)$ is about 1.4 times too high. Being able to capture more precisely how the number of idle processors evolves might lead to a bound that would be around 30% times smaller.

5.4.3. Bound $\gamma < 4.03$ and impact of the number of processors

In our analysis, we show that $\gamma = g(p-1)$ and we bound γ by 4.03. In fact, the value of 4.03 corresponds to what happens when p goes to infinity but smaller values of p leads to smaller values of γ. This explains why in Figure 3, we observe that the overhead ratio decreases with the number of processors, from around 5 for 32 processors to around 4 to 4.5 for 256 processors.

In our simulation, we observe that the overhead ratio is between 4 to 5. Based on our experimental study of the evolution of the number of work requests with time, we believe that the worst-case analysis $\gamma = g(p-1)$ and the bound of 4.03 contributes to a factor of about 1.5 of the overhead ratio. The remaining factor (of about 3) is mostly due to the bound h that we obtained in Lemma 1. Refining this lemma, for example by being able to estimate the decrease of potential due to the work execution or by using a different potential definition, would lead to a tighter bound.

6. Extension to a Model of Tasks with Precedence

In this section, we show that the analysis presented in Section 4 can be adapted to the case of tasks with precedence constraints. The main difficulty here is to redefine a new potential function, which we do by combining the ideas of [5, 30] to our model of latency of Section 4. This demonstrates the power of an approach by potential function.

6.1. Model

We consider a model similar to [5, 30] where the workload is composed of $\mathcal{W}$ unitary tasks that have precedence constraints represented by a DAG (Directed Acyclic Graph). This DAG has a single source that represents the first task and that is originally located on a given processor. We consider that the scheduling is done as in [5]: each processor maintains a double-ended queue (called deque) of activated tasks. If a processor has one or more task in its deque, it executes the task at the head of its deque. This takes one unit of time. After completion, a task might activates 0, 1 or 2 tasks that are pushed at the end of the deque. The activation tree is a binary tree whose shape depends on the execution

of the algorithm. It is a subset of the original DAG and has the same critical path. We define the height of node of this tree as follows. The height of the source as D (*i.e.*, the length of the critical path). The height of another task is equal to the length of its father minus one. We assume that when a processor steals work from another processor, it steals the activated tasks with the largest height. In the case of the DAG of precedence, a processor can only answer positively if it has two or more activated tasks in its deque. In this case, it sends its task that has the largest height.

6.2. Potential function

The potential will depend on the maximal height of the tasks that a processor has in its lists. Denoting by $h_i(k)$ the maximal height of the tasks that processor i has in its deque, we define a quantity $w_i(k)$, that we call the potential work of processor i, as:

$$w_i(k) = \begin{cases} (2\sqrt{2})^{h_i(k)} & \text{if the processor } i \text{ has two or more tasks in its deque,} \\ \frac{1}{2}(2\sqrt{2})^{h_i(k)} & \text{if the processor } i \text{ has only one task in its deque,} \\ 0 & \text{if the processor } i \text{ does not have any tasks.} \end{cases}$$

For the tasks in transit, we define the potential work in transit $s_i(k) = \frac{1}{2}(2\sqrt{2})^h$, where h is the height of the task in transit.

The potential is then defined similarly as in Equation (2):

$$\phi(k) = 1 + \sum_i \phi_i(k) \qquad \text{where } \phi_i(k) = w_i^2(k) + 2s_i^2(k). \tag{8}$$

We are now ready to prove the following lemma, that is an analogue of Lemma 1 for the case of dependent tasks.

Lemma 4. *For the case of DAG, the expected ratio between $\phi(k+1)$ and $\phi(k)$ knowing $\mathcal{F}_k$ is bounded by:*

$$\mathbb{E}[\phi(k+1) \mid \mathcal{F}_k] \leq h(r(k))\phi(k),$$

where the potential is defined as in Equation (8) and $h(r)$ is as in Lemma 1, i.e.*, $h(r) = \frac{3}{4} + \frac{1}{4}\left(\frac{p-2}{p-1}\right)^r$.*

Proof. Similarly to the proof of Lemma 1, we study the expected decrease of the potential by distinguishing three cases: the case $s_i > 0$ (work arriving at a thief) and the cases where a processor is or becomes idle correspond to cases 1 and 4 of Lemma 1 and can be analyzed exactly as what was done since Cases 1 and 4 of the proof of Lemma 1 do not depend on the task dependence model. Moreover, the distinction between Case 2 and Case 3 that was done for independent tasks is not necessary for DAGs as there is no steal threshold for DAG. Last, the analysis of Case 2b of Lemma 1 is similar: if a processor does not receive any work request, the potential does not grow.

Hence, in the reminder of the proof, we focus on the only interesting case which what happens when a processor receives one or more work request (Case 2b of the proof of Lemma 1). We distinguish two cases depending on how many tasks are in the deque of processor i when it receives a work request.

1. If processor i has only one task (say of height h), then it cannot send work. In this case, it will complete its tasks that will activate at most 2 tasks of height $h-1$. In such a case, the potential work of processor i will go from $w_i(k) = \frac{1}{2}(2\sqrt{2})^h$ to at most $w_i(k+1) = (2\sqrt{2})^{h-1}$. This shows that

$$\phi_i(k+1) = w_i^2(k+1) \leq (2\sqrt{2})^{2h-2} = (2\sqrt{2})^{2h}/8 = w_i(k)^2/2 \leq \frac{1}{2}\phi_i(k).$$

2. If processor i has two or more tasks, then by construction, it can have at most two tasks of maximal height (say h). In this case, the processor will send one task (of height h) to the thief, and will at the mean time execute the other task. In this case, the maximal height of the task of its deque will be $h-1$ and the task sent will be of height h. This implies that $w_i(k+1) \leq (2\sqrt{2})^{2h-1} \leq w_i(k)/(2\sqrt{2})$ and $s_i(k+1) \leq \frac{1}{2}(2\sqrt{2})^h \leq w_i(k)/2$. This shows that:

$$\phi_i(k+1) = w_i^2(k+1) + 2s_i^2(k+1) \leq w_i^2(k)/8 + 2w_i^2(k)/4 = \frac{5}{8}\phi_i(k).$$

As both $1/2$ and $5/8$ are strictly smaller than the $3/4$ of Equation (9), the rest of the proof of Lemma 1 can be applied *mutatis mutandis* to prove Lemma 4.

6.3. Analysis of the Makespan

The bound on the expected decrease of the potential of Lemma 4 is the same as the bound of Lemma 1 for independent tasks. Since the proof of Lemma 2 does not depend on the dependence structure of the tasks but only the bound on the expected decrease of Lemma 1, this implies that Lemma 2 also applies for the case of tasks with precedence.

We are now ready to prove the bound on the total completion time $C_{\max}$ for the DAG model that is summarized by the following theorem:

Theorem 5. *Let C_{max} be the Makespan of $\mathcal{W}$ unit tasks with a DAG of precedence that has a critical path of D scheduled by WS with latency λ. Then*

$$(i) \qquad \mathbb{E}\left[C_{max}\right] \leq \frac{\mathcal{W}}{p} + 6\lambda\gamma D;$$

$$(ii) \qquad \mathbb{P}\left[C_{max} \geq \frac{\mathcal{W}}{p} + 6\lambda\gamma D + x\right] \leq 2^{-x/(2\lambda\gamma)}.$$

The constant γ is the same as in Lemma 2. In particular $\gamma < 4.03$.

PROOF. The case of DAG is similar as the independent tasks model the only difference being the expression of the potential at time 0. In the case of a DAG, the potential at time 0 is equal to $1 + \frac{1}{4}(2\sqrt{2})^{2D} = 1 + \frac{1}{4}8^D \leq 8^D$ where D is the critical path. This shows that $\log_2 \phi(0) \leq D \log_2 8 = 3D$ which implies that

$$\mathbb{E}\left[C_{max}\right] \leq \frac{\mathcal{W}}{p} + 6\lambda\gamma D.$$

The bounds that we obtained in Theorem 5 and Theorem 3 are composed of a term due to the computation of tasks ($\mathcal{W}/p$) and an overhead term related to the depth. In case of independent tasks the depth is $\log(W)$ but we obtain a slightly better bound due to tasks not being divided below λ. We actually believe that the bounds on the DAG could be refined when considering non-unit tasks.

7. Conclusion

We presented in this paper a new analysis of Work Stealing algorithm where each communication has a latency of λ. Our main result was to show that the expected Makespan of a load of $\mathcal{W}$ independent tasks on a cluster of p processors is bounded by $\mathcal{W}/p + 16.12\lambda \log_2(\mathcal{W}/\lambda)$. Our analysis makes use of potential functions whose expected decrease per unit time can be bounded as a function of the number of work requests. We then use this to derive a theoretical upper bound on the expected makespan. We also extend this analysis one step further, by providing a bound on the probability to exceed the bound of the makespan. We showed also that we can extend the same analysis for a model of tasks with precedence.

To assess the tightness of this analysis we developed an *ad-hoc* simulator. We showed by comparing the theoretical bound and the experimental results that our bound is realistic. We observed moreover that our bound (established on worst case analysis) is four to five times greater than the experimental results and it is stable for all the tested values. By using traces of execution, we quantify the various approximations that are made in the analysis and suggest where the analysis could be made more accurate.

For future work, we intend to study work stealing strategies in complex topologies where the latency can depend on the topology. This work is a first step towards this as it allows a full understanding of the work stealing in the basic, homogeneous, setting.

References

[1] Acar, U.A., Blelloch, G.E., Blumofe, R.D., 2000. The data locality of work stealing, in: Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, NY, USA. pp. 1–12. URL: http://doi.acm.org/10.1145/341800.341801, doi:10.1145/341800.341801.

[2] Acar, U.A., Blelloch, G.E., Blumofe, R.D., 2002. The data locality of work stealing. Theory of Computing Systems 35, 321–347. URL: https://doi.org/10.1007/s00224-002-1057-3, doi:10.1007/s00224-002-1057-3.

[3] Agrawal, K., Leiserson, C.E., Sukha, J., 2010. Executing task graphs using work-stealing, in: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–12. doi:10.1109/IPDPS.2010.5470403.

[4] Arafat, H., Dinan, J., Krishnamoorthy, S., Balaji, P., Sadayappan, P., 2016. Work stealing for gpu-accelerated parallel programs in a global address space framework. Concurr. Comput. : Pract. Exper. 28, 3637–3654. URL: https://doi.org/10.1002/cpe.3747, doi:10.1002/cpe.3747.

[5] Arora, N.S., Blumofe, R.D., Plaxton, C.G., 2001. Thread scheduling for multiprogrammed multiprocessors, in: In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta, pp. 119–129.

[6] Bender, M.A., Rabin, M.O., 2002. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. Theory of Computing Systems 35, 289–304. URL: http://dx.doi.org/10.1007/s00224-002-1055-5, doi:10.1007/s00224-002-1055-5.

[7] Berenbrink, P., Friedetzky, T., Goldberg, L.A., 2003. The natural work-stealing algorithm is stable. SIAM Journal on Computing 32, 1260–1279. URL: http://dx.doi.org/10.1137/S0097539701399551, doi:10.1137/S0097539701399551, arXiv:http://dx.doi.org/10.1137/S0097539701399551.

[8] Blumofe, R.D., Leiserson, C.E., 1999. Scheduling multithreaded computations by work stealing. J. ACM 46, 720–748. URL: http://doi.acm.org/10.1145/324133.324234, doi:10.1145/324133.324234.

[9] Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F., 2014. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. Journal of Parallel and Distributed Computing 74, 2899–2917. URL: https://hal.inria.fr/hal-01017319, doi:10.1016/j.jpdc.2014.06.008.

[10] Cosnard, M., Trystram, D., 1995. Parallel algorithms and architectures. International Thomson.

[11] Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J., 2009. Scalable work stealing, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA. pp. 53:1–53:11. URL: http://doi.acm.org/10.1145/1654059.1654113, doi:10.1145/1654059.1654113.

[12] Durrett, R., 1996. Probability: theory and examples .

[13] Frigo, M., Leiserson, C.E., Randall, K.H., 1998. The implementation of the cilk-5 multithreaded language. SIGPLAN Not. 33, 212–223. URL: http://doi.acm.org/10.1145/277652.277725, doi:10.1145/277652.277725.

[14] Gast, N., Bruno, G., 2010. A mean field model of work stealing in large-scale systems. SIGMETRICS Perform. Eval. Rev. 38, 13–24. URL: http://doi.acm.org/10.1145/1811099.1811042, doi:10.1145/1811099.1811042.

[15] Gautier, T., Besseron, X., Pigeon, L., 2007. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multiprocessors, in: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, ACM, New York, NY, USA. pp. 15–23. URL: http://doi.acm.org/10.1145/1278177.1278182, doi:10.1145/1278177.1278182.

[16] Guo, Y., Zhao, J., Cave, V., Sarkar, V., 2010. Slaw: A scalable locality-aware adaptive work-stealing scheduler, in: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–12. doi:10.1109/IPDPS.2010.5470425.

[17] Hamid, N., Walters, R., Wills, G., 2015a. Simulation and mathematical analysis of multi-core cluster architecture, in: 2015 17th UKSim-AMSS International Conference on Modelling and Simulation (UKSim), pp. 476–481. doi:10.1109/UKSim.2015.54.

[18] Hamid, N., Walters, R.J., Wills, G.B., 2015b. An analytical model of multi-core multi-cluster architecture (mcmca). URL: http://eprints.soton.ac.uk/374744/.

[19] Khatiri, M., Trystram, D., Wagner, F., 2019. Work stealing simulator. CoRR abs/1910.02803. URL: http://arxiv.org/abs/1910.02803, arXiv:1910.02803.

[20] Leiserson, C.E., 2009. The cilk++ concurrency platform, in: Proceedings of the 46th Annual Design Automation Conference, ACM, New York, NY, USA. pp. 522–527. URL: http://doi.acm.org/10.1145/1629911.1630048, doi:10.1145/1629911.1630048.

[21] Li, S., Hu, J., Cheng, X., Zhao, C., 2013. Asynchronous work stealing on distributed memory systems, in: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 198–202. doi:10.1109/PDP.2013.35.

[22] Lüling, R., Monien, B., 1993. A dynamic distributed load balancing algorithm with provable good performance, in: Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, NY, USA. pp. 164–172. URL: http://doi.acm.org/10.1145/165231.165252, doi:10.1145/165231.165252.

[23] Seung jai Min, C.I., Yelick, K., 2011. Hierarchical work stealing on manycore clusters, in: In: Fifth Conference on Partitioned Global Address Space Programming Models. Galveston Island.

[24] Mitzenmacher, M., 1998. Analyses of load stealing models based on differential equations, in: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, NY, USA. pp. 212–221. URL: http://doi.acm.org/10.1145/277651.277687, doi:10.1145/277651.277687.

[25] Muller, S.K., Acar, U.A., 2016. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing, in: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, ACM, New York, NY, USA. pp. 71–82. URL: http://doi.acm.org/10.1145/2935764.2935793, doi:10.1145/2935764.2935793.

[26] Perarnau, S., Sato, M., 2014. Victim selection and distributed work stealing performance: A case study, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 659–668. doi:10.1109/IPDPS.2014.74.

[27] Robison, A., Voss, M., Kukanov, A., 2008. Optimization via reflection on work stealing in tbb, in: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–8. doi:10.1109/IPDPS.2008.4536188.

[28] Rudolph, L., Slivkin-Allalouf, M., Upfal, E., 1991. A simple load balancing scheme for task allocation in parallel machines, in: Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, NY, USA. pp. 237–245. URL: http://doi.acm.org/10.1145/113379.113401, doi:10.1145/113379.113401.

[29] Sanders, P., 1999. Asynchronous Random Polling Dynamic Load Balancing. Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 37–48. URL: http://dx.doi.org/10.1007/3-540-46632-05, doi:10.1007/3-540-46632-0_5.

[30] Tchiboukdjian, M., Gast, N., Trystram, D., 2013. Decentralized list scheduling. Annals of Operations Research 207, 237–259. URL: http://dx.doi.org/10.1007/s10479-012-1149-7, doi:10.1007/s10479-012-1149-7.

[31] Yang, J., He, Q., 2017. Scheduling parallel computations by work stealing: A survey. International Journal of Parallel Programming , 1–25URL: http://dx.doi.org/10.1007/s10766-016-0484-8, doi:10.1007/s10766-016-0484-8.

A. Technical Proofs

A.1. Proof of Lemma 1

To analyze the decrease of the potential function, we distinguish different cases that corresponds to processors that are executing work, sending or answering work requests. We show that each case contributes to a diminution of the potential.

Between time $k\lambda$ and $(k+1)\lambda$, a processor does (at least) one the following four things. These cases cover all possible behaviors of the processor.

Case 1 ($s_i > 0$) The processor started to send work to another (idle) processor j before time $k\lambda$. This means that processor j will receive $s_i(k)$ tasks at a time $t < (k+1)\lambda$. Note that by assumption, $s_i(k) \geq w_i(k)$ because processor i has executed some if its own work since it started to send half of its work to j. There are two subcases:

- **Case 1a**. If no additional work requests have been received between t and $(k+1)\lambda$, it holds that

$$\begin{aligned} w_i(k+1) &\leq w_i(k) \\ w_j(k+1) &\leq s_i(k) \\ s_i(k+1) = s_j(k+1) &= 0. \end{aligned}$$

  This implies that the potential of i and j at time step $k+1$ satisfies:

$$\begin{aligned} \phi_i(k+1) + \phi_j(k+1) &= w_i(k+1)^2 + w_j(k+1)^2 + 2(s_i(k+1)^2 + s_j(k+1)^2) \\ &\leq w_i(k)^2 + s_i(k)^2 \\ &= w_i(k)^2 + 2s_i(k)^2 - s_i(k)^2 \\ &\leq \frac{2}{3}\phi_i(k) \\ &= \frac{2}{3}(\phi_i(k) + \phi_j(k)). \end{aligned}$$

  The last inequality holds because $w_i(k)^2 + 2s_i^2(k) = \phi_i(k)$ and $s_i^2(k) = (s_i^2(k) + 2s_i^2(k))/3 \geq (w_i^2(k) + 2s_i^2(k))/3 = \phi_i(k+1)/3$. The last equality holds because $\phi_j(k) = 0$.
- **Case 1b**. If one or more work request has been received between t and $(k+1)\lambda$ (by either processor i or j), then this processor will send some of its work of this processor. It should be clear that this will further decrease the potential (see Case 2b below). This shows the inequality :

$$\phi_i(k+1) + \phi_j(k+1) \leq \frac{2}{3}(\phi_i(k) + \phi_j(k))$$

  also holds in this case.

Note that if $w_i < \lambda$, processor i might become idle before $(k+1)\lambda$. In this case, it will send a work request. This will not modify the potential as the work request will be received after time $(k+1)\lambda$.

Case 2 ($s_i = 0$ and $w_i \geq 2\lambda$) The processor has work and it is available to respond to work requests. We distinguish two cases: (case 2a) if this processor receives one or more requests or (case 2b) if it does not receive any request.

- **Case 2a** – If processor i receives one or more work requests between $k\lambda$ and $(k+1)\lambda$, it will respond positively to one processor (say processor j) by sending it half of its work. All other work requests will fail. This implies that $w_i(k+1) \leq w_i(k)/2$ and $s_i(k+1) \leq w_i(k)/2$ and $w_j(k+1) = s_j(k+1) = 0$, which implies that

$$\begin{aligned} \phi_i(k+1) &= w_i^2(k+1) + 2s_i^2(k+1) \\ &\leq \frac{3}{4}w_i^2(k) \\ &= \frac{3}{4}\phi_i(k). \end{aligned} \tag{9}$$

- **Case 2b** – If processor i does not receive any work requests, it will only execute work, in which case $\phi_i(k+1) \leq \phi_i(k)$

A given idle processor will choose processor i as its victim with probability $1/(p-1)$. Hence, if $r(k)$ processors sent a work requests between $(k-1)\lambda$ and $k\lambda$, then processor i will receive no work requests between $k\lambda$ and $(k+1)\lambda$ with probability $((p-2)/(p-1))^{r(k)}$. This shows that Case 2a occurs with probability $1-((p-2)/(p-1))^{r(k)}$ while Case 2b occurs with probability $((p-2)/(p-1))^{r(k)}$. Hence

$$\begin{aligned}\mathbb{E}\left[\phi_i(k+1) \mid \mathcal{F}_k\right] &\leq \phi_i(k)\left[\left(\frac{p-2}{p-1}\right)^{r(k)} + \frac{3}{4}\left(1-\left(\frac{p-2}{p-1}\right)^{r(k)}\right)\right] \\ &= h(r(k))\phi_i(k).\end{aligned}$$

Case 3 ($s_i = 0$ and $\lambda \leq w_i < 2\lambda$) The processor has less than 2λ units of work and therefore may or may not be able to answer work requests depending if they arrive before its remaining work is less than λ units of work. If a work request is received then we fall back to Case 2a. Otherwise, the processor only executes work and

$$\begin{aligned}\phi_i(k+1) &= (\max(0, w_i(k)-\lambda))^2 \\ &\leq \frac{1}{2}w_i(k)^2 \\ &= \frac{1}{2}\phi_i(k).\end{aligned}$$

Case 4 ($s_i = 0$ and $w_i < \lambda$) If processor i is idle or became idle between $k\lambda$ and $(k+1)\lambda$, there are two sub-cases. The first one is if this processor receives some work between $k\lambda$ and $(k+1)\lambda$. In this case this processor is the processor j of the Case 1 above and its contribution to the potential has already been taken into account. The second one is if this processor does not receive work during $k\lambda$ and $(k+1)\lambda$ in which case its potential is $\phi_i(k+1) = 0$.

Note that in addition to all this decrease, at least one processor executed λ units of work during $[k\lambda, (k+1)\lambda)$ (otherwise there would be nothing to compute and the schedule would be finished). This contributes to the decrease of (at least) $\lambda^2/3$ to one of the $\phi_i(k)$.

Using the variation of each of these scenarios we find that the expected potential time $k+1$ is bounded by:

$$\begin{aligned}\mathbb{E}[\phi(k+1) \mid \mathcal{F}_k] &\leq 1 + \frac{1}{\lambda^2}\left(-\frac{\lambda^2}{3} + \sum_{i\in\text{Case 1}}\frac{2}{3}\phi_i(k) + \sum_{i\in\text{Case 2}} h(r(k))\phi_i(k) + \sum_{i\in\text{Case 3}}\frac{1}{2}\phi_i(k) + \sum_{i\in\text{Case 4}} 0\right) \\ &\leq \max\left(\frac{2}{3}, h(r(k)), \frac{1}{2}, 0\right)\phi(k) \\ &= h(r(k))\phi(k),\end{aligned}$$

where the last equality holds because $h(r(k)) \geq 3/4 \geq 2/3$.

A.2. Proof of Lemma 2

A.2.1. Expected number of work requests

By definition of γ, for a number of work requests $r \in \{0, \dots, p-1\}$, we have $\log_2(h(r)) \leq \frac{r}{-p\gamma}$ which implies that $h(r) \leq 2^{-r/(p\gamma)}$. Let $X_k = \phi(k)\prod_{i=0}^{k-1} 2^{r(i)/(p\gamma)}$. By Lemma 1, this shows that

$$\begin{aligned}\mathbb{E}\left[X_{k+1} \mid \mathcal{F}_k\right] &= \mathbb{E}\left[\phi(k+1) \mid \mathcal{F}_k\right]\prod_{i=0}^{k} 2^{r(i)/(p\gamma)} \\ &\leq \phi(k)2^{-r(k)/(p\gamma)}\prod_{i=0}^{k} 2^{r(i)/(p\gamma)} \\ &= X_k.\end{aligned}$$

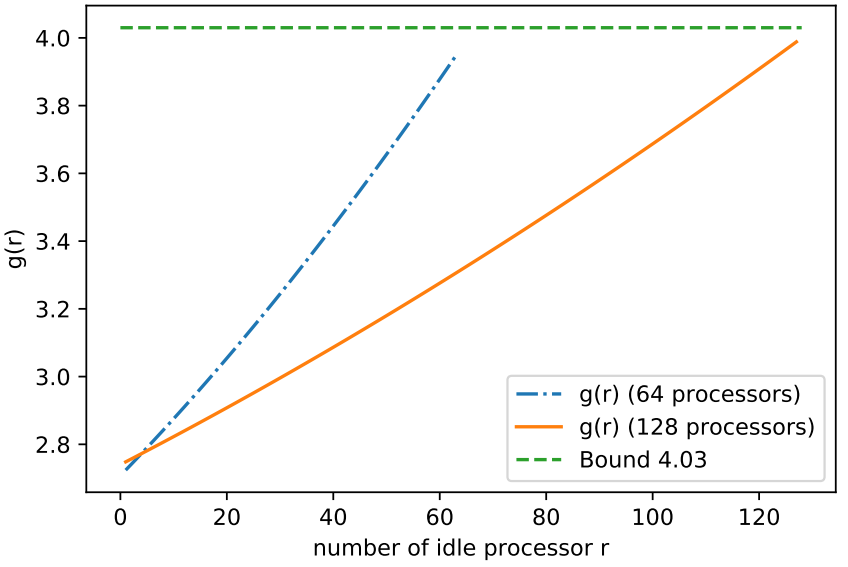


Figure 5: Function $g(r)$ as a function of the number of idle processor r for $p = \{64, 128\}$ processors. We observe that g is increasing and upper bounded by 4.03.

This shows that $(X_k)_k$ is a supermartingale for the filtration $\mathcal{F}$. As τ is a stopping time for the filtration $\mathcal{F}$, Doob's optional stopping theorem (see *e.g.*, [12, Theorem 4.1]) implies that

$$\mathbb{E}\left[X_\tau\right] \leq \mathbb{E}\left[X_0\right]. \tag{10}$$

By definition of X, we have $X_0 = \phi(0)$ and $X_\tau = \phi(\tau)2^{R/(p\gamma)}$. As $\phi(\tau) = 1$, this implies that

$$\mathbb{E}\left[2^{R/(p\gamma)}\right] = \mathbb{E}\left[\phi(\tau)2^{R/(p\gamma)}\right] \leq \phi(0). \tag{11}$$

By Jensen's inequality (see *e.g.*, [12, Equation (3.2)]), we have $\mathbb{E}[R/(p\gamma)] \leq \log_2\left(\mathbb{E}[2^{R/(p\gamma)}]\right)$. This shows that

$$\mathbb{E}\left[R\right] \leq p\gamma \log_2 \phi(0).$$

Moreover, by Markov's inequality, Equation (11) implies that for all $a > 0$:

$$\mathbb{P}\left[2^{R/(p\gamma)} \geq a\right] \leq \frac{\phi(0)}{a}.$$

By using $a = \phi(0)2^x$, this implies that $\mathbb{P}[R \geq p\gamma(\log_2 \phi(0) + x)] \leq 2^{-x}$.

A.2.2. Bound $\gamma < 4.03$

Let define the function g as

$$g(r) = \frac{r}{-p \log_2\left(\frac{3}{4} + \frac{1}{4}\left(\frac{p-2}{p-1}\right)^r\right)}. \tag{12}$$

By definition, the constant γ is the maximum of g:

$$\gamma = \max_{1 \leq r \leq p-1} g(r).$$

The function g is displayed in Figure 5 for the case of 64 processors and 128 processors. As we observe on this curve, g is increasing and is thus bounded by $g(p-1) < 4.03$. This is what we prove in the remainder of this proof.

Let $x = (p-2)/(p-1)$ and $y = x^r$, we have $y \in (0,1)$. We have

$$\frac{1}{g(r)} = -p\frac{\log_2(3/4 + y/4)}{\log_x(y)} = -\frac{p\log x}{\log 2} f(y),$$

where $f(y) = \log(3/4 + y/4)/\log(y)$. The first derivative of f is

$$f'(y) = \frac{y\log y - (3+y)\log(3/4 + y/4)}{y(3+y)(\log y)^2}.$$

The first derivative of the numerator of $f'(y)$ is $\log y - \log(3/4 + y/4)$ which is negative for $y < 1$. Thus it implies that $f'(y)$ is decreasing. As $f'(1) = 1$, this shows that $f'(y) \geq 0$ and therefore that f is increasing. This implies that $g(r)$ is increasing (because y is decreasing in r and $g(r) = \alpha/f(y)$ for $\alpha = -\log 2/(p\log x) > 0$).

As g is increasing, $\gamma = g(p-1)$. Now, for all $p \geq 2$:

$$\begin{aligned}\left(\frac{p-2}{p-1}\right)^{p-1} &= \left(1 - \frac{1}{p-1}\right)^{p-1} \\ &= \exp\left((p-1)\ln(1 - \frac{1}{p-1})\right) \\ &\leq \exp\left(-(p-1)\frac{1}{p-1}\right) = \frac{1}{e}.\end{aligned}$$

This shows that

$$\gamma = g(p-1) \leq \frac{1}{2 - \log_2\left(3 + \frac{1}{e}\right)} < 4.03.$$