Fault-tolerant parallel scheduling of arbitrary length jobs on a shared channel *

Marek Klonowski[1], Dariusz R. Kowalski[2,3], Jarosław Mirek[4], and Prudence W.H. Wong[4]

[1] Wrocław University of Science and Technology, Wrocław, Poland
Marek.Klonowski@pwr.edu.pl.
[2] School of Computer and Cyber Sciences, Augusta University, Augusta, USA
[3] SWPS University of Social Sciences and Humanities, Warsaw, Poland
dariusz.kowalski@swps.edu.pl
[4] University of Liverpool, Ashton Building, Ashton Street, Liverpool L69 3BX, UK
{J.Mirek, pwong}@liverpool.ac.uk

Abstract. We study the problem of scheduling n jobs on m identical, fault-prone machines f of which are prone to crashes by an adversary, where communication takes place via a multiple access channel without collision detection. Performance is measured by the total number of available machine steps during the whole execution (work). Our goal is to study the impact of preemption (i.e., interrupting the execution of a job and resuming it later by the same or different machine) and failures on the work performance of job processing. We identify features that determine the difficulty of the problem, and in particular, show that the complexity is asymptotically smaller when preemption is allowed.

1 Introduction

We examine the problem of performing n jobs by m machines reliably on a multiple access shared channel (MAC). This problem, originated by Chlebus et al. [6], has already been studied for unit length jobs, whereas this paper extends it by considering jobs with arbitrary lengths and studying the impact of features, such as preemption and the severity of failures, on the work performance.

The notion of preemption may be understood as the possibility of not performing a particular job in one attempt. This means that a single job may be interrupted partway through performing it and then resumed later by the same machine or even by another machine. Intuitively, the model without preemption is more general, yet both have subtleties that distinguish them noticeably.

Communication takes place on a shared channel, also known as multiple access channel, without collision detection. A message is transmitted successfully only when one machine transmits, and when more than one message is transmitted then it is no different from background noise (i.e., no transmission on the

* This work is supported by the Polish National Science Center (NCN) grant UMO-2017/25/B/ST6/02553, and by Networks Sciences & Techologies (NeST), School of EEECS, University of Liverpool.

channel). Therefore, we say that a job completion is confirmed only when the corresponding machine transmits such message successfully (before any failure it may suffer).

We consider the impact of an *adaptive f-bounded* adversary on the performance of algorithms for the problem. This kind of adversary may decide to crash up to f machines at any time of the execution. We use work as the effectiveness measure, i.e., the total number of machine steps available for computations. It is related (after dividing by m) to the average time performance of machines. Work may also correspond to energy consumption - an operational machine generates (consumes) a unit of work (energy) in every time step.

Previous and related results. Our work can be seen as an extension of the *Do-All* problem defined in a seminal work by Dwork, Halpern and Waarts [9]. This line of research was followed up by several other papers [3,4,5,8,10] which considered the message-passing model with point-to-point communication. In all these papers the authors assumed that performing a single job contributes a unit to work complexity. Paper [8] introduced a model, wherein the performance measure for *Do-All* solutions is extended to the *available machine steps* (i.e., including idle rounds). Authors in [8] developed an algorithm solving the problem with work $\mathcal{O}(n+(f+1)m)$ and message complexity $\mathcal{O}((f+1)m)$. We adopt this kind of work measurement in our paper.

The authors of [4] developed a deterministic algorithm with effort (i.e. sum of work and messages sent during the execution) $\mathcal{O}(n+m^a)$, for a specific constant $1<a<2$, against the unbounded adversary which may crash all but one machine. They presented the first algorithm for this type of adversary with both work and communication $o(n+m^2)$, where communication is understood as the total number of point-to-point messages sent during the execution. They also gave an algorithm achieving both work and communication $\mathcal{O}(n+m\log^2 m)$ against a strongly-adaptive linearly-bounded adversary.

In [12] there is an algorithm based on a gossiping protocol, solving the problem with work $\mathcal{O}(n+m^{1+\varepsilon})$, for any fixed constant ε. In [16] *Do-All* is studied for an asynchronous message-passing model, where executions are restricted in a way that the delay of each message is at most d. The authors proved $\Omega(n+md\log_d m)$ bound on the expected work. They also developed several algorithms - among them a deterministic one with work $\mathcal{O}((n+md)\log m)$. Further developments and comprehensive related literature can be found in the book by Georgiou and Shvartsman [13].

The line of research closest to ours is the *Do-All* problem of unit length jobs on a multiple access channel with no collision detection, which was first studied in [6], where the authors have shown a lower bound of $\Omega(n+m\sqrt{n})$ on work when there are no crashes, and $\Omega(n+m\sqrt{n}+m\min\{f,n\})$ in presence of crashes caused by an adaptive f-bounded adversary. They also proposed an algorithm, called Two-Lists, with optimal performance $\Theta(n+m\sqrt{n}+m\min\{f,n\})$. A recent paper [15] discusses different models of adversaries on a multiple access channel in the context of performing unit length jobs.

Extending the study of unit length jobs to arbitrary length jobs has taken place in a different context, c.f., [17], where one can find recent results and references to such extensions in the context of fault-tolerant centralized scheduling. There has been a long history of studying the preemptive vs non-preemptive model, e.g., [19,18,7], to which our work also contributes.

The problem we study finds application in other areas, e.g. in window scheduling. It is common in applications (like broadcast systems [1,14]) that requests need to be served periodically in the form of windows. Our problem could be applied to help selecting a window length based on controlling work (aka the number of available machine steps).

Our results. In this paper, we consider fault-tolerant scheduling of n arbitrary length jobs on m identical machines reliably over a multiple access channel. Our contributions are threefold, on: deterministic preemptive setting and deterministic non-preemptive setting. In the setting with job preemption, we prove a lower bound on work $\Omega(L+m\sqrt{L}+m\min\{f,L\}+m\alpha)$, where L is the sum of lengths of all jobs and α is the maximum length of a job, which holds for both deterministic and randomized algorithms against an adaptive f-*bounded* adversary. We design a corresponding deterministic distributed algorithm, called ScaTri, achieving work $\mathcal{O}(L + m\sqrt{L} + m\min\{f,L\} + m\alpha)$, which matches the lower bound asymptotically with respect to an overhead.

In the model without job preemption, we show a slightly higher lower bound on work $\Omega\left(L + \frac{L}{n}m\sqrt{n} + m\min\{f,n\} + m\alpha\right)$, implying a separation between the two settings: with and without preemption. Similarly as in the previous setting, it holds for both deterministic and randomized algorithms against an adaptive f-*bounded* adversary, thus showing that randomization does not help against an adaptive adversary regardless of the (non-)preemption setting. We develop a corresponding deterministic distributed algorithm, called DefTri, achieving work $\mathcal{O}(L + \alpha m\sqrt{n} + \alpha m\min\{f,n\})$. Results are discussed and compared in detail in Section 5.

2 Technical preliminaries

In this section we formally describe the model for the considered problem and also provide a high-level specification of a black-box procedure TaPeBB that we use in our algorithms. Additionally we present definitions of performing jobs and technicalities regarding preemption.

Machines. In our model we assume having m identical machines, with unique identifiers from the set $\{1,\ldots,m\}$. The distributed system of these machines is synchronized with a global clock, and time is divided into synchronous time slots, called *rounds*. All machines start simultaneously at a certain moment. Furthermore every machine may halt voluntarily. Note that a halted machine does not restart. Every operational machine generates a unit of work per round (even when it is idle), that has to be included while computing the complexity of an algorithm. In this paper by M we denote the number of operational, i.e., not crashed, machines. M may change during the execution.

Communication model. The communication channel for machines is the, widely considered in literature, *multiple access channel* [2,11], also called a *shared channel*, where a broadcasted message reaches every operational machine. We do not allow simultaneous transmissions of several messages, what means that we consider a channel without collision detection. Hence when more than one message is transmitted at a certain round, then machines hear a signal indifferent from background noise. A job is said to be *confirmed* if a machine, completing the job, successfully communicates this fact via the channel.

Adversarial model. Machines may fail by crashing, which happens because of an adversary activity. One of the factors describing the adversary is its power f, ie., the total number of failures that it may enforce. We assume that $0 \leq f \leq m-1$, thus at least one machine remains operational until an algorithm terminates. Machines that were crashed neither restart nor contribute to work.

We consider two adversarial models. An *adaptive f-bounded adversary* can observe the execution and make decisions about up to f crashes arbitrarily, at any moment of the execution. A *non-adaptive f-bounded adversary*, in addition to choosing the faulty subset of f machines, has to declare prior the execution in which rounds crashes will occur, i.e., for every machine declared as faulty, there must be a corresponding round number in which the fault will take place.

It is worth noticing that in the context of deterministic algorithms such an adversary is consistent with the adaptive adversary that may decide online which machines will be crashed at any time, as the algorithm may be simulated by the adversary in advance, before its execution, providing knowledge about the best decisions to be taken. Finally, an $(m-1)$-bounded adversary is also called an *unbounded* adversary.

Complexity measure. The complexity measure to be used in our analysis is, as mentioned before, *work*, also called the *total number of available machine steps*. It is the number of machine steps available for computations. This means that each operational machine that did not halt contributes a unit of work in each round, even if it stays idle.

Precisely, assume that an execution starts when all the machines begin simultaneously in some fixed round r_0. Let r_v be the round when machine v halts (or is crashed). Then its work contribution is equal to $r_v - r_0$. Consequently the algorithm complexity is the sum of such expressions over all machines i.e.: $\sum_{1 \leq v \leq m}(r_v - r_0)$.

Jobs and reliability. We assume that the list of jobs is known to all machines and we expect that machines perform all n jobs as a result of executing an algorithm. We assume that *jobs have arbitrary lengths*, are *independent* (may be performed in any order) and *idempotent* (may be performed many times, even concurrently by different machines).

Furthermore a *reliable* algorithm satisfies the following conditions in any execution: *all jobs are eventually performed, if at least one machine remains non-faulty* and *each machine eventually halts, unless it has crashed.*

We assume that jobs have some minimal (atomic or unit) length. Consequently, we can also assume that each job's length is a multiple of the minimal

length. As the model that we consider is synchronous, this minimal length may be justified by the round duration required for local computations for each machine. By ℓ_a we denote the length of job a. We also use L to denote the sum of lengths of all jobs, i.e., $L = \sum_i \ell_i$. Finally, by α we denote the length of the longest job.

Preemptive vs Non-Preemptive Model. By the means of *preemption* we define the possibility of performing jobs in several pieces. Precisely, consider job a, of length ℓ_a (for simplicity we assume that ℓ_a is even) and machine v is scheduled to perform job a at some time of the algorithm execution. Assume that v performs $\ell_a/2$ units of job a and then reports such progress. When preemption is available the remaining $\ell_a/2$ units of job a may be performed by some machine w where $w \neq v$.

Length ℓ_a of job a means that job a requires ℓ_a rounds to be fully performed. Such view allows to think that job a is a chain of ℓ_a *tasks*. Hence we conclude that all jobs form a set of chains of unit length tasks.

We further denote by a_k task k of job a. However, when we refer to a single job, disregarding its tasks, we refer to it simply as job a.

We define two types of jobs regarding how intermediate progress is handled:

Oblivious job — it is sufficient to have knowledge that previous tasks were done, in order to perform remaining tasks from the same job. In other words, any information from in between progress does not have to be announced.

Non-Oblivious job — any intermediate progress needs to be reported through the channel when interrupting the job to resume it later, and possibly pass the job to another machine.

In the preemptive oblivious model a job may be abandoned by machine v on some task k without confirming progress up to this point on the channel and then continued from the same task k by any machine w.

As an example of the preemptive oblivious model, consider a scenario that there is a job that a shared array of length x needs to be erased. If a machine stopped performing this job at some point, another one may reclaim that job without the necessity of repeating previous steps by simply reading to which point it has been erased.

For the preemptive non-oblivious model, consider that a machine executes Dijkstra's algorithm for finding the shortest path. If it becomes interrupted, then another machine cannot reclaim this job otherwise than by performing the job from the beginning, unless intermediate computations have been shared. In other words preemption is available with respect to maintaining information about tasks.

On the contrary to the preemptive model, we also consider the model without preemption i.e. where each job can be performed by a machine only in one piece - when a machine is crashed while performing such job, the whole progress is lost, even if it was reported on the channel before the crash took place.

The Task-Performing Black Box (**TaPeBB*****).*** The algorithms we design in this paper employ a black-box procedure for arbitrary length jobs that is able to

reliably perform a subset of input (in the form of jobs consisting of chains of consecutive tasks) or report that something went wrong. In what follows, we specify this procedure, called the *Task-Performing Black Box* (TaPeBB for short), and argue that it can be implemented and employed to our considerations.

We can use the procedure in both deterministic and randomized solutions. Precisely, all our algorithms use TaPeBB, despite the fact that they perform differently in the sense of work complexity.

Most important ideas of our results lie within how to preprocess the input rather than how to actually perform the jobs, thus employing such a black-box could improve the clarity of presentation.

General properties of TaPeBB. A synchronous system is characterized by time being divided into synchronous slots, that we already called rounds. In what follows, each round is a possibility for machines to transmit.

The nature of arbitrary length jobs leads, however, to a concept that the time between consecutive broadcasts needs to be adjusted. Specifically, for sets containing long jobs in the non-preemptive configuration of the problem, it may be better to broadcast the fact of performing the job fully, rather than semi-broadcasting multiple times. Only the final transmission brings valuable information about progress in performing jobs and any intermediate transmissions congest the channel, indicating only that the broadcasting machine is still operational.

Therefore, we assume that TaPeBB has a feature of changing the duration between consecutive broadcasts. We will call the actual time step between consecutive broadcasts a *phase*. Denote the length of a phase by ϕ. Unless stated otherwise, we assume that $\phi = 1$, i.e., the duration of a phase and a round is consistent, thus machines may transmit in any round.

Input: `TaPeBB`$(v, d,$ *`JOBS`*, *`MACHINES`*$, \phi)$

– v represents the id of a machine executing TaPeBB.

– TaPeBB takes a list of machines `MACHINES` and a list of jobs `JOBS` as the input, yet from the task perspective, i.e., jobs are provided as a chain of tasks. All necessary information about tasks is available through list `JOBS`, including their id's, and how, as well as, in what order they build jobs. It may happen that a job is not done fully and only some initial segment of tasks forming that job is performed. Because list `JOBS` maintains information about tasks, such a situation can be successfully handled.

– Additionally, the procedure takes an integer value d. It specifies the number of machines that will be used in the procedure for performing provided job input. The procedure works in such a way that each working machine is responsible for performing a number of jobs. For clarity we assume that TaPeBB always uses the initial d machines from the list of machines. Using a certain number of machines in the procedure allows us to set an upper bound on the amount of work accrued during a single execution.

– We call a single execution an *epoch* and the parameter d is called the length of an epoch (i.e., the number of phases that form an epoch).

– ϕ is the length of a phase i.e. the duration between consecutive broadcasts.

Output: what jobs/tasks have been done and which machines have crashed.
– Having explained what is the length of an epoch in TaPeBB we now describe the capability of performing tasks in a single epoch, which is understood as the maximal number of jobs that may be confirmed in an epoch, when it is executed fully without any adversarial distractions. Firstly, let us note, that TaPeBB allows to confirm j tasks in some round j. This comes from the fact that if a machine worked for j rounds and was able to perform one task per round, then it can confirm at most j tasks when it comes to broadcasting in round j. Therefore, the capability of performing tasks in an epoch is at most $\sum_{j=1}^{d} j$ which is the sum of an arithmetic series with common difference equal 1 over all rounds of an epoch.
– As a result of running a single epoch we have an output information about which tasks were actually done and whether there were any machines identified as crashed, when machines were communicating progress (a crash is consistent with a machine being silent when scheduled to broadcast).

A candidate algorithm to serve as TaPeBB is the Two-Lists algorithm from [6] and we refer readers to the details therein. Nevertheless, we assume that it may be substituted with an arbitrary algorithm fulfilling the requirements that we stated above.

3 Preemptive model

In this section we consider the scheduling problem in the model with preemption, which is, intuitively, an easier model to tackle. We show a lower bound for oblivious jobs (Section 3.1) and then present the algorithm for non-oblivious jobs (Section 3.2).

3.1 Lower bound

We first recall the minimal work complexity introduced in [6].

Lemma 1 ([6], Lemma 2). *A reliable, distributed algorithm, possibly randomized, performs work $\Omega(n + m\sqrt{n})$ in an execution in which no failures occur.*

Recall that L denotes the total length of jobs and α denotes the length of the longest job. As jobs are built with unit length tasks, we can look at this result from the task perspective. Precisely, as L can be considered to represent the number of tasks needed to be performed by the system, then the lower bound for our model translates in a straightforward way. Furthermore, in our model there is a certain bottleneck dictated by the longest job. Reason being is that there may be an execution with one long job, in comparison to others being short. Thus the longest job determines the magnitude of work in the complexity formula.

We conclude with the following theorem, setting the lower bound for the considered problem.

Theorem 1. *The adaptive f-bounded adversary, for $0 \leq f < m$, can force any reliable, possibly randomized and centralized algorithm and a set of oblivious jobs of arbitrary lengths with preemption, to perform work $\Omega(L + m\sqrt{L} + m\min\{f, L\} + m\alpha)$.*

3.2 Algorithm ScaTri

In this section we present our algorithm Scaling-Triangle MAC Scheduling (ScaTri for short). In ScaTri machines have access to all the jobs and the corresponding tasks that build those jobs. This means that they know their id's and lengths.

We assume that each machine maintains three lists: `MACHINES`, `TASKS` and `JOBS`. The first list is a list of operational machines and is updated according to the information broadcasted through the communication channel. If there is information that some machine was recognized as crashed, then this machine is removed from the list. In the context of the TaPeBB procedure, recall that this is realized as the output information. TaPeBB returns the list of crashed machines so that operational machines may update their lists.

List `TASKS` represents all the tasks that are initially computed from the list of jobs. Every task has its unique identifier, which allows to discover to which job it belongs and its position in that job. This allows to preserve consistency and coherency: task k cannot be performed before task $k-1$. If some tasks are performed then this fact is also updated on the list.

List `JOBS` represents the set of jobs — it is a convenient way to know what are the consecutive parts of input for the TaPeBB procedure. Jobs are assigned to each machine at the beginning of an epoch (TaPeBB execution). We assume that machines have instant access to jobs lengths. Information on list `JOBS` may be updated directly from information maintained on list `TASKS`. To ease the discussion, by $|\texttt{XYZ}|$ we denote the length of list XYZ and by $M = |\texttt{MACHINES}|$ the actual number of operational machines.

The capability of performing tasks in a TaPeBB epoch is at most $\sum_{j=1}^{d} j$ (cf. Section 2) which is the sum of an arithmetic series with common difference equal 1 over all rounds of an epoch. If we take a geometric approach and draw lines or boxes of increasing lengths one next to the other, then drawing this sequence would form a triangle. Hence, providing a subset of jobs as the input is consistent with packing them into such a triangle, cf., Figure 1.

In TaPeBB executed with $\phi = 1$ broadcasts take place in each round. This means that in round 1 machine 1 is scheduled to broadcast, in round 2 machine 2 and, in general, in round j machine j is scheduled to broadcast. Consequently, j tasks can be confirmed as performed in round j, unless machine j is crashed and hence silence is heard in round j. Therefore, the capability of performing tasks by TaPeBB is referred to as the triangle.

We assume d describes the current length of an epoch. Initially it is set to m, but it may be reduced while the algorithm is running, cf. Figure 1 (b), (c). In what follows, let us assume that the length of the epoch d is set to $m/2^i$ for some i. We need to fill in a triangle of size $\sum_{j=1}^{\frac{m}{2^i}} j$. Initially we need to provide

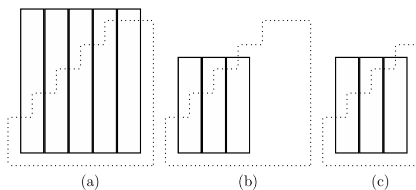

Fig. 1. A very general idea about how ScaTri works. The vertical stripes represent jobs and the dotted-line triangle is the capability of the algorithm to perform jobs in a single epoch (or parts of them i.e. some consecutive tasks). Consequently, if there are enough jobs to pack into an epoch, then it is executed (a). Otherwise (b), the length of the epoch is reduced (c). This helps preventing excessive idle work.

$m/2^i$ jobs, that will form the base of the triangle. Jobs are provided as the input in such a way that the shortest ones are preferred for machines with lower id's. If there are several jobs with same lengths, then those with lower id's are preferred, cf. Figure 2. After having the base filled, it is necessary to look whether there are any gaps in the triangle, see Figure 2 (b). If so, another layer of jobs is placed on top of the base layer, and the procedure is repeated, see Figure 2 (c), (d). Otherwise, we are done and ready to execute TaPeBB.

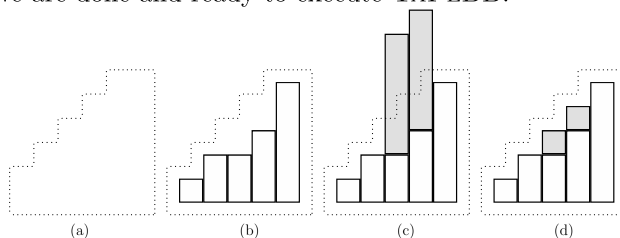

Fig. 2. An illustration of job input. (a) Initially there is a certain capability of TaPeBB (triangle size) for performing jobs. (b) The initial layer is filled with jobs (white blocks), yet there are still some gaps. (c) An additional layer (gray blocks) is placed to fill in the triangle entirely. (d) In fact those additional jobs will only be done partially.

This approach allows to "trim" longer jobs preferably — these will have more tasks completed after executing TaPeBB than shorter ones, because they are scheduled to be done by machines with higher id's, which are broadcasting in further rounds. As machines with higher id's are broadcasting in further rounds, then they can confirm more tasks with a single broadcast.

One can observe that performing a transmission is an opportunity to confirm on the channel that the tasks that were assigned to a machine are done. Additionally this confirms that a certain machine is still operational. In what follows these two types of aggregated pieces of information: which jobs were done, and which machines were crashed, are eventually provided as the output of TaPeBB.

When $d = m/2^i$ for some $i = 1, \ldots, \log m$ it may happen that there are not enough tasks (jobs) to fill in a maximal triangle. If this happens we will, in

some cases, reduce the job-schedule triangle (and simultaneously the length of the epoch) to $m/2^{i+1}$ and try to fill in a smaller triangle, cf., Figure 1 (b), (c).

Finally, we use TaPeBB$(v, d, \texttt{JOBS}, \texttt{MACHINES}, \phi)$ to denote which machine executes the procedure, the size of the schedule and the length of the epoch, the list of jobs from which the input will be provided, the list of operational machines, and the phase duration. We emphasize that we described the process of assigning jobs to machines from the algorithm perspective, i.e., we illustrated that the system needs to provide input to TaPeBB. Nevertheless, for the sake of clarity we assume that TaPeBB collects the appropriate input by itself according to the rules described above, after having lists `JOBS` and `MACHINES` provided as the input. Figures 1 and 2 illustrate the idea standing behind ScaTri.

Algorithm 1: ScaTri; code for machine v

```
1  - initialize MACHINES to a sorted list of all m names of machines;
2  - initialize JOBS to a sorted list of all n names of jobs;
3  - initialize TASKS to a sorted list of all tasks according to the information from
     JOBS;
4  - initialize variable d representing the length of an epoch;
5  - initialize variable φ := 1 representing the length of a phase;
6  - initialize i := 0;
7  repeat
8      d := m/2^i;
9      if |TASKS| ≥ d(d + 1)/2 then
10         execute TaPeBB(v, d, JOBS, MACHINES, φ);
11         update JOBS, MACHINES, TASKS according to the output information from
             TaPeBB;
12     end
13     else
14         i := i + 1;
15     end
16 until |JOBS| = 0;
```

The following theorem summarizes work performance of ScaTri:

Theorem 2. ScaTri *performs work* $\mathcal{O}(L + m\sqrt{L} + m\min\{f, L\} + m\alpha)$ *against the adaptive adversary and a set of non-oblivious, arbitrary length jobs with preemption.*

As mentioned in the beginning of Section 3, the lower bound here is proved for oblivious jobs in the preemptive model, while the algorithm works reliably for non-oblivious jobs in the same model, because TaPeBB procedure provides any intermediate job performing progress as the output information and all the information is updated sequentially after each epoch. This implies that the distinction between oblivious and non-oblivious jobs in the preemptive model and against an adaptive adversary does not matter, thus we finish this section with the following corollary:

Corollary 1. ScaTri *is optimal in asymptotic work efficiency for jobs with arbitrary lengths with preemption for both oblivious and non-oblivious jobs.*

4 Non-preemptive model

In this section we consider the complementary problem of performing jobs when preemption is not available. We begin with the lower bound and then proceed to the algorithm description.

4.1 Lower bound

In the non-preemptive model each job can only be performed by a machine in one piece. This reflects an all-or-nothing policy, i.e., a machine cannot make any intermediate progress while performing a job. If it starts working on a job then either it must be done entirely or it is abandoned without any tasks performed.

In what follows any intermediate broadcasts while performing a job are not helpful for the system. The only meaningful transmissions are those which allow to confirm certain jobs being done.

Theorem 3. *A reliable, distributed algorithm, possibly randomized, performs work* $\Omega\left(L + \frac{L}{n}m\sqrt{n} + m\min\{f,n\} + m\alpha\right)$ *in an execution with at most* f *failures against an adaptive adversary.*

4.2 Algorithm DefTri

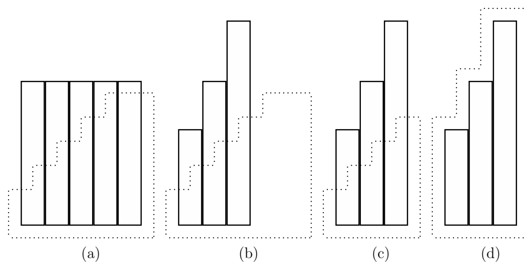

Fig. 3. Most important features of DefTri. Similarly to ScaTri we apply the method of reducing idle work by reducing the input size (epoch length) for the task performing procedure ((a), (b), (c)). Additionally we change the duration on phases (time between consecutive broadcasts) in order to be able to fill in jobs entirely - in the model without preemption jobs cannot be done partially (d).

The key reason to consider the notion of a phase (i.e. time between consecutive broadcasts), introduced in the model section, is that it allows to assume broadcasts are done after jobs are done entirely. The only question is how to set the phase parameter appropriately.

We analyze our algorithm from the average length of the current set of jobs perspective. To justify this setting let us consider two scenarios. For the first one, when the phase parameter is set to 1, which is consistent with transmissions taking place each round we already mentioned above that most of the transmissions have no effect on progressing with performing jobs. However, setting the phase parameter to the length of the longest job α can generate excessive idle work.

In what follows, setting the phase parameter to the average length of the current set of jobs allows us to estimate the number of jobs that can be performed in a certain period. What is more it proves that this setting always allows to schedule a significant number of jobs to be done, while preventing from excessive idle work. Hence we begin with a simple fact showing that the number of jobs with length twice the average does not exceed half of the total number of jobs.

Fact 1 *Let* n *be the number of jobs,* L *be the sum of all the lengths of jobs and let* $\frac{L}{n}$ *represent the average length of a job. Then we have that* $\left|\left\{a : \ell_a > 2\frac{L}{n}\right\}\right| \le \frac{n}{2}$.

Algorithm 2: DefTri; pseudocode for machine v

```
1  - initialize MACHINES to a sorted list of all m names of machines;
2  - initialize JOBS to a sorted list of all n names of jobs;
3  - initialize TASKS to a sorted list of all tasks;
4  - initialize variable d representing the length of an epoch;
5  - initialize variable φ representing the length of a phase;
6  - initialize i := 0;
7  repeat
8      d := m/2^i;
9      if |JOBS| ≥ d(d + 1)/2 then
10         φ := |TASKS|/|JOBS| // set φ to current average length of a job
11         execute TaPeBB(v, d, JOBS, MACHINES, φ);
12         update JOBS, MACHINES, TASKS according to TaPeBB output;
13     end
14     else
15         i := i + 1;
16     end
17 until |JOBS| = 0;
```

Algorithm Deforming-Triangle MAC Scheduling (DefTri for short) is similar to ScaTri introduced in the previous section and on the top of its design there is the TaPeBB procedure for performing jobs. Roughly speaking, the algorithm, repetitively, tries to choose jobs that could be packed into a specific triangle (parameters of which are controlled by the algorithm) and feed them to TaPeBB. Furthermore, once again the main goal of the algorithm is to avoid redundant or idle work. Hence, the main feature is to examine whether there is an appropriate number of jobs to fill in an epoch of TaPeBB. However, as we are dealing with the non-preemptive model, jobs cannot be done in pieces.

Hence, apart from checking the number of jobs (and possibly reducing the size of an epoch) the algorithm also changes the phase parameter, i.e., the duration between consecutive broadcasts — this is somehow convenient in order to know how the input jobs should be fed into TaPeBB. It settles a "framework" (epoch) with "pumped rounds" that should be filled in appropriately.

Using Fact 1 assures that if the size of an epoch is reduced accordingly to the number of jobs and the phase parameter is set accordingly to the actual average length of the current set of jobs (i.e., |TASKS|/JOBS| in the code of Algorithm 2), then we are able to provide the input for TaPeBB appropriately and expect

that at least $\sum_{j=1}^{\frac{m}{2^i}} j$ jobs will be performed for some size parameter i as a result of executing TAPEBB, yet without taking crashes into account.

Figure 3 illustrates the idea of DEFTRI.

Theorem 4. DEFTRI *performs work* $\mathcal{O}(L+\alpha m\sqrt{n}+\alpha m\min\{f,n\})$ *against the adaptive* f*-bounded adversary.*

5 Comparison of results for the two models

In this section we compare the upper bound for preemptive scheduling from Theorem 2 with the lower bound in the model without preemption from Theorem 3. This comparison shows the range of dependencies between the model parameters for which both models are separated, i.e., the upper bound in the model with preemption is asymptotically smaller than the lower bound in the model without preemption. We settle the scope on the, intuitively greater, bound for the model without preemption and examine how the ranges of the parameters influence the magnitude of the formulas. Let us recall both formulas:

Preemptive, upper bound: $\mathcal{O}(L + m\sqrt{L} + m\min\{f, L\} + m\alpha)$;
Non-preemptive, lower bound: $\Omega(L + \frac{L}{n}m\sqrt{n} + \min\{f, n\} + m\alpha)$.

If L is the factor that dominates the bound in the non-preemptive model, then by simple comparison to other factors we have that L also dominates the bound in the preemptive model. In what follows both formulas are asymptotically equal when the total number of tasks is appropriately large.

When $\frac{L}{n}m\sqrt{n}$ dominates the non-preemptive formula, then by a simple observation we have that $1 \leq \sqrt{\frac{L}{n}}$, because the number of jobs is greater than the number of tasks, and applying a square root does not affect the inequality. Multiplying both sides of the inequality by $m\sqrt{L}$ gives $m\sqrt{L} \leq \frac{L}{n}m\sqrt{n}$. Thus, if $\frac{L}{n}m\sqrt{n}$ dominates the bound in the non-preemptive formula then the non-preemptive formula is asymptotically greater than the preemptive one.

On the other hand when L dominates the bound then they are asymptotically equal, as both formulas linearize for such magnitude. These results confirm that in the channel without collision detection the non-preemptive model is more demanding for most settings of parameters.

6 Conclusions

We addressed the problem of performing jobs of arbitrary lengths on a multiple-access channel without collision detection. Specifically, we analysed two scenarios. In the preemptive model, we showed a lower bound for the considered problem that is $\Omega(L + m\sqrt{L} + m\min\{f, L\} + m\alpha)$ and designed an algorithm that meets the proved bound, hence settled the problem. We also answered the question of how to deal with jobs without preemption, for which we showed a lower

bound $\Omega(L+\frac{L}{n}m\sqrt{n}+m\min\{f,n\}+m\alpha)$ and developed a solution, basing on the one for preemptive jobs which work complexity is $\mathcal{O}(L+\frac{L}{n}m\sqrt{n}+\alpha m\min\{f,n\})$.

Considering open directions for research considered in this paper, it is natural to address the question of channels with collision detection. Furthermore it is worth considering whether randomization could help improving the results, as it took place in similar papers considering the Do-All problem on a shared channel [15]. We conjecture that the use of randomization against non-adaptive adversaries leads to a solution with expected work $\mathcal{O}(L+m\sqrt{L}+m\alpha)$, and thus could prove that randomization helps against the non-adaptive adversary.

Finally, the primary goal of this work was to translate scheduling from classic models to the model of a shared channel, in which it was not considered in depth; therefore, a natural open direction is to extend the model further with other features considered in scheduling literature.

References

1. Amotz Bar-Noy, Joseph Naor, and Baruch Schieber. Pushing dependent data in clients-providers-servers systems. *Wireless Networks*, 9(5):421–430, 2003.
2. Bogdan S Chlebus. *Randomized communication in radio networks, a chapter. In: Pardalos, P.M., Rajasekaran, S., Reif, J.H., Rolim, J.D.P. (eds.), Handbook on Randomized Computing.*, volume 1. Kluwer Academic Publisher, 2001.
3. Bogdan S. Chlebus, Roberto De Prisco, and Alex A. Shvartsman. Performing tasks on synchronous restartable message-passing processors. *Distrib. Comput.*, 14(1):49–64, January 2001.
4. Bogdan S. Chlebus, Leszek Gasieniec, Dariusz R. Kowalski, and Alexander A. Shvartsman. Bounding work and communication in robust cooperative computation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 295–310, London, UK, UK, 2002. Springer-Verlag.
5. Bogdan S. Chlebus and Dariusz R. Kowalski. Randomization helps to perform independent tasks reliably. *Random Structures and Algorithms*, 24(1):11–41, 2004.
6. Bogdan S. Chlebus, Dariusz R. Kowalski, and Andrzej Lingas. Performing work in broadcast networks. *Distributed Computing*, 18(6):435–451, 2006.
7. E. G. Coffman, Jr. and M. R. Garey. Proof of the 4/3 conjecture for preemptive vs. nonpreemptive two-processor scheduling. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 241–248, New York, NY, USA, 1991. ACM.
8. Roberto De Prisco, Alain Mayer, and Moti Yung. Time-optimal message-efficient work performance in the presence of faults. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 161–172, New York, NY, USA, 1994. ACM.
9. Cynthia Dwork, Joseph Y. Halpern, and Orli Waarts. Performing work efficiently in the presence of faults. *SIAM J. Comput.*, 27(5):1457–1491, 1998.
10. Z. Galil, A. Mayer, and Moti Yung. Resolving message complexity of byzantine agreement and beyond. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, page 724, Washington, DC, USA, 1995. IEEE Computer Society.
11. Robert G. Gallager. A perspective on multiaccess channels. *IEEE Trans. Information Theory*, 31:124–142, 1985.

12. Chryssis Georgiou, Dariusz R. Kowalski, and Alexander A. Shvartsman. Efficient gossip and robust distributed computation. *Theor. Comput. Sci.*, 347(1-2):130–166, November 2005.
13. Chryssis Georgiou and Alexander A. Shvartsman. *Cooperative Task-Oriented Computing: Algorithms and Complexity*. Morgan & Claypool Publishers, 2011.
14. Veena Gondhalekar, Ravi Jain, and John Werth. Scheduling on Airdisks: Efficient access to personalized information services via periodic wireless data broadcast. Technical Report TR-96-25, Department of Computer Science, University of Texas, Austin, TX, 1996.
15. Marek Klonowski, Dariusz R. Kowalski, and Jarosław Mirek. Ordered and delayed adversaries and how to work against them on a shared channel. *Distributed Computing*, Sep 2018.
16. Dariusz R. Kowalski and Alex A. Shvartsman. Performing work with asynchronous processors: Message-delay-sensitive bounds. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 265–274, New York, NY, USA, 2003. ACM.
17. Dariusz R. Kowalski, Prudence W. Wong, and Elli Zavou. Fault tolerant scheduling of tasks of two sizes under resource augmentation. *J. of Scheduling*, 20(6):695–711, December 2017.
18. Giorgio Lucarelli, Abhinav Srivastav, and Denis Trystram. From preemptive to non-preemptive scheduling using rejections. In Thang N. Dinh and My T. Thai, editors, *Computing and Combinatorics*, pages 510–519, Cham, 2016. Springer International Publishing.
19. Robert McNaughton. Scheduling with deadlines and loss functions. *Manage. Sci.*, 6(1):1–12, October 1959.