7

Asynchrony and Delay-Sensitive Bounds

COMMON impediments to effective coordination in distributed settings, as we have seen, include failures and asynchrony that manifest themselves, e.g., in disparate processor speeds and varying message latency. Fortunately, the *Do-All* problem can always be solved as long as at least one processor continues to make progress. In particular, assuming that initially there n tasks that need to be performed, and the tasks are known to all p processors, the problem can be solved by a communication-oblivious algorithm where each processor performs all tasks. Such a solution has total-work $S = O(n \cdot p)$, and either it requires no communication, or it cannot rely on communication because of very long delays. On the other hand, $\Omega(n)$ is the obvious lower bound on work; additionally we show in this chapter that a lower bound of $S = \Omega(n + p \log p)$ holds for any asynchronous algorithm for *Do-All*, no matter how small is the message delay. Therefore it is reasonable to have the goal that, given a non-trivial and non-negligible delay d, effective use of messaging should result in the decrease in work from the trivial upper bound of $S = O(n \cdot p)$ so that work becomes sub-quadratic in n and p.

Obtaining algorithmic efficiency in asynchronous models of computation is difficult. For an algorithm to be interesting, it must be better than the oblivious algorithm, in particular, it must have sub-quadratic work complexity. However, if messages can be delayed for a "long time", then the processors cannot coordinate their activities, leading to an immediate lower bound on work of $\Omega(n \cdot p)$. In particular, it is sufficient for messages to be delayed by $\Theta(n)$ time for this lower bound to hold. Algorithmic techniques for synchronous processors assume constant-time message delay. In general it is not clear how such algorithms can be adapted to deal with asynchrony. Thus it is interesting to develop algorithms that are correct for any pattern of asynchrony and failures (with at least one surviving processor), and whose work depends on the message latency upper bound, such that work increases gracefully as the latency grows. The quality of the algorithms can be assessed by comparing their work to the corresponding *delay-sensitive* lower bounds.

In this chapter our goal is to obtain complexity bounds for work-efficient message-passing algorithms for the *Do-All* problem. We require that the algorithms tolerate any pattern of processor crashes with at least one surviving processor. More significantly, we are interested in algorithms whose work degrades gracefully as a function of the worst case message delay d. Here the requirement is that work must be subquadratic in n and p as long as $d = o(n)$. Thus for our algorithms we aim to develop *delay-sensitive* analysis of work and message complexity. Noting again that work must be $\Omega(p \cdot n)$ for $d \geq n$, we give a comprehensive analysis for $d < n$, achieving substantially better work complexity.

Chapter structure.

We define the model of adversity and expand on complexity measures in Section 7.1. In Section 7.2 we develop a delay-sensitive lower bounds for *Do-All*. In Section 7.3 we deal with permutations and their combinatorial properties used in the algorithm analysis. In Section 7.4 we present and analyze a work-efficient asynchronous deterministic *Do-All* algorithm. In Section 7.5 we present and analyze two randomized and one deterministic algorithm that satisfy our efficiency criteria. We discuss open problems in Section 7.6.

7.1 Adversarial Model and Complexity

Processors communicate over a fully connected network by sending point-to-point messages via reliable asynchronous channels. When a processor sends a message to a group of processors, we call it a multicast message, however in the analysis we treat a multicast message as multiple point-to-point messages. Messages are subject to delays, but are not corrupted or lost.

We assume an omniscient (on-line) adversary that introduces delays. We call this adversary $\mathcal{A}_D$. The adversary can introduce arbitrary delays between local processor steps and cause processor crashes (crashes can be viewed as infinite delays). The only restriction is that at least one processor is non-faulty. Adversary $\mathcal{A}_D$ also causes arbitrary message delays.

We specialize adversary $\mathcal{A}_D$ by imposing a constraint on message delays. We assume the existence of a global real-timed clock that is unknown to the processors. For convenience we measure time in terms of units that represent the smallest possible time between consecutive clock-ticks of any processor. We define the delay-constrained adversary as follows. We assume that there exists an integer parameter d, that is not assumed to be a constant and that is *unknown* to the processors, such that messages are delayed by at most d time units. We call this adversary $\mathcal{A}_D^{(d)}$. It is easy to see that $\mathcal{A}_D^{(d)} \subseteq \mathcal{A}_D^{(d+1)}$ for any $d \geq 0$, because increasing the maximum delays introduces new adversarial behaviors. We also note that $\mathcal{A}_D = \bigcup_{d \in \mathbb{N}} \mathcal{A}_D^{(d)}$.

In this chapter we are interested in algorithms that are correct against adversary $\mathcal{A}_D$, i.e., for any message delays. For the purpose of analysis of such algorithms, we are interested in complexity analysis under adversary $\mathcal{A}_D^{(d)}$, for some specific positive d that is unknown to the algorithm. Note that by the choice of the time units, a processor can take at most d local steps during any global time period of duration d.

For an algorithm A, let $\mathcal{E} = \mathcal{E}(A, \mathcal{A}_D^{(d)})$ be the set of all executions of the algorithm in our model of computation subject to adversary $\mathcal{A}_D^{(d)}$. For the purposes of this chapter, we define the *weight* of an adversarial pattern to be the *maximum delay incurred by any message*. Thus, for any execution $\xi \in \mathcal{E}$, the maximum weight of the adversarial pattern $\xi|_{\mathcal{A}_D^{(d)}}$ is d, that is $||\xi|_{\mathcal{A}_D^{(d)}}|| \leq d$.

We assess the efficiency of algorithms in terms of total-work (Definition 2.4) and message complexity (Definition 2.6) under adversary $\mathcal{A}_D^{(d)}$. We use the notation $S(n, p, d)$ to denote work, and $M(n, p, d)$ to denote message complexity. Expected work and message complexity are denoted by $ES(n, p, d)$ and $EM(n, p, d)$ respectively.

When work or messages complexities do not depend d we omit d and use, for example, $S(n, p)$ and $M(n, p)$ for work and message complexity (and $ES(n, p)$ and $EM(n, p)$ for expected work and message complexity).

Next we formulate a proposition leading us to not consider algorithms where a processor may halt voluntarily before learning that all tasks have been performed.

Proposition 7.1. *Let Alg be a Do-All algorithm such that there is some execution ξ of Alg in which there is a processor that (voluntarily) halts before it learns that all tasks have been performed. Then there is an execution ξ' of Alg with unbounded work in which some task is never performed.*

Proof. For the proof we assume a stronger model of computation where in one local step any processor can learn the complete state of another processor, including, in particular, the complete computation history of the other processor. Assume that, in some execution ξ, the *Do-All* problem is solved, but some processor i halts in ξ without learning the a certain task z was performed. First we observe that for any other processor j that i learns about in ξ, j does not perform task z by the time i learns j's state. (Otherwise i would know that z was performed.) We construct another execution ξ' from ξ as follows. Any processor j (except for i) proceeds as in ξ until it attempts to perform task z. Then j is delayed forever. We show that processor i can proceed exactly as in ξ. We claim that i is not able to distinguish between ξ and ξ'. Consider the histories of all processors that i learned about in ξ' (directly or indirectly). None of the histories contain information about task z being performed. Thus the history of any processor j was recorded in advance of j's delay in ξ'. Then by the definition of ξ' these histories are identical to those in ξ. This means that in ξ' processor i halts as in ξ. Since the problem

remains unsolved, processor i continues to be charged for each local clock tick (recall that work is charged until the problem is solved). □

As the result of Proposition 7.1, we will only consider algorithms where a processor may voluntarily halt only after it knows that all tasks are complete, i.e., for each task the processor has local knowledge that either it performed the task or that some other processor did.

Note that for large message delays the work of any *Do-All* algorithm is necessarily $\Omega(n \cdot p)$. The following proposition formalizes this lower bound and motivates our delay-sensitive approach.

Proposition 7.2. *Any algorithm that solves the Do-All problem in the presence of adversary $\mathcal{A}_D^{(c \cdot n)}$, for a constant $c > 0$, has work $S(n, p) = \Omega(n \cdot p)$.*

Proof. We choose the adversary that delays each message by $c \cdot n$ time units, and does not delay any processor. If a processor halts voluntarily before learning that all tasks are complete, then by Proposition 7.1 work may be unbounded. Assume then that no processor halts voluntarily until it learns that all tasks are done. A processor may learn this either by performing all the tasks by itself and contributing n to the work of the system, or by receiving information from other processors by waiting for messages for $c \cdot n$ time steps. In either case the contribution is $\Omega(n)$ to the work of the algorithm. Since there are p processors, the work is $\Omega(n \cdot p)$. □

Lastly we note that since in this chapter we are trading communication for work, we design algorithms with the focus on work.

7.2 Delay-Sensitive Lower Bounds on Work

In this section we develop delay-sensitive lower bounds for asynchronous algorithms for the *Do-All* problem. for deterministic and randomized algorithms. We show that any deterministic (randomized) algorithm with p asynchronous processors and n tasks has worst-case total-work (respectively expected total-work) of $\Omega(n + p\, d \log_{d+1} n)$ under adversary $\mathcal{A}_D^{(d)}$, where d is the upper bound on message delay (unknown to the processors). This shows that work grows with d and becomes $\Omega(p\, n)$ as d approaches n.

We start by showing that the lower bound on work of $\Omega(n + p \log p)$ from Theorem 5.2 for the model with crashes and restarts also applies to the asynchronous model of computation, regardless of the delay. Note that the explicit construction in the proof below shows that the same bound holds in the asynchronous setting where no processor crashes.

Theorem 7.3. *Any asynchronous p-processor algorithm solving the Do-All problem on inputs of size n has total-work $S(n, p) = n + \Omega(p \log p)$.*

Proof. We present a strategy for the adversary that results in the worst case behavior. Let A be the best possible algorithm that solves the *Do-All* problem. The adversary imposes delays on the processor steps (regardless of what the message delay is) as described below:
Stage 1: Let $u > 1$ be the number of remaining tasks. Initially $u = n$. The adversary induces no delays as long as the number of remaining tasks, u, is more than p. The work needed to perform $n-p$ tasks when there are no delays is at least $n - p$.
Stage 2: As soon as a processor is about to perform some task $n-p+1$ making $u \leq p$, the adversary uses the following strategy. For the upcoming iteration, the adversary examines the algorithm to determine how the processors are assigned to the remaining tasks. The adversary then lists the remaining tasks with respect to the number of processors assigned to them. The adversary delays the processors assigned to the first half remaining tasks ($\lfloor \frac{u}{2} \rfloor$) with the least number of processors assigned to them. By an averaging argument, there are no more than $\lceil \frac{p}{2} \rceil$ processors assigned to these $\lfloor \frac{u}{2} \rfloor$ tasks. Hence at least $\lfloor \frac{p}{2} \rfloor$ processors will complete this iteration having performed no more than half of the remaining tasks.

The adversary continues this strategy which results in performing at most half of the remaining tasks at each iteration. Since initially $u = p$ in this stage, the adversary can continue this strategy for at least $\log p$ iterations. Considering these two stages the work performed by the algorithm is:

$$S(n,p) \geq \underbrace{n-p}_{Stage\ 1} + \underbrace{\lfloor p/2 \rfloor \log p}_{Stage\ 2} = n + \Omega(p \log p). \qquad \square$$

The above lower bound holds for arbitrarily small delays. We next develop a lower bound for the settings where the delay is non-negligible, specifically we assume $d \geq 1$.

7.2.1 Deterministic Delay-Sensitive Lower Bound

First we prove a lower bound on work that shows how the efficiency of work-performing deterministic algorithms depends on the number of processors p, the number of tasks n, and the message delay d.

Theorem 7.4. *Any deterministic algorithm solving Do-All with n tasks using p asynchronous message-passing processors against adversary $\mathcal{A}_D^{(d)}$ performs work $S(n,p,d) = \Omega(n + p \min\{d,n\} \log_{d+1}(d+n))$.*

Proof. That the required work is at least n is obvious — each task must be performed. We present the analysis for $n > 5$ and n that is divisible by 6 (this is sufficient to prove the lower bound). We present the following adversarial strategy. The adversary partitions computation into stages, each containing $\min\{d, n/6\}$ steps. We assume that the adversary delivers all messages sent to a processor in stage s at the end of stage s (recall that the receiver can

process any such message later, according to its own local clock) — this is allowed since the length of stage s is at most d. For stage s we will define the set of processors P_s such that the adversary delays all processors not in P_s. More precisely, each processor in P_s is not delayed during stage s, but any processor not in P_s is delayed so it does not complete any step during stage s.

Consider stage s. Let $u_s > 0$ be the number of tasks that remain unperformed at the beginning of stage s, and let U_s be the set of such tasks. We now show how to define the set P_s. Suppose first that each processor is not delayed during stage s (with respect to the time unit). Let $J_s(i)$, for every processor i, $i \in \mathcal{P}$ (recall that $\mathcal{P}$ is the set of all processors), denote the set of tasks from U_s (we do not consider tasks not in U_s in the analysis of stage s since they were performed before) which are performed by processor i during stage s (recall that inside stage s processor i does not receive any message from other processors, by the assumption on consider kind of the adversary). Note that $|J_s(i)|$ is at most $\min\{d, n/6\}$, which is the length of a stage.

Claim. *There are at least* $\frac{u_s}{3\min\{d,n/6\}}$ *tasks* z *such that each of them is contained in at most* $2p\min\{d, n/6\}/u_s$ *sets in the family* $\{J_s(i) \mid i \in \mathcal{P}\}$.

We prove the claim by the pigeonhole principle. If the claim is not true, then there would be more than $u_s - \frac{u_s}{3\min\{d,t/6\}}$ tasks such that each of them would be contained in more than $2p\min\{d, n/6\}/u_s$ sets in the family $\{J_s(i) \mid i \in \mathcal{P}\}$. This yields a contradiction because the following inequality holds

$$\begin{aligned} p\ \min\{d, n/6\} &= \sum_{i\in\mathcal{P}} |J_s(i)| \\ &\geq \Big(u_s - \frac{u_s}{3\min\{d, n/6\}}\Big) \cdot \frac{2p\min\{d, n/6\}}{u_s} \\ &= \Big(2 - \frac{2}{3\min\{d, n/6\}}\Big) \cdot p\min\{d, n/6\} \\ &> p\ \min\{d, n/6\}\ , \end{aligned}$$

since $d \geq 1$ and $n > 4$. This proves the claim.

We denote the set of $\frac{u_s}{3\min\{d,n/6\}}$ tasks from the above claim by J_s. We define P_s to be the set $\{i : J_s \cap J_s(i) = \emptyset\}$. By the definition of tasks $z \in J_s$ we obtain that

$$|P_s| \geq p - \frac{u_s}{3\min\{d, n/6\}} \cdot \frac{2p\min\{d, n/6\}}{u_s} \geq p/3\ .$$

Since all processors, other that those in P_s, are delayed during the whole stage s, work performed during stage s is at least $\frac{p}{3} \cdot \min\{d, n/6\}$, and all tasks from J_s remains unperformed. Hence the number u_{s+1} of undone tasks after stage s is still at least $\frac{u_s}{3\min\{d,n/6\}}$.

If $d < n/6$ then work during stage s is at least $p\ d/6$, and there remain at least $\frac{u_s}{3d}$ unperformed tasks. Hence this process may be continued, starting

with n tasks, for at least $\log_{3d} n = \Omega(\log_{d+1}(d+n))$ stages, until all tasks are performed. The total work is then $\Omega(p\, d \log_{d+1}(d+n))$.

If $d \geq n/6$ then during the first stage work performed is at least $p\, n/18 = \Omega(p\, n \log_{d+1}(d+n)) = \Omega(p\, n)$, and at the end of stage 1 at least $\frac{n}{3n/6} = 2$ tasks remain unperformed. Notice that this asymptotic value does not depend on whether the minimum is selected among d and n, or among d and $n/6$. More precisely, the works is

$$\Omega(p \min\{d, n\} \log_{d+1}(d+n)) = \Omega(p \min\{d, n/6\} \log_{d+1}(d+n)),$$

which completes the proof. □

7.2.2 Delay-sensitive Lower Bound for Randomized Algorithms

In this section we prove a delay-sensitive lower bound for randomized work-performing algorithms. We first state a technical lemma (without a proof) that we put to use in the lower bound proof.

Lemma 7.5. *For* $1 \leq d \leq \sqrt{u}$ *the following holds* $\frac{1}{4} \leq \frac{\binom{u-d}{u/(d+1)}}{\binom{u}{u/(d+1)}} \leq \frac{1}{e}$.

The idea behind the lower bound proof for randomized algorithms we present below is similar to the one for deterministic algorithms in the previous section, except that sets $J_s(i)$ are random, hence we have to modify the construction of set P_s also. We partition the execution of the algorithms into stages, similarly to the lower bound for deterministic algorithms. Recall that $\mathcal{P}$ is the set of p processors. Let U_s denote the remaining tasks at the beginning of stage s. Suppose first that all processors are not delayed during stage s, and the adversary delivers all messages sent to processor i during stage s at the end of stage s. The set $J_s(i)$, for processor $i \in \mathcal{P}$, denotes a certain set of tasks from U_s that i is going to perform during stage s. The size of $J_s(i)$ is at most d, because we consider at most d steps in advance (the adversary may delay all messages by d time steps, and so the choice of $J_s(i)$ does not change during next d steps, provided $|J_s(i)| \leq d$). The key point is that the set $J_s(i)$ is random, since we consider randomized algorithms, and so we deal with the probabilities that $J_s(i) = Y$ for the set of tasks $Y \subseteq U_s$ of size at most d. We denote these probabilities by $p_i(Y)$. For some given set of processors P, let $J_s(P)$ denote set $\bigcup_{i \in P} J_s(i)$.

The goal of the adversary is to prevent the processors from completing some sufficiently large set J_s of tasks during stage s. Here we are interested in the events where there is a set of processors P_s that is "large enough" (linear size) so that the processors do not perform any tasks from J_s.

In the next lemma we prove that, for some set J_s, such set of processors P_s exists with high probability. This is the main difference compared to the deterministic lower bound — instead of finding a suitably large set J_s *and* a

linear-size set P_s, we prove that the set J_s exists, and we prove that the set P_s of processors not performing this set of tasks during stage s *exists with high probability*. However in the final proof, the existence with high probability is sufficient — we can define the set on-line using the rule that if some processor wants to perform a task from the chosen set J_s, then we delay it, and do not put it in P_s. In the next lemma we assume that s is known, so we skip lower index s from the notation for clarity of presentation.

Lemma 7.6. *There exists set $J \subseteq U$ of size $\frac{u}{d+1}$ such that*

$$\Pr[\exists_{P \subseteq \mathcal{P}} : |P| = p/64 \ \wedge \ J(P) \cap J = \emptyset] \geq 1 - e^{-p/512} \ .$$

Proof. First observe that

$$\begin{aligned} \sum_{(J:\ J \subseteq U,\ |J| = \frac{u}{d+1})} \ \sum_{(v \in \mathcal{P})} \ \sum_{(Y:\ Y \subseteq U,\ Y \cap J = \emptyset,\ |Y| \leq d)} p_v(Y) \ &= \\ = \ \sum_{(v \in \mathcal{P})} \ \sum_{(Y:\ Y \subseteq U,\ |Y| \leq d)} p_v(Y) \cdot \binom{u - |Y|}{u/(d+1)} \\ \geq \ p \cdot \binom{u-d}{u/(d+1)} \ . \end{aligned}$$

It follows that there exists set $J \subseteq U$ of size $\frac{u}{d+1}$ such that

$$\sum_{(v \in \mathcal{P})} \ \sum_{(Y:\ Y \subseteq U,\ Y \cap J = \emptyset,\ |Y| \leq d)} p_v(Y) \geq \frac{p \cdot \binom{u-d}{u/(d+1)}}{\binom{u}{u/(d+1)}} \geq \frac{p}{4} \ , \tag{7.1}$$

where the last inequality follows from Lemma 7.5. Fix such a set J. For every node $v \in \mathcal{P}$, let

$$Q_v = \sum_{(Y:\ Y \subseteq U,\ Y \cap J = \emptyset,\ |Y| \leq d)} p_v(Y) \ .$$

Notice that $Q_v \leq 1$. Using the pigeonhole principle to Inequality 7.1, there is a set $V' \subseteq \mathcal{P}$ of size $p/8$ such that for every $v \in V'$

$$Q_v \geq \frac{1}{8} \ .$$

(Otherwise more than $7p/8$ nodes $v \in \mathcal{P}$ would have $Q_v < 1/8$, and fewer than $p/8$ nodes $v \in \mathcal{P}$ would have $Q_v \leq 1$. Consequently $\sum_{v \in \mathcal{P}} S_v < 7p/64 + p/8 < p/4$, which would contradict (7.1)). For every $v \in V'$, let X_v be the random variable equal 1 with probability Q_v, and 0 with probability $1 - Q_v$. These random variables constitute sequence of independent 0-1 trials. Let $\mu = \mathbb{E}[\sum_{v \in V'} X_v] = \sum_{v \in V'} Q_v$. Applying Chernoff bound we obtain

$$\Pr\left[\sum_{v \in V'} X_v < \mu/2\right] < e^{-\mu/8} \ ,$$

and consequently, since $\mu \geq \frac{p}{8} \cdot \frac{1}{8} = \frac{p}{64}$, we have

$$\begin{aligned} \Pr\left[\sum_{v \in V'} X_v < p/64\right] &\leq \Pr\left[\sum_{v \in V'} X_v < \mu/2\right] \\ &< e^{-\mu/8} \leq e^{-p/512} . \end{aligned}$$

Finally observe that

$$\begin{aligned} &\Pr\left[\exists_{P \subseteq \mathcal{P}} : |P| = p/64 \ \wedge \ J(P) \cap J = \emptyset\right] \\ &\geq 1 - \Pr\left[\sum_{v \in V'} X_v < p/64\right] , \end{aligned}$$

which completes the proof of the lemma. □

We apply Lemma 7.6 in proving the following lower bound result.

Theorem 7.7. *Any randomized algorithm solving Do-All with n tasks using p asynchronous message-passing processors against adversary $\mathcal{A}_D^{(d)}$ performs expected work $ES(n, p, d) = \Omega(n + p \min\{d, n\} \log_{d+1}(d + n))$.*

Proof. That the lower bound of $\Omega(t)$ holds with probability 1 is obvious. We consider three cases, depending on how large is d comparing to n: in the first case d is very small comparing to n (in this case the thesis follows from the simple calculations), in the second case we assume that d is larger than in the first, but still no more than $\sqrt{n}$ (this is the main case), and in the third case d is large than $\sqrt{n}$ (here the proof is similar to the second case, but is restricted to one stage). We now give the details.

Case 1: Inequalities $1 \leq d \leq \sqrt{n}$ and $1 - e^{-p/512} \cdot \log_{d+1} n < 1/2$ hold.

This case is a simple derivation. It follows that $\log_{d+1} n > e^{p/512}/2$, and next $\sqrt[3]{n} > p + d + \log_{d+1} n$ for sufficiently large p and n. More precisely:

$\sqrt[3]{n} > 3p$	for sufficiently large p, since $n > \log_{d+1} n > e^{p/512}$;
$\sqrt[3]{n} > 3d$	for sufficiently large p, since $d^{e^{p/512}/2} < n$;
$\sqrt[3]{n} > 3 \log_{d+1} n$	for sufficiently large n, since $d \geq 1$ and by the properties of the logarithm function.

Consequently, $n = (\sqrt[3]{n})^3 > pd \log_{d+1} n$ for sufficiently large p and n, and the lower bound

$$\Omega(n) = \Omega(p \ d \log_{d+1} n) = \Omega(p \ d \log_{d+1}(d + n))$$

holds, with the probability 1, in this case.

Case 2: Inequalities $1 \leq d \leq \sqrt{n}$ and $1 - e^{-p/512} \cdot \log_{d+1} n \geq 1/2$ hold.

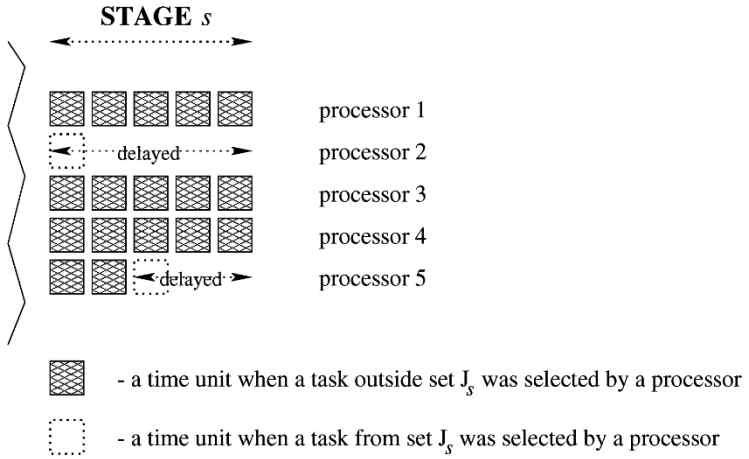
Consider any *Do-All* algorithm. Similarly as in the proof of Theorem 7.4, the adversary partitions computation into stages, each containing d steps.

Let us fix an execution of the algorithm through the end of stage $s-1$. Consider stage s. We assume that the adversary delivers to a processor all messages sent in stage s at the end of stage s, provided the processor is not delayed at the end of stage s (any such message is processed by the receivers at a later time). Let $U_s \subseteq T$ denote set of tasks that remain unperformed by the end of stage $s-1$. Here, by the adversarial strategy (no message is received and processed during stage s), given that the execution is fixed at the end of stage $s-1$, one can fix a distribution of processor i performing the set of tasks Y during stage s — this distribution is given by the probabilities $p_i(Y)$. The adversary derives the set $J_s \subseteq U_s$, using Lemma 7.6 according to the set of all processors, the set of the unperformed tasks U_s, and the distributions $p_i(Y)$ fixed at the beginning of stage s according to the action of processors i in stage s. (In applying Lemma 7.6 we use the same notation, except that the quantities are subscripted according to the stage number s.)

The adversary additionally delays any processor i, not belonging to some set P_s, that attempts to perform a task from J_s before the end of stage s. The set P_s is defined on-line (this is one of the difference between the adversarial constructions in the proofs of the lower bounds for deterministic and randomized *Do-All* algorithms): at the beginning of stage s set P_s contains all processors; every processor i that is going to perform some task $z \in J_s$ at time τ in stage s, is delayed till the end of stage s and removed from set P_s. We illustrate the adversarial strategy for five processors and $d = 5$ in Figure 7.1.

We now give additional details of the adversarial strategy. Suppose $u_s = |U_s| > 0$ tasks remain unperformed at the beginning of stage s. As described above, we apply Lemma 7.6 to the set U_s and probabilities $p_i(Y)$ to find, at the very beginning of stage s, the set $J_s \subseteq U_s$ such that the probability that there exists a subset of processors P_s of cardinality $p/64$ such that none of them would perform any tasks from J_s during stage s is at least $1 - e^{-p/512}$. Next, during stage s the adversary delays (to the end of stage s) all processors that (according to the random choices during stage s) are going to perform some task from J_s. By Lemma 7.6, the set P_s of not-delayed processors contains at least $p - 63p/64 \geq p/64$ processors, and the set of the remaining tasks $U_{s+1} \supseteq J_s$ contains at least $\frac{u_s}{d+1}$ tasks, all with probability at least $1-e^{-p/512}$. If this happens, we call stage s *successful*.

It follows that the probability, that every stage $s < \log_{d+1} n$ is successful is at least $1 - e^{-p/512} \cdot \log_{d+1} n$. Hence, using the assumption for this case, with the probability at least $1-e^{-p/512} \cdot \log_{d+1} n \geq 1/2$, at the beginning of stage s there will be at least $n \cdot \left(\frac{1}{d+1}\right)^{\log_{d+1} n - 1} > 1$ unperformed tasks and work will be at least $(\log_{d+1} n - 1) \cdot dp/64$, since the work in one successful stage is at least $p/64$ (the number of non-delayed processors) times d (the duration of one stage). It follows that the expected work of this algorithm in the presence of our adversary is $\Omega(pd \log_{d+1} n) = \Omega(pd \log_{d+1}(d+n))$, because $1 \leq d \leq \sqrt{n}$. This completes the proof of Case 2.



Strategy of the adversary during stage s, where $p = d = 5$. Using the set J_s, which exists by Lemma 7.6, the adversary delays a processor from the moment where it wants to perform a task from J_s. Lemma 7.6 guarantees that at least a fraction of processors will not be delayed during stage s, with high probability.

Fig. 7.1. Illustration of the adversarial strategy leading to the delay-sensitive lower bound on total-work for randomized algorithms.

Case 3: Inequality $d > \sqrt{t}$ holds.

Here we follow similar reasoning as in the Case 2, except that we consider a single stage.

Consider first $\min\{d, n/6\}$ steps. Let T be the set of all tasks, and $p_i(Y)$ denote the probability that processor $i \in \mathcal{P}$ performs tasks in $Y \subseteq T$ of cardinality $\min\{d, n/6\}$ during the considered steps. Applying Lemma 7.6 we obtain, that at least $p/64$ processors are non-delayed during the considered steps, and after these steps at least $\frac{\min\{d,n/6\}}{d+1} \geq 1$ tasks remain unperformed, all with the probability at least $1 - e^{-p/512}$. Since $1 \leq \log_{d+1}(d+n) < 2$, work is $\Omega(p \min\{d, n/6\}) = \Omega(p \min\{d, n\} \log_{d+1}(d+n))$. This completes the proof of the third case and of the theorem. □

7.3 Contention of Permutations

In this section we present and generalize the notion of *contention* of permutations, and state several properties of contention (without proofs). Contention properties turn out to be important in the analysis of algorithms we present later in this chapter.

We use braces $\langle \ldots \rangle$ to denote an ordered list. For a list L and an element a, we use the expression $a \in L$ to denote the element's membership in the list, and the expression $L - K$ to stand for L with all elements in K removed.

We next provide a motivation for the material in this section. Consider the situation where two asynchronous processors, $\mathfrak{p}_1$ and $\mathfrak{p}_2$, need to perform n independent tasks with known unique identifiers from the set $[n] = \{1, \ldots, n\}$. Assume that before starting a task, a processor can check whether the task is complete; however if both processors work on the task concurrently, then the task is done twice because both find it to be not complete. We are interested in the number of tasks done redundantly.

Let $\pi_1 = \langle a_1, \ldots, a_n \rangle$ be the sequence of tasks giving the order in which $\mathfrak{p}_1$ intends to perform the tasks. Similarly, let $\pi_2 = \langle a_{s_1}, \ldots, a_{s_n} \rangle$ be the sequence of tasks of $\mathfrak{p}_2$. We can view π_2 as π_1 permuted according to $\sigma = \langle s_1, \ldots, s_t \rangle$ (π_1 and π_2 are permutations). With this, it is possible to construct an asynchronous execution for $\mathfrak{p}_1$ and $\mathfrak{p}_2$, where $\mathfrak{p}_1$ performs all t tasks by itself, and any tasks that $\mathfrak{p}_2$ finds to be unperformed are performed redundantly by both processors.

In the current context it is important to understand how does the structure of π_2 affect the number of redundant tasks. Clearly $\mathfrak{p}_2$ may have to perform task a_{s_1} redundantly. What about a_{s_2}? If $s_1 > s_2$ then by the time $\mathfrak{p}_2$ gets to task a_{s_2}, it is already done by $\mathfrak{p}_1$ according to π_1. Thus, in order for a_{s_2} to be done redundantly, it must be the case that $s_2 > s_1$. It is easy to see, in general, that for task a_{s_j} to be done redundantly, it must be the case that $s_j > \max\{s_1, \ldots, s_{j-1}\}$. Such s_j is called the *left-to-right maximum* of σ. The total number of tasks done redundantly by $\mathfrak{p}_2$ is thus the number of left-to-right maxima of σ. Not surprisingly, this number is minimized when $\sigma = \langle n, \ldots, 1 \rangle$, i.e, when π_2 is the reverse order of π_1, and it is maximized when $\sigma = \langle 1, \ldots, n \rangle$, i.e., when $\pi_1 = \pi_2$. In this section we will define the notion *contention* of permutations that captures the relevant left-to-right maxima properties of permutations that are to be used as processor schedules.

Now we proceed with formal presentation. Consider a list of some idempotent computational *jobs* with identifiers from the set $[n] = \{1, \ldots, n\}$. (We make the distinction between *tasks* and *jobs* for convenience to simplify algorithm analysis; a job may be composed of one or more tasks.) We refer to a list of job identifiers as a *schedule*. When a schedule for n jobs is a permutation of job identifiers π in $\mathcal{S}_n$, we call it a *n-schedule*. Here $\mathcal{S}_n$ is the symmetric group, the group of all permutations on the set $[n]$; we use the symbol $\circ$ to denote the composition operator, and $\mathbf{e}_n$ to denote the identity permutation. For a n-schedule $\pi = \langle \pi(1), \ldots, \pi(n) \rangle$ a *left-to-right maximum* is an element $\pi(j)$ of π that is larger than all of its predecessors, i.e., $\pi(j) > \max_{i<j}\{\pi(j-i)\}$.

Given a n-schedule π, we define $\textsc{lrm}(\pi)$, to be the number of left-to-right maxima in the n-schedule π. For a list of permutations $\Psi = \langle \pi_0, \ldots, \pi_{n-1} \rangle$ from $\mathcal{S}_n$ and a permutation δ in $\mathcal{S}_n$, the *contention* of Ψ with respect to δ is defined as $\mathrm{Cont}(\Psi, \delta) = \sum_{u=0}^{n-1} \textsc{lrm}(\delta^{-1} \circ \pi_u)$. The *contention of the list of schedules* Ψ is defined as $\mathrm{Cont}(\Psi) = \max_{\delta \in \mathcal{S}_n}\{\mathrm{Cont}(\Psi, \delta)\}$. Note that for any Ψ, we have $n \leq \mathrm{Cont}(\Psi) \leq n^2$. It turns out that it is possible to construct a family of permutations with following low contention (H_n is the nth harmonic number, $H_n = \sum_{j=1}^{n} \frac{1}{j}$).

Lemma 7.8. *For any $n > 0$ there exists a list of permutations $\Psi = \langle \pi_0, \ldots, \pi_{n-1} \rangle$ with* $\mathrm{Cont}(\Psi) \leq 3nH_n = \Theta(n \log n)$.

For a constant n, a list Ψ with $\mathrm{Cont}(\Psi) \leq 3nH_n$ can be found by exhaustive search. This costs only a constant number of operations on integers (however, this cost might be of order $(n!)^n$).

7.3.1 Contention and Oblivious Tasks Scheduling

Assume now that n distinct asynchronous processors perform the n jobs such that processor i performs the jobs in the order given by π_i in Ψ. We call this oblivious algorithm ObliDo and give the code in Figure 7.2. (Here each "processor" may be modeling a group of processors, where each processor follows the same sequence of activities.)

```
00 const Ψ = {π_r | 1 ≤ r ≤ n ∧ π_r ∈ S_n}    % Fixed set of n permutations of [n]
01 for each processor PID = 1..n begin
02   for r = 1 to n do
03         perform Job(π_pid(r))
04   od
05 end.
```

Fig. 7.2. Algorithm ObliDo.

Since ObliDo does not involve any coordination among the processors the total of n^2 jobs are performed (counting multiplicities). However, it can be shown that if we count only the job executions such that each job has not been previously performed by any processor, then the total number of such job executions is bounded by $\mathrm{Cont}(\Psi)$, again counting multiplicities. We call such job executions *primary*; we also call all other job executions *secondary*. Note that the number of primary executions cannot be smaller than n, since each job is performed at least once for the first time. In general this number is going to be between n and n^2, because several processors may be executing the same job concurrently for the first time.

Note that while an algorithm solving the *Do-All* problem may attempt to reduce the number of secondary job executions by sharing information about complete jobs among the processors, it is not possible to eliminate (redundant) primary job executions in the asynchronous model we consider. The following lemma formalizes the relationship between the primary job executions and the contention of permutations used as schedules.

Lemma 7.9. *In algorithm* ObliDo *with n processors, n tasks, and using the list Ψ of n permutations, the number of primary job executions is at most* $\mathrm{Cont}(\Psi)$.

7.3.2 Generalized Contention

Now we generalize the notion of contention and define *d-contention*. For a schedule $\pi = \langle \pi(1), \ldots, \pi(n) \rangle$, an element $\pi(j)$ of π is a *d-left-to-right maximum* (or d-lrm for short) if the number the elements in π preceding and greater than $\pi(j)$ is less than d, i.e., $|\{i : i < j \ \wedge \ \pi(i) > \pi(j)\}| < d$.

Given a n-schedule π, we define (d)-LRM(π) as the number of d-lrm's in the schedule π. For a list $\Psi = \langle \pi_0, \ldots, \pi_{p-1} \rangle$ of permutations from $\mathcal{S}_n$ and a permutation δ in $\mathcal{S}_n$, the d-contention of Ψ with respect to δ is defined as

$$(d)\text{-Cont}(\Psi, \delta) = \sum_{u=0}^{p-1} (d)\text{-LRM}(\delta^{-1} \circ \pi_u) \ .$$

The *d-contention of the list of schedules* Ψ is defined as

$$(d)\text{-Cont}(\Psi) = \max_{\delta \in \mathcal{S}_n} \{(d)\text{-Cont}(\Psi, \delta)\} \ .$$

We first show a lemma about the d-contention of a set of permutations with respect to $\mathbf{e}_n$, the identity permutation.

Lemma 7.10. *Let Ψ be a list of p random permutations from $\mathcal{S}_n$. For every fixed positive integer d, the probability that (d)-Cont$(\Psi, \mathbf{e}_n) > n \ln n + 8pd \ln(e + n/d)$ is at most $e^{-\left(n \ln n + 7pd \ln(e + \frac{n}{3d})\right) \ln(7/e)}$.*

Proof. For $d \geq n/5$ the thesis is obvious. In the remainder of the proof we assume $d < n/5$.

First we describe a well known method for generating a random schedule by induction on the number of elements $n' \leq n$ to be permuted. For $n' = 1$ the schedule consists of a single element chosen uniformly at random. Suppose we can generate a random schedule of $n' - 1$ different elements. Now we show how to schedule n' elements uniformly and independently at random. First we choose uniformly and independently at random one element among n' and put it as the last element in the schedule. By induction we generate random schedule from remaining $n' - 1$ elements and put them as the first $n' - 1$ elements. Simple induction proof shows that every obtained schedule of n' elements has equal probability (since the above method is a concatenation of two independent and random events).

A random list of schedules Ψ can be selected by using the above method p times, independently.

For a schedule $\pi \in \Psi$, let $X(\pi, i)$, for $i = 1, \ldots, n$, be a random value such that $X(\pi, i) = 1$ if $\pi(i)$ is a d-lrm, and $X(\pi, i) = 0$ otherwise.

Claim. *For any $\pi \in \Psi$, $X(\pi, i) = 1$ with probability $\min\{d/i, 1\}$, independently from other values $X(\pi, j)$, for $j > i$. Restated precisely, we claim that $\Pr[X(\pi, i) = 1 \mid \bigwedge_{j>i} X(\pi, j) = a_j] = \min\{d/i, 1\}$, for any 0-1 sequence $a_{i+1}, \ldots, a_n$.*

This is so because $\pi(i)$ might be a d-lrm if during the $(n-i-1)$th step of generating π, we select uniformly and independently at random one among the d greatest remaining elements (there are i remaining elements in this step). This proves the claim.

Note that

1. for every $\pi \in \Psi$ and every $i = 1, \ldots, d$, $\pi(i)$ is d-lrm, and
2. $\mathbb{E}\left[\sum_{\pi\in\Psi}\sum_{i=d+1}^{n} X(\pi,i)\right] = p\,d \cdot \sum_{i=d+1}^{n} \frac{1}{i} = p\,d\,(H_n - H_d)$.

Applying the well known Chernoff bound of the following form: for 0-1 independent random variables Y_j and any constant $b > 0$,

$$\Pr\left[\sum_j Y_j > \mathbb{E}\left[\sum_j Y_j\right](1+b)\right] < \left(\frac{e^b}{(1+b)^{1+b}}\right)^{\mathbb{E}\left[\sum_j Y_j\right]} < e^{-\mathbb{E}\left[\sum_j Y_j\right](1+b)\ln\frac{1+b}{e}},$$

and using the fact that $2 + \dfrac{n\ln n}{pd(H_n - H_d)} > 0$, we obtain

$$\begin{aligned}
&\Pr\left[\sum_{\pi\in\Psi}\sum_{i=d+1}^{n} X(\pi,i) > n\ln n + 3pd(H_n - H_d)\right] \\
&\quad = \Pr\left[\sum_{\pi\in\Psi}\sum_{i=d+1}^{n} X(\pi,i) > pd(H_n - H_d)\left(1 + \left(2 + \frac{n\ln n}{pd(H_n - H_d)}\right)\right)\right] \\
&\quad \le e^{-(n\ln n + 3pd(H_n - H_d))\ln\frac{n\ln n + 3pd(H_n - H_d)}{e\cdot pd(H_n - H_d)}} \\
&\quad \le e^{-[n\ln n + 3pd(H_n - H_d)]\ln(3/e)}.
\end{aligned}$$

Since $\ln i \le H_i \le \ln i + 1$ and $n > 5d$, we obtain that

$$\begin{aligned}
\Pr&\left[\sum_{\pi\in\Psi}\sum_{i=1}^{n} X(\pi,i) > n\ln n + 5pd\ln\left(e + \frac{n}{d}\right)\right] \\
&\le \Pr\left[\sum_{\pi\in\Psi}\sum_{i=d+1}^{n} X(\pi,i) > n\ln n + 3pd(H_n - H_d) + pd\right] \\
&\le e^{-[n\ln n + 3pd(H_n - H_d)]\ln(3/e)}.
\end{aligned}$$

□

Now we generalize the result of Lemma 7.10.

Theorem 7.11. *For a random list of schedules Ψ containing p permutations from $\mathcal{S}_n$, the event:*

"for every positive integer d, (d)-Cont$(\Psi) > n\ln n + 8pd\ln(e + n/d)$",

holds with probability at most $e^{-n\ln n\cdot\ln(7/e^2)-p}$.

Proof. For $d \geq n/5$ the result is straightforward, moreover the event holds with probability 0. In the following we assume that $d < n/5$.

Note that since Ψ is a random list of schedules, then so is $\sigma^{-1} \circ \Psi$, where $\sigma \in \mathcal{S}_n$ is an arbitrary permutation. Consequently, by Lemma 7.10, $(d)\text{-Cont}(\Psi, \sigma) > n \ln n + 8pd \ln(e + n/d)$ holds with probability at most $e^{-[n \ln n + 7pd \ln(e + \frac{n}{3d})] \ln \frac{7}{e}}$.

Hence the probability that a random list of schedules Ψ has d-contention greater than $n \ln n + 8pd \ln(e + n/d)$ is at most

$$\begin{aligned} n! \cdot e^{-[n \ln n + 7pd \ln(e + \frac{n}{3d})] \ln \frac{7}{e}} &\leq e^{n \ln n - [n \ln n + 7pd \ln(e + \frac{n}{3d})] \ln \frac{7}{e}} \\ &\leq e^{-n \ln n \cdot \ln \frac{7}{e^2} - 7pd \ln(e + \frac{n}{d})} . \end{aligned}$$

Then the probability that, for every d, $(d)\text{-Cont}(\Psi) > n \ln n + 8pd \ln(e + n/d)$, is at most

$$\begin{aligned} &\sum_{d=1}^{\infty} \Pr\left[(d)\text{-Cont}(\Psi) > n \ln n + 8pd \ln(e + n/d)\right] \\ &\qquad \leq \sum_{d=1}^{n/5-1} e^{-n \ln n \cdot \ln(7/e^2) - 7pd \ln(e+n/d)} + \sum_{d=n/5}^{\infty} 0 \\ &\qquad \leq e^{-n \ln n \cdot \ln(7/e^2)} \cdot \sum_{d=1}^{n/5-1} (e^{-7p})^d \\ &\qquad \leq e^{-n \ln n \cdot \ln(7/e^2)} \cdot \frac{e^{-7p}}{1 - e^{-7p}} \\ &\qquad \leq e^{-n \ln n \cdot \ln(7/e^2) - p} . \end{aligned}$$

□

Using the probabilistic method we obtain the following.

Corollary 7.12. *There is a list of p schedules Ψ from $\mathcal{S}_n$ such that $(d)\text{-Cont}(\Psi) \leq n \log n + 8pd \ln(e + n/d)$, for every positive integer d.*

We put to use our generalized notion of contention in the delay-sensitive analysis of work-performing algorithms in Section 7.5.

7.4 Deterministic Algorithms Family DA

We now present a deterministic solution for the *Do-All* problem with p processors and n tasks. We develop a family of deterministic algorithms DA, such that for any constant $\varepsilon > 0$ there is an algorithm with total-work $S = O(np^{\varepsilon} + p\ d\lceil n/d \rceil^{\varepsilon})$ and message complexity $M = O(p \cdot S)$.

More precisely, algorithms from the family DA are parameterized by a positive integer q and a list Ψ of q permutations on the set $[q] = \{1, \ldots, q\}$, where $2 \leq q < p \leq n$. We show that for any constant $\varepsilon > 0$ there is a constant q and a corresponding set of permutation Ψ, such that the resulting algorithm has total-work $S = O(np^{\varepsilon} + p\ d\lceil n/d \rceil^{\varepsilon})$ and message complexity $M = O(p \cdot S)$. The work of these algorithms is within a small polynomial factor of the corresponding lower bound (see Section 7.2.1).

7.4.1 Construction and Correctness of Algorithm DA(q)

Let q be some constant such that $2 \leq q \leq p$. We assume that the number of tasks t is an integer power of q, specifically let $t = q^h$ for some $h \in \mathbb{N}$. When the number of tasks is not a power of q we can use a standard padding technique by adding just enough "dummy" tasks so that the new number of tasks becomes a power of q; the final results show that this padding does not affect the asymptotic complexity of the algorithm. We also assume that $\log_q p$ is a positive integer. If it is not, we pad the processors with at most qp "infinitely delayed" processors so this assumption is satisfied; in this case the upper bound is increased by a (constant) factor of at most q.

The algorithm uses any list of q permutations $\Psi = \langle \pi_0, \ldots \pi_{q-1} \rangle$ from $\mathcal{S}_q$ such that Ψ has the minimum contention among all such lists. We define a family of algorithms, where each algorithm is parameterized by q, and a list Ψ with the above contention property. We call this algorithm DA(q). In this section we first present the algorithm for $p \geq n$, then state the parameterization for $p < n$.

Algorithm DA(q), utilizes a q-ary boolean *progress tree* with n leaves, where the tasks are associated with the leaves. Initially all nodes of the tree are 0 (false) indicating that no tasks have been performed. Instead of maintaining a global data structure representing a q-ary tree, in our algorithms each processor has a replica of the tree.

Whenever a processor learns that all tasks in a subtree rooted at a certain node have been performed, it sets the node to 1 (true) and shares the good news with all other processors. This is done by multicasting the processor's progress tree; the local replicas at each processor are updated when multicast messages are received.

Each processor, acting independently, searches for work in the smallest immediate subtree that has remaining unperformed tasks. It then performs any tasks it finds, and moves out of that subtree when all work within it is completed. When exploring the subtrees rooted at an interior node at height m, a processor visits the subtrees in the order given by one of the permutations in Ψ. Specifically, the processor uses the permutation π_s such that s is the value of the m-th digit in the q-ary expansion of the processor's identifier (pid). We now present this in more detail.

```
00 const q                                              % Arity of the progress tree
01 const Ψ = ⟨π_r | 0 ≤ r < q ∧ π_r ∈ S_q⟩              % Fixed list of q permutations of [q]
02 const l = (qt−1)/(q−1)                               % The size of the progress tree
03 const h = log_q n                                    % The height of the progress tree
04 type ProgressTree: array [0 .. l − 1] of boolean     % Progress tree
05 for each processor pid = 1 to p begin
06   ProgressTree Tree_pid                              % The progress tree at processor pid
10   thread                                             % Traverse progress tree in search of work
11         integer ν init = 0                           % Current node, begin at the root
12         integer η init = 0                           % Current depth in the tree
13         DoWork(ν, η)
14   end
20   thread                                             % Receive broadcast messages
21         set of ProgressTree B                        % Incoming messages
22         while Tree_pid[0] ≠ 1 do                     % While not all tasks certified
23               receive B                              % Deliver the set of received messages
24               Tree_pid := Tree_pid ∨ (⋁_{b∈B} b)     % Learn progress
25         od
26   end
27 end.
```

```
40 procedure DoWork(ν, η)                               % Recursive progress tree traversal
41                                                      % ν : current node index ; η : node depth
42 const array x[0 .. h − 1] = pid_(base q)              % h least significant q-ary digits of pid
43   if Tree_pid[ν] = 0 then                            % Node not done – still work left
44         if η = h then                                % Node ν is a leaf
45               perform Task(n − l + ν + 1)            % Do the task
46         else                                         % Node ν is not a leaf
47               or r = 1 to q do                       % Visit subtrees in the order of π_x[η]
48                     DoWork(qν + π_x[η](r), η + 1)
49               od
50         fi
51         Tree_pid[ν] := 1                             % Record completion of the subtree
52         broadcast Tree_pid                           % Share the good news
53   fi
54 end.
```

Fig. 7.3. The deterministic algorithm DA ($p \geq n$).

Data Structures: Given the n tasks, the progress tree is a q-ary ordered tree of height h, where $n = q^h$. The number of nodes in the progress tree is $l = \sum_{i=0}^{h-1} q^i = (q^{h+1}-1)/(q-1) = (qn-1)/(q-1)$. Each node of the tree is a boolean, indicating whether the subtree rooted at the node is done (value 1) or not (value 0).

The progress tree is stored in a boolean array $Tree[0 \,..\, l-1]$, where $Tree[0]$ is the root, and the q children of the interior node $Tree[\nu]$ being the nodes $Tree[q\nu + 1]$, $Tree[q\nu + 2], \ldots, Tree[q\nu + q]$. The space occupied by the tree

is $O(n)$. The n tasks are associated with the leaves of the progress tree, such that the leaf $Tree[\nu]$ corresponds to the task $Task(\nu + n + 1 - l)$.

We represent the *pid* of each of the p processors in terms of its q-ary expansion. We care only about the h least significant q-ary digits of each *pid* (thus when $p > n$ several processors may be indistinguishable in the algorithm). The q-ary expansions of each *pid* is stored in the array $x[0..h-1]$.

Control Flow: The code is given in Figure 7.3. Each of the p processors executes two concurrent threads. One thread (lines 10-14) traverses the local progress tree in search work, performs the tasks, and broadcasts the updated progress tree. The second thread (lines 20-26) receives messages from other processors and updates the local progress tree. (Each processor is asynchronous, but we assume that its two threads run at approximately the same speed. This is assumed for simplicity only, as it is trivial to explicitly schedule the threads on a single processor.) Note that the updates of the local progress tree *Tree* are always monotone: initially each node contain 0, then once a node changes its value to 1 it remains 1 forever. Thus no issues of consistency arise.

The progress tree is traversed using the recursive procedure DOWORK (lines 40-54). The order of traversals within the progress tree is determined by the list of permutations $\Psi = \langle \pi_0, \pi_1, \ldots, \pi_{q-1} \rangle$. Each processor uses, at the node of depth η, the η^{th} q-ary digit $x[\eta]$ of its *pid* to select the permutation $\pi_{x[\eta]}$ from Ψ (recall that we use only the h least significant q-ary digits of each *pid* when representing the *pid* in line 42). The processor traverses the q subtrees in the order determined by $\pi_{x[\eta]}$ (lines 47-49); the processors starts the traversal of a subtree only if the corresponding bit in the progress tree is not set (line 43).

In other words, each processor *pid* traverses its progress tree in a post-order fashion using the q-ary digits of its *pid* and the permutations in Ψ to establish the order of the subtree traversals, except that when the messages from other processors are received, the progress tree of processor *pid* can be pruned based on the progress of other processors.

Parameterization for Large Number of Tasks: When the number of input tasks n' exceeds the number of processors p, we divide the tasks into *jobs*, where each job consists of at most $\lceil n'/p \rceil$ tasks. The algorithm in Figure 7.3 is then used with the resulting p jobs ($p = n$), where $Task(j)$ now refers to the job number j ($1 \leq j \leq n$). Note that in this case the cost of work corresponding to doing a single job is $\lceil n'/p \rceil$.

Correctness: We claim that algorithm DA(q) correctly solves the *Do-All* problem. This follows from the observation that a processor leaves a subtree by returning from a recursive call to DOWORK if and only if the subtree contains no unfinished work and its root is marked accordingly. We formalize this as follows.

Lemma 7.13. *In any execution of algorithm* DA(q)*, whenever a processor returns from a call to* DOWORK(ν, η)*, all tasks associated with the leaves that are the descendants of node ν have been performed.*

Proof. First, by code inspection (Figure 7.3, lines 45, 51, and 52), we note that processor *pid* reaching a leaf n at depth $\eta = h$ broadcasts its $Tree_{pid}$ with the value $Tree_{pid}[\nu]$ set to 1 if and only if it performs the task corresponding to the leaf.

We now proceed by induction on η.
Base case, $\eta = h$:
In this case, processor *pid* makes the call to DOWORK(ν, η). If $Tree_{pid}[\nu] = 0$, as we have already observed, the processor performs the task at the leaf (line 45), broadcasts its $Tree_{pid}$ with the leaf value set to 1 (lines 51-52), and returns from the call. If $Tree_{pid}[\nu] \neq 0$ then the processor must have received a message from some other processor indicating that the task at the leaf is done. This can be so if the sender itself performed the task (as observed above), or the sender learned from some other processor the fact that the task is done.
Inductive step, $0 \leq \eta < h$:
In this case, processor *pid* making the call to DOWORK(ν, η) executes q calls to DOWORK($\nu', \eta + 1$), one for each child ν' of node ν (lines 47-49). By inductive hypothesis, each return from DOWORK($\nu', \eta + 1$) indicates that all tasks associated with the leaves that are the descendants of node ν' have been performed. The processor then broadcasts its $Tree_{pid}$ with the the value $Tree_{pid}[\nu]$ set to 1 (lines 51-52), indicating that all tasks associated with the leaves that are the descendants of node ν have been performed, and returns from the call. □

Theorem 7.14. *Any execution of algorithm* DA(q) *terminates in finite time having performed all tasks.*

Proof. The progress tree used by the algorithm has finite number of nodes. By code inspection, each processor executing the algorithm makes at most one recursive call per each node of the tree. Thus the algorithm terminates in finite time. By Lemma 7.13, whenever a processor returns from the call to DOWORK(ν (= 0), η (= 0)), all tasks associated with the leaves that are the descendants of the node $\nu = 0$ are done, and the value of node is set to 1. Since this node is the root of the tree, all tasks are done. □

7.4.2 Complexity Analysis of Algorithm DA(q)

We start by showing a lemma that relates the work of the algorithm, against adversary $\mathcal{A}_D^{(d)}$ to its recursive structure.

We consider the case $p \geq n$. Let $S(n, p, d)$ denote total-work of algorithm DA(q) through the first global step in which some processor completes the last remaining task and broadcasts the message containing the progress tree where $T[0] = 1$. We note that $S(1, p, d) = O(p)$. This is because the progress tree

has only one leaf. Each processor makes a single call to DoWork, performs the sole task and broadcasts the completed progress tree.

Lemma 7.15. *For p-processor, n-task algorithm* DA(q) *with* $p \geq n$ *and* n *and* p *divisible by* q*:*

$$S(n,p,d) = O(\mathrm{Cont}(\Psi) \cdot S(p/q, n/q, d) + p \cdot q \cdot \min\{d, n/q\}) \ .$$

Proof. Since the root of the progress tree has q children, each processor makes the initial call to DoWork$(0,0)$ (line 13) and then (in the worst case) it makes q calls to DoWork (line 47-49) corresponding to the children of the root. We consider the performance of all tasks in the specific subtree rooted at a child of the progress tree as a job, thus such a job consists of all invocations of DoWork on that subtree. We now account separately for the primary and secondary job executions (recall the definitions in Section 7.3).

Observe that the code in lines 47-49 of DA is essentially algorithm ObliDo (lines 02-04 in Figure 7.2) and we intend to use Lemma 7.9. The only difference is that instead of q processors we have q groups of p/q processors where in each group the *pid*s differ in their q-ary digit corresponding to the depth 0 of the progress tree. From the recursive structure of algorithm DA it follows that the work of each such group in performing a single job is $S(p/q, n/q, d)$, since each group has p/q processors and the job includes n/q tasks. Using Lemma 7.9 the primary task executions contribute $O(\mathrm{Cont}(\Psi) \cdot S(p/q, n/q, d))$ work.

If messages were delivered without delay, there would be no need to account for secondary job executions because the processors would instantly learn about all primary job completions. Since messages can be delayed by up to d time units, each processor may spend up to d time steps, but no more than $O(n/q)$ steps performing a secondary job (this is because it takes a single processor $O(n/q)$ steps to perform a post-order traversal of a progress tree with n/q leaves). There are q jobs to consider, so for p processors this amounts to $O(p \cdot q \cdot \min\{d, n/q\})$ work.

For each processor there is also a constant overhead due to the fixed-size code executed per each call to DoWork. The total-work contribution is $O(p \cdot q)$. Finally, given the assumption about thread scheduling, the work of message processing thread does not exceed asymptotically the work of the DoWork thread. Putting all these work contributions together yields the desired result. □

We now prove the following theorem about total-work.

Theorem 7.16. *Consider algorithm* DA(q) *with* p *processors and* n *tasks where* $p \geq n$*. Let* d *be the maximum message delay. For any constant* $\varepsilon > 0$ *there is a constant* q *such that the algorithm has total-work* $S(n,p,d) = O(p \min\{n, d\} \lceil n/d \rceil^{\varepsilon})$.

Proof. Fix a constant $\varepsilon > 0$; without loss of generality we can assume that $\varepsilon \leq 1$. Let a be the sufficiently large positive constant "hidden" in the big-oh

upper bound for $S(n,p,d)$ in Lemma 7.15. We consider a constant $q > 0$ such that $\log_q(4a \log q) \leq \varepsilon$. Such q exists since $\lim_{q\to\infty} \log_q(4a \log q) = 0$ (however, q is a constant of order $2^{\frac{\log(1/\varepsilon)}{\varepsilon}}$).

First suppose that $\log_q n$ and $\log_q p$ are positive integers. We prove by induction on p and n that

$$S(n,p,d) \leq q \cdot n^{\log_q(4a \log q)} \cdot p \cdot d^{1-\log_q(4a \log q)} ,$$

For the *base case* of $n = 1$ the statement is correct since $S(1,p,d) = O(p)$. For $n > 1$ we choose the list of permutations Ψ with $\mathrm{Cont}(\Psi) \leq 3q \log q$ per Lemma 7.8. Due to our choice of parameters, $\log_q n$ is an integer and $n \leq p$. Let β stand for $\log_q(4a \log q)$. Using Lemma 7.15 and inductive hypothesis we obtain

$$\begin{aligned} S(n,p,d) &\leq a \cdot \left(3q \log q \cdot q \cdot \left(\frac{n}{q}\right)^{\beta} \cdot \frac{p}{q} \cdot d^{1-\beta} + \; p \cdot q \cdot \min\{d, n/q\}\right) \\ &\leq a \cdot \left(\left(q \cdot n^{\beta} \cdot p \cdot d^{1-\beta}\right) \cdot 3 \log q \cdot q^{-\beta} + \; p \cdot q \cdot \min\{d, n/q\}\right) . \end{aligned}$$

We now consider two cases:

Case 1: $d \leq n/q$. It follows that

$$p \cdot q \cdot \min\{d, n/q\} = p \; q \; d \leq p \; q \; d^{1-\beta} \cdot \left(\frac{n}{q}\right)^{\beta} .$$

Case 2: $d > n/q$. It follows that

$$p \cdot q \cdot \min\{d, n/q\} = p \; n \leq p \; q \; d^{1-\beta} \cdot \left(\frac{n}{q}\right)^{\beta} .$$

Putting everything together we obtain the desired inequality

$$S(n,p,d) \leq a \left(\left(q \cdot n^{\beta} \cdot p \cdot d^{1-\beta} \cdot q^{-\beta}\right) 4 \log q\right) \leq q \cdot n^{\beta} \cdot p \cdot d^{1-\beta} .$$

To complete the proof, consider any $n \leq p$. We add $n' - n$ new "dummy" tasks, where $n' - n < q \; n - 1$, and $p' - p$ new "virtual" processors, where $p' - p < q \; p - 1$, such that $\log_q n'$ and $\log_q p'$ are positive integers. We assume that all "virtual" crash at the start of the computation (else they can be thought of as delayed to infinity). It follows that

$$S(n,p,d) \leq S(n',p',d) \leq q \cdot (n')^{\beta} p' \cdot d^{1-\beta} \leq q^{2+\beta} n^{\beta} p \cdot d^{1-\beta} .$$

Since $\beta \leq \varepsilon$, we obtain that total-work of algorithm DA(q) is $O(\min\{n^{\varepsilon} p \; d^{1-\varepsilon}, n \; p\}) = O(p \min\{n,d\} \lceil n/d \rceil^{\varepsilon})$, which completes the proof of the theorem. □

Now we consider the case $p < n$. Recall that in this case we divide the n tasks into p jobs of size at most $\lceil n/p \rceil$, and we let the algorithm work with these jobs. It takes a processor $O(n/p)$ work (instead of a constant) to process a single job.

Theorem 7.17. *Consider algorithm* DA(q) *with p processors and n tasks where $p < n$. Let d be the maximum message delay. For any constant $\varepsilon > 0$ there is a constant q such that* DA(q) *has total-work $S(n,p,d) = O(np^{\varepsilon} + p\min\{n,d\}\lceil n/d\rceil^{\varepsilon})$.*

Proof. We use Theorem 7.16 with p jobs (instead of n tasks), were a single job takes $O(n/p)$ units of work. The upper bound on the maximal delay for receiving messages about the completion of some job is $d' = \lceil pd/n\rceil = O(1+pd/n)$ "job units", where a single job unit takes $\Theta(n/p)$ time. We obtain the following bound on work:

$$\begin{aligned} O\left(p\min\{p,d'\}\lceil p/d'\rceil^{\varepsilon}\cdot\frac{n}{p}\right) &= O\left(\min\left\{p^2, p^{\varepsilon}p(d')^{1-\varepsilon}\right\}\cdot\frac{n}{p}\right) \\ &= O\left(\min\left\{n\ p, n\ p^{\varepsilon} + pn^{\varepsilon}d^{1-\varepsilon}\right\}\right) \\ &= O\left(n\ p^{\varepsilon} + p\min\{n,d\}\left\lceil\frac{n}{d}\right\rceil^{\varepsilon}\right)\ . \end{aligned}$$

□

Finally we consider message complexity.

Theorem 7.18. *Algorithm* DA(q) *with p processors and n tasks has message complexity $M(n,p,d) = O(p \cdot S(n,p,d))$.*

Proof. In each step, a processor broadcasts at most one message to $p-1$ other processors. □

Note again that our focus is on optimizing work on the assumption that performing a task is substantially more costly that sending a message. It may also be interesting to optimize communication costs first.

7.5 Permutation Algorithms Family PA

In this section we present and analyze a family of algorithms that are simpler than algorithms DA and that directly rely on permutation schedules. Two algorithms are randomized (algorithms PaRan1 and PaRan2), and one is deterministic (algorithm PaDet).

7.5.1 Algorithm Specification

The common pattern in the three algorithms is that each processor, while it has not ascertained that all tasks are complete, performs a specific task from its local list and broadcasts this fact to other processors. The known complete tasks are removed from the list. The code is given in Figure 7.4. The common code for the three algorithms is in lines 00-29.

The three algorithms differ in two ways:

1. The initial ordering of the tasks by each processor, implemented by the call to procedure ORDER on line 20.
2. The selection of the next task to perform, implemented by the call to function SELECT on line 24.

We now describe the specialization of the code made by each algorithm (the code for ORDER+SELECT in Figure 7.4).

```
00 use package ORDER+SELECT                          % Algorithm-specific procedures
01 type TaskId : [n]
02 type TaskList : list of TaskId
03 type MsgBuff : set of TaskList

10 for each processor pid = 1 to p begin
11   TaskList Tasks_pid init [n]
12   MsgBuf B                                        % Incoming messages
13   TaskId tid                                      % Task id; next to done

20   ORDER(Tasks_pid)
21   while Tasks_pid ≠ ∅ do
22         receive B                                 % Deliver the set of received messages
23         Tasks_pid := Tasks_pid − (⋃_{b∈B} b)       % Remove tasks
24         tid := SELECT(Tasks_pid)                  % Select next task
25         perform Task(tid)
26         Tasks_pid := Tasks_pid − {tid}             % Remove done task
27         broadcast Tasks_pid                       % Share the news
28   od
29 end.
```

```
40 package ORDER+SELECT                              % Used in algorithm PARAN1
41 list Ψ = ⟨TaskList π_r | 1 ≤ r ≤ p  ∧  π_r = random list of [n]⟩
42  % Ψ is a list of p random permutations
43  procedure ORDER(T) begin  T := π_pid  end
44  TaskId function SELECT(T) begin  return(T(1))  end
```

```
50 package ORDER+SELECT                              % Used in algorithm PARAN2
51  procedure ORDER(T) begin  no-op  end
52  TaskId function SELECT(T) begin  return(random(T))  end
```

```
60 package ORDER+SELECT                              % Used in algorithm PADET
61 const list Ψ = ⟨TaskList π_r | 1 ≤ r ≤ p  ∧  π_r ∈ S_n⟩
62  % Ψ is a fixed list of p permutations
63  procedure ORDER(T) begin  T := π_pid  end
64  TaskId function SELECT(T) begin  return(T(1))  end
```

Fig. 7.4. Permutation algorithm and its specializations for PARAN1, PARAN2, and PADET ($p \geq n$).

As with algorithm DA, we initially consider the case of $p \geq n$. The case of $p < n$ is obtained by dividing the n tasks into p jobs, each of size at most $\lceil n/p \rceil$. In this case we deal with jobs instead of tasks in the code of permutation algorithms.

Randomized algorithm PaRan1**.** The specialized code is in Figure 7.4, lines 40-44. Each processor *pid* performs tasks according to a local permutation π_{pid}. These permutations are selected uniformly at random at the beginning of computation (line 41), independently by each processor. We refer to the collection of these permutation as Ψ. The drawback of this approach is that the number of random selections is $p \cdot \min\{n, p\}$, each of $O(\log \min\{n, p\})$ random bits (we have $\min\{n, p\}$ above because when $p < n$, we use p jobs, each of size $\lceil n/p \rceil$, instead of n tasks).

Randomized algorithm PaRan2**.** The specialized code is in Figure 7.4, lines 50-52. Initially the tasks are left unordered. Each processor selects tasks uniformly and independently at random, one at a time (line 52). Clearly the expected work *ES* is the same for algorithms PaRan1 and PaRan2, however the (expected) number of random bits needed by PaRan2 becomes at most $ES \cdot \log n$ and, as we will see, this is an improvement.

Deterministic algorithm PaDet**.** The specialized code is in Figure 7.4, lines 60-64. We assume the existence of the list of permutations Ψ chosen per Corollary 7.12. Each processor *pid* permutes its list of tasks according to the local permutation $\pi_{pid} \in \Psi$.

7.5.2 Complexity Analysis

In the analysis we use the quantity t defined as $t = \min\{n, p\}$. When $n < p$, t represents the number of tasks to be performed. When $n \geq p$, t represents the number of jobs (of size at most $\lceil n/p \rceil$) to be performed; in this case, each task in Figure 7.4 represents a single job. In the sequel we continue referring to "tasks" only — from the combinatorial perspective there is no distinction between a task and a job, and the only accounting difference is that a task costs $\Theta(1)$ work, while a job costs $\Theta(\lceil n/p \rceil)$ work.

Recall that we measure global time units according to the time steps defined to be the smallest time between any two clock-ticks of any processor (Section 7.1). Thus during any d global time steps no processor can take more than d local steps.

For the purpose of the next lemma we introduce the notion of adversary $\mathcal{A}_D^{(d,\sigma)}$, where σ is a permutation of n tasks. This is a specialization of adversary $\mathcal{A}_D^{(d)}$ that schedules the asynchronous processors so that each of the n tasks is performed for the first time in the order given by σ. More precisely, if the execution of the task σ_i is completed for the first time by some processor at the global time τ_i (unknown to the processor), and the task σ_j, for

any $1 \leq i < j \leq n$, is completed for the first time by some processor at time τ_j, then $\tau_i \leq \tau_j$. Note that any execution of an algorithm solving the *Do-All* problem against adversary $\mathcal{A}_D^{(d)}$ corresponds to the execution against some adversary $\mathcal{A}_D^{(d,\sigma)}$ for the specific σ.

Lemma 7.19. *For algorithms* PaDet *and* PaRan1*, the respective total-work and expected total-work is at most* (d)-Cont(Ψ) *against adversary* $\mathcal{A}_D^{(d)}$.

Proof. Suppose processor i starts performing task z at (real) time τ. By the definition of adversary $\mathcal{A}_D^{(d)}$, no other processor successfully performed task z and broadcast its message by time $(\tau - d)$. Consider adversary $\mathcal{A}_D^{(d,\sigma)}$, for any permutation $\sigma \in \mathcal{S}_n$.

For each processor i, let J_i contain all pairs (i, r) such that i performs task $\pi_i(r)$ during the computation. We construct function L from the pairs in the set $\bigcup_i J_i$ to the set of all d-lrm's of the list $\sigma^{-1} \circ \Psi$ and show that L is a bijection. We do the construction independently for each processor i. It is obvious that $(i, 1) \in J_i$, and we let $L(i, 1) = 1$. Suppose that $(i, r) \in J_i$ and we defined function L for all elements from J_i less than (i, r) in lexicographic order. We define $L(i, r)$ as the first $s \leq r$ such that $(\sigma^{-1} \circ \pi_i)(s)$ is a d-lrm not assigned by L to any element in J_i.

Claim. *For every* $(i, r) \in J_i$*,* $L(i, r)$ *is well defined.*

For $r = 1$ we have $L(i, 1) = 1$. For the (lexicographically) first d elements in J_i this is also easy to show. Suppose L is well defined for all elements in J_i less than (i, r), and (i, r) is at least the $(d+1)$st element in J_i. We show that $L(i, r)$ is also well defined. Suppose, to the contrary, that there is no position $s \leq r$ such that $(\sigma^{-1} \circ \pi_i)(s)$ is a d-lrm and s is not assigned by L before the step of the construction for $(i, r) \in J_i$. Let $(i, s_1) < \ldots < (i, s_d)$ be the elements of J_i less than (i, r) such that $(\sigma^{-1} \circ \pi_i)(L(i, s_1)), \ldots, (\sigma^{-1} \circ \pi_i)(L(i, s_d))$ are greater than $(\sigma^{-1} \circ \pi_i)(r)$. They exist from the fact, that $(\sigma^{-1} \circ \pi_i)(r)$ is not a d-lrm and all "previous" d-lrm's are assigned by L. Let τ_r be the global time when task $\pi_i(r)$ is performed by i. Obviously task $\pi_i(L(i, s_1))$ has been performed at time that is at least $d+1$ local steps (and hence also global time units) before τ_r. It follows from this and the definition of adversary $\mathcal{A}_D^{(d,\sigma)}$, that task $\pi_i(r)$ has been performed by some other processor in a local step, which ended also at least $(d + 1)$ time units before τ_r. This contradicts the observation made at the beginning of the proof of lemma. This proves the claim.

That L is a bijection follows directly from the definition of L. It follows that the number of performances of tasks – equal to the total number of local steps until completion of all tasks – is at most (d)-Cont(Ψ, σ), against any adversary $\mathcal{A}_D^{(d,\sigma)}$. Hence total work is at most (d)-Cont(Ψ) against adversary $\mathcal{A}_D^{(d)}$. □

Now we give the result for total-work and message complexities for algorithms PaRan1 and PaRan2.

Theorem 7.20. *Algorithms* PaRan1 *and* PaRan2*, under adversary* $\mathcal{A}_D^{(d)}$*, perform expected total-work*

$$ES(n,p,d) = O(n \log t + p \min\{n,d\} \log(2 + n/d))$$

and have expected message complexity

$$EM(n,p,d) = O(n\ p \log t + p^2 \min\{n,d\} \log(2 + n/d))\ .$$

Proof. We prove the work bound for algorithm PaRan1 using the random list of schedules Ψ and Theorem 7.11, together with Lemma 7.19. If $p \geq n$ we obtain the formula $O(n \log n + p \min\{n,d\} \log(2+n/d))$ with high probability, in view of Theorem 7.11, and the obvious upper bound for work is np. If $p < n$ then we argue that $d' = \lceil p\ d/n \rceil$ is the upper bound, in terms of the number of "job units", that it takes to deliver a message to recipients, and consequently we obtain the formula

$$O(p \log p + p\ d' \log(2 + p/d')) \cdot O(n/p) = O(t \log p + p\ d \log(2 + n/d)),$$

which, together with the upper bound $n\ p$, yields the formula

$$O(n \log p + p \min\{n,d\} \log(2 + n/d)).$$

Since the only difference in the above two cases is the factor $\log n$ that becomes $\log p$ in the case where $p < n$, we conclude the final formula for work. All these derivations hold with the probability at least $1 - e^{-t \ln t \cdot \ln(7/e^2) - p}$. Since the work can be in the worst case $n\ p$ with probability at most $e^{-t \ln t \cdot \ln(7/e^2) - p}$, this contributes at most the summand n to the expected work.

Message complexity follows from the fact that in every local step each processor sends $p - 1$ messages. The same result applies to PaRan2 (this is given as an observation in the description of the the algorithm.) □

Next is the result for total-work and messages for algorithm PaDet.

Theorem 7.21. *There exists a deterministic list of schedules* Ψ *such that algorithm* PaDet*, under adversary* $\mathcal{A}_D^{(d)}$*, performs total-work*

$$S(n,p,d) = O(n \log t + p \min\{n,d\} \log(2 + n/d))$$

and has message complexity

$$M(n,p,d) = O(n\ p \log t + p^2 \min\{n,d\} \log(2 + n/d))\ .$$

Proof. The result follows from using the set Ψ from Corollary 7.12 together with Lemma 7.19, using the same derivation for work formula as in the proof of Theorem 7.20. Message complexity follows from the fact, that in every local step each processor sends $p - 1$ messages. □

We now specialize Theorem 7.20 for $p \leq n$ and $d \leq n$ and obtain our main result for algorithms PaRan1 and PaRan2.

Corollary 7.22. *Algorithms* PaRan1 *and* PaRan2*, under adversary* $\mathcal{A}_D^{(d)}$*, perform expected total-work*

$$ES(n, p, d) = O(n \log p + p \; d \log(2 + n/d))$$

and have expected message complexity

$$EM(n, p, d) = O(n \; p \log p + p^2 d \log(2 + n/d))$$

for any $d < n$*, when* $p \leq n$*.*

Finally we specialize Theorem 7.21 for $p \leq n$ and $d \leq n$ and obtain our main result for algorithm PaDet.

Corollary 7.23. *There exists a list of schedules* Ψ *such that algorithm* PaDet *under adversary* $\mathcal{A}_D^{(d)}$ *performs work*

$$S(n, p, d) = O(n \log p + p \; d \log(2 + n/d))$$

and has message complexity

$$M(n, p, d) = O(n \; p \log p + p^2 d \log(2 + n/d)),$$

for any $d \leq n$*, when* $p \leq n$*.*

7.6 Open Problems

In this chapter we presented the message-delay-sensitive lower and upper bounds for the *Do-All* problem for asynchronous processors. One of the two deterministic algorithms relies on large permutations of tasks with certain combinatorial properties. Such schedules can be constructed deterministically in polynomial time, however the efficiency of the algorithms using these constructions is slightly detuned (polylogarithmically). This leads to the open problem of how to construct permutations with better quality and more efficiently.

There also exists a gap between the upper and the lower bounds shown in this chapter. It will be very interesting to narrow the gap.

The focus of this chapter is on the work complexity. It is also important to investigate algorithms that simultaneously control work and message complexity.

Lastly, we have used the omniscient adversary definition. The analysis of complexity of randomized algorithms against an oblivious adversary is also an interesting open question.

7.7 Chapter Notes

In the message-passing settings, the *Do-All* problem has been substantially studied for synchronous failure-prone processors under a variety of assumptions, e.g., [15, 16, 20, 30, 25, 38, 44]. However there is a dearth of efficient asynchronous algorithms. The presentation in this paper is based on a paper by Kowalski and Shvartsman [77]; the proof of Lemma 7.5 appears there.

A lower bound $\Omega(n + p \log p)$ on work for algorithms in the presence of processor crashes and restarts was shown by Buss, Kanellakis, Ragde, and Shvartsman [14]. The strategy in that work is adapted to the message-passing setting without failures but with delays by Kowalski, Momenzadeh, and Shvartsman [74], where Theorem 7.3 is proved.

The notion of *contention* of permutations was proposed and studied by Anderson and Woll [5]. Lemmas 7.8 and 7.9 appear in that paper [5]. Algorithms in the family DA are inspired by the shared-memory algorithm of the same authors [5]. The notion of the *left-to-right maximum* is due to Knuth [71] (vol. 3, p. 13). Kowalski, Musial, and Shvartsman [75] explore ways of efficiently constructing permutations with low contention. They show that such permutations can be constructed deterministically in polynomial time, however the efficiency of the algorithms using these constructions is slightly detuned.

For applications of Chernoff bounds see Alon and Spencer [4].